6-2014

Ranking-based approaches for localizing faults

Lucia LUCIA
Singapore Management University, lucia.2009@phdis.smu.edu.sg

Citation

LUCIA, Lucia. Ranking-based approaches for localizing faults. (2014). 1-157.
Available at: https://ink.library.smu.edu.sg/etd_coll/107

Ranking-Based Approaches for Localizing Faults

by

Lucia

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

David LO (Chair)
Assistant Professor of Information Systems
Singapore Management University

Lingxiao JIANG
Assistant Professor of Information Systems
Singapore Management University

Feida ZHU
Assistant Professor of Information Systems
Singapore Management University

Julia LAWALL
Director of Research
Inria - Laboratoire d'Informatique de Paris 6

Singapore Management University
2014

Ranking-Based Approaches for Localizing Faults

Lucia

Abstract

A fault is the root cause of program failures where a program behaves differently from the intended behavior. Finding or localizing faults is often laborious (especially so for complex programs), yet it is an important task in the software lifecycle. An automated technique that can accurately and quickly identify the faulty code is greatly needed to alleviate the costs of software debugging.

Many fault localization techniques assume that faults are localizable, i.e., each fault manifests only in a single or a few lines of code that are close to one another. To verify this assumption, we study how faults spread across program elements. We find that most faults are localizable within a few lines of code or a few methods, while around 30% of the faults manifest in a single line of code.

Spectrum-based fault localization approach is a lightweight approach that analyzes execution traces to highlight top-most suspicious program elements (i.e., statement, blocks, etc.) for inspection by developers. Our fault localization technique can be categorized into spectrum-based approach. Our technique localizes faults by measuring the strength of the relationship between the execution of a program element and the occurrence of a program failure. Various association measures are proposed in the domains of statistics and data mining to quantify the strength of the relationship between two variables of interest. However, their effectiveness in localizing faults is not well studied. We investigate the effectiveness of 40 association measures in localizing faults in single-bug and multiple-bug programs. Some of the measures achieve smaller percentage of code inspected on average than the two well-known spectrum-based techniques, namely Ochiai and Tarantula, while a number of the measures are comparable to Ochiai and Tarantula.

Different fault localization techniques have different effectiveness in localizing faults for different buggy programs. We propose an approach called *Fusion Localizer* to leverage their differences and boost the effectiveness in localizing faults. Our approach combines scores or ranking information produced by existing spectrum-based fault localization techniques in particular, 40 association measures, Ochiai, and Tarantula, to inexpensively rank the faulty program elements using data fusion methods that have been studied in the domain of information retrieval. Our evaluation demonstrates that our approach can significantly improve the effectiveness of existing state-of-the-art fault localization techniques.

The above approaches localize potential faulty elements using execution traces. However at times, full execution traces are not available for debugging. Code clones (i.e., pieces of similar code) have been shown to be useful for detecting bugs because the inconsistent changes among clones in a clone group may indicate potential bugs. However, clone-based bug detection techniques suffer from an excessive number of false positives. Our technique ranks the anomaly reports that contain bugs earlier in the list as compared to the original list. By actively and incrementally incorporating user feedback to iteratively refine our classification model and reorder the anomaly reports, our technique can successfully reduce the false positive rate.

In summary, this dissertation has empirically demonstrated the need of and proposed a number of novel ranking-based approaches for localizing faults, which advances the previous state-of-the-art.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor, Prof. David Lo. His guidance, never ending support, and valuable advice mean a lot to me during my PhD journey. I would also like to thank my co-advisor, Prof. Lingxiao Jiang for the fruitful discussions and the valuable advice. Both of them are my role models. Their intelligence and excellent personalities have inspired me to keep improving my knowledge and keep the good attitude in life. Many thanks to my defence committee members: Prof. Feida Zhu, and Prof. Julia Lawall for the valuable comments and feedback for this dissertation.

Many thanks to Dr. Foyzur Rahman and Prof. Prem Devanbu for not only providing us a database containing the Lucene bugs with the associated fixes which we evaluate in this dissertation, but also for the fruitful discussions and the valuable insights during the collaboration. Many thanks to Prof. Artur Dubrawski as my research advisor while I was in CMU. It was such precious experience to work with Prof. Dubrawski and his analytic team. Also, many thanks to Aditya Budi, Wang Shaowei, Ferdian Thung, and Tien-Duy B. Le for the great collaboration.

Last but not least, many thanks to my family for your never ending love to me. Many thanks to my friends in Singapore Management University and many great friends out there. My PhD life becomes so precious, not only we can learn from one another, but we can also have fun together. A special thank to my best friend, Swee Won for always being there for me and providing valuable comments to this dissertation. Above all, thanks be to God for His blessing and making everything possible and beautiful in its time.

Dedication

for my beloved mother, in memory of my father, my lovely brothers and sisters

for never ending love and support...

List of Publications

Conference Papers

Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive Evaluation of Association Measures for Fault Localization. In *Proceedings of the 26th IEEE International Conference of Software Maintenance*, Romania, 2010.

Lucia, F. Thung, D. Lo, and L. Jiang. Are Fault Localizable? In *Proceedings of the 9th IEEE Working Conference on Mining Repositories*, Switzerland, 2012.

Lucia, D. Lo, L. Jiang, and A. Budi. Active Refinement of Clone Anomaly Reports. In *Proceedings of the 34th IEEE International Conference of Software Systems*, Switzerland, 2012.

Lucia, D. Lo, and X. Xia. Fusing Fault Localizers. In *Proceedings of 29th IEEE/ACM International Conference on Automated Software Engineering*, Sweden, 2014. (under submission)

Journal Papers

Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended Comprehensive Study of Association Measures for Fault Localization. *Journal of Software Evolution and Process*, 26(2), 2014, pp. 172-219.

Chapter 1

Introduction

1.1 Automatic Software Fault Localization

Developing a software that is free from bugs, is a very difficult requirement to be met. In fact, it is almost impossible for most software, especially complex ones. Software bugs cause financial loss to both users and developers. As reported by the US National Institute of Standards and Technology, software bugs cost the US economy 59.5 billion dollars annually [134].

A software fault (or bug) is the source of an error in a software or program, while the error itself is a state of the program that causes program failures [15]. A program failure is an event when the program behaves differently from what was intended [15]. When a program failure occurs, manually finding the root cause of the failure is often a laborious task, especially when the root cause of the failure is far away from the failure point. An increase in the size and number of features provided by a software system in turn increases the complexity of the system as well as the possibility of defects manifesting in the software [25]. Indeed, testing and debugging activities account for 30 to 90 percent of the labor expended for a project [20, 49]. Therefore, an automated technique that can accurately and quickly identify the faulty code is greatly needed to reduce the cost of software debugging.

Automated software debugging has attracted much research interest. Many approaches have been proposed to automate software debugging activities, especially in localizing root causes of failures (i.e., faults) in programs. Some approaches are tailored for one kind of bugs, while others are general-purpose.

There are many approaches that locate a specific kind of bugs; some examples include those that are designed to specifically detect deadlock bugs [23, 38, 111], data race bugs [28, 38, 99, 111, 123], atomicity violation bugs [40, 87, 145], etc. Deadlock bugs occur when two or more operations circularly wait for one another to release the acquired resource. Data race bugs occur when two conflicting executions access one shared variable without proper synchronization. Atomicity violation bugs occur when concurrent processes unexpectedly execute a critical code region at the same time.

Many other fault localization approaches are general purpose. They can catch various kinds of bugs and thus are more applicable to a wide range of software and problems that they face. This family can be further divided into those that primarily analyze textual information, execution traces, or source code. Many approaches have been proposed in the literature that belong to each of these sub-families. We highlight these approaches in the following paragraphs.

A number of approaches locate faults from the textual descriptions of bugs in bug reports that are submitted by users or developers during testing. The information described in these reports can be used to retrieve the failure-relevant code. Many of these text-based approaches employ information retrieval techniques to analyze information written in such reports and retrieve files that are similar to the reports [88, 100, 115, 126, 154]. The effectiveness of such approaches in retrieving the failure-relevant code depends on the quality of the information written in the bug reports. If a bug report poorly describes a bug, it will be hard to find buggy code from the description.

A number of approaches locate faults primarily by analyzing software execution traces. Model-based debugging techniques [39, 91–94] analyze execution traces by using logical reasoning over formal models of programs to accurately locate the faults. Even though these techniques are more accurate, they are often heavyweight because such reasoning is expensive. On the other hand, spectrum-based fault localization [3, 5, 26, 27, 31, 63, 65, 67, 84, 151, 153] is a lightweight fault localization approach, where program traces or abstractions of traces (called *program spectra*) are used to represent program behaviors and are analyzed to identify potentially faulty program elements (e.g., statements, basic blocks, functions, and components) by computing suspiciousness scores. Among spectrum-based fault localization approaches, some of them focus on finding a single bug in software, e.g., [31, 65], while others focus on directly finding multiple bugs in software, e.g., [5, 67]. Some others focus on extracting contextual information that can help developers in performing debugging tasks [26,63,153]. The accuracy of these approaches in locating faults is limited by the information contained in the program traces [5]. When a program failure occurs, the execution trace may be long and contain failure-irrelevant information. Also, full execution traces are often not available for debugging if programs fail in the users' sites.

A number of approaches primarily analyze source code and do not consider execution traces [43, 53, 64, 74, 81, 119]. Some of these approaches rely on information of files that are affected by bugs before, to predict future buggy files, e.g., [74]. Some others rely on a set of rules whose violations indicate bugs or code smells, e.g., [119]. Some others rely on detection of common patterns (e.g., software clones) and flag violations of these patterns (aka. anomalies) as potential bugs, e.g., [43,53,64]. One disadvantage of these approaches is the high false positive rate where many codes are predicted to have defects, while in fact there is no defect.

The above families of approaches have their own weaknesses and strengths. This dissertation focuses on improving the accuracy of general purpose fault localization approaches since they are applicable to a wide range of software and problems. In particular, we focus on lightweight fault localization techniques that can scale in analyzing software of various sizes. Specifically, we focus on improving state-of-the-art spectrum-based fault localization techniques and clone-anomaly-based fault localization techniques which have been shown promising in many past studies, e.g., [31, 43, 53, 64, 65].

Although fault localization approaches have improved much since they were first proposed, they are still far from perfect. The emerging information system technologies and the increasing program complexity further challenge their effectiveness. Thus, improving the effectiveness of these approaches remains an open research problem.

1.2 Research Problems

Most of the automated fault localization techniques attempt to pinpoint the location of bugs and expect developers to investigate a certain number of program elements to find faults. These techniques thus assume that faults are localizable, i.e., only one or a few lines of code that are close to one another are responsible for each fault. In reality, are faults localizable? This research question can impact the applicability of the fault localization techniques to reveal the faulty program elements.

Among many techniques to locate the root cause of program failures (i.e., fault localization techniques), spectrum-based fault localization techniques are often lightweight and have good accuracy. These techniques use program traces or abstractions of traces (called *program spectra*) to represent program behaviors. The likelihood that a program element is a faulty element is often inferred by comparing the spectra of correct and failed executions using statistical analysis [3,27,31,65,84]. Although they have good accuracy, the effectiveness of current state-of-the-art fault localization techniques is not yet perfect. Often, faulty program elements are ranked low in the returned ranked list. Therefore, the effectiveness of the state-of-the-art of fault localization techniques needs further improvement to minimize the debugging time.

There have been a number of studies proposed in the statistics and data mining community on measuring of association between two variables of interest since the early 20th century such as Yule's Q [148] and Yule's Y [149]. The properties, benefits and limitations of each measure have been investigated from a computational point of view [44, 132]. However, the effectiveness of various association measures in measuring the strength of the relationship between the execution of a program element and the occurrence of a program failure to localize faults has not been comprehensively evaluated. Some of the association measures have been studied for fault localizations, e.g. Jaccard [3, 4], Sorensen-Dice [4], Anderberg [4], Simple Matching [4], Rogers and Tanimoto [4], and Ochiai II [4]. In this work, we also evaluate the effectiveness of these measures because their effectiveness in localizing various kinds of bugs and localizing bugs in multiple-bug programs have not been evaluated.

Different spectrum-based fault localization techniques (e.g., association measures) could have different effectiveness in localizing bugs for different buggy programs. For a given buggy program, some techniques could have better effectiveness than others, and vice versa. The question is: could we leverage diversity of the techniques in localizing bugs such that we could better rank the faulty program elements as compared to individual techniques?

Wang et al. [137] propose a technique that linearly combines the scores of a number of association measures to rank faulty program elements at the top positions. Their approach generates a weight for each association measure using genetic algorithms based on a set of training data (i.e., program traces of the past program failures and the corresponding faults of the past failures). The effectiveness of the technique in localizing faults depends on the set of buggy programs used in the training set. It is possible that the bugs and buggy programs in the training set are not representative enough for the incoming buggy programs which could limit the effectiveness of the techniques. Furthermore, the training data is often unavailable in many cases such as new programs, developers do not store the traces of the past program failures, etc.

It is challenging to combine different fault localization techniques to better localize the bugs in programs. Not only do we need to handle different techniques, but we also need to handle different buggy programs and different bugs that manifest in the programs. Addressing these challenges is essential to effectively locate the bugs for different programs.

The techniques mentioned in the previous paragraphs require the availability of test cases to generate execution traces of the buggy programs; the traces are then used to justify the likelihood that a program element is the faulty element. In order to have useful program traces for localizing faults, at least one of the test cases used to test the program must be able to reveal program failures (i.e., the test case results in failure). However, obtaining such test cases can be difficult, moreover such test cases may be unavailable. Consider a case when a program failure occurs in the client's site; often, developers cannot obtain the test case that can reproduce the same program failure as what was reported by the client. Hence, we could not collect the execution traces that can be used to infer the faulty program elements. Thus, it would be useful if we have techniques that can locate the faulty program elements without the availability of test cases.

In the absence of test cases, we can analyze buggy programs to infer faulty program elements. Software clones have been shown to be useful for finding bugs [43, 53, 64, 68, 78, 81]. The inconsistent changes among clones in a clone group may indicate potential bugs. Clone-based anomaly detection techniques have been developed to look for the inconsistencies among code clones in every clone group (i.e., a group of code fragments similar to each other) and report them as anomalies (i.e., potential bugs). However, these techniques still produce an excessive number of false positive anomaly reports [64], which impede broad adoption of the techniques. The ability to present the anomaly reports that contain the bugs in early positions is important to improve the adaption of clone-based bug detection tools.

1.3 Contributions

This dissertation makes four contributions. The first contribution empirically studies how faults manifest in program elements of buggy programs. The second and third contributions are related to better ranking the faulty program element using spectrum-based fault localization approach. The last contribution aims to better rank clone anomaly reports. We briefly describe our contributions as follows.

1. This dissertation presents a study of how faults manifest across program elements in buggy programs. Hundreds of real faults in several software systems are evaluated and the assumption that faults are localizable within a few lines of code is studied. We report how faults spread across program elements in different levels of granularity.

2. In order to help developers to localize faults in programs, this dissertation assesses the likelihood that a program element is the root cause of a program failure by measuring the strength of the relationship between the execution of the program element and the occurrence of program failure. Various association measures have been proposed in the statistics and data mining literature to measure the strength of the relationship between two variables of interest. However, their effectiveness in localizing faults has not been well studied. We investigate the effectiveness of 40 association measures to rank faulty program elements and present our empirical evaluation of the effectiveness of the association measures in localizing bugs in single-bug and multiple-bug programs. Different types of bugs are also considered in our evaluation.

3. Different fault localization techniques could have different effectiveness in localizing faults. This dissertation aims to better localize the bugs by leveraging the diversity of existing spectrum-based fault localization techniques in particular, 40 association measures, Tarantula, and Ochiai. Our approach incorporates data fusion methods that have been studied in information retrieval

domain to rank faulty program elements higher in the list by combining the scores or ranks assigned to program elements by different fault localization techniques. Our approach is bug specific. It selects a different set of techniques to be combined for different buggy programs based on their program spectra, also our approach does not require any training data i.e., program spectra of the past program failures to select techniques to be fused.

4. In the absence of program traces, inconsistency among code clones or pieces of similar codes in a clone group have been shown to be useful as an indicator to detect bugs in the programs. This dissertation presents an approach that actively and incrementally incorporate users' feedback to refine the rank of clone anomaly reports produced by clone-based bug detection tools, so that the reports that contain the buggy clones would be presented to the users earlier than the default ordering.

1.4 Dissertation Outline

We present our contributions in four chapters in this dissertation. The structure of the dissertation is organized as follows. Chapter 2 discusses a number of research works related to our contributions. We perform an empirical study to investigate how faults manifest across program elements in buggy programs, which is presented in Chapter 3. Chapter 4 discusses the effectiveness of a number of association measures that have been introduced in statistics and data mining community to better rank the faulty program elements in single-bug or multiple-bug programs. In Chapter 5, we utilize the scores or the ranks produced by a number of measures presented in Chapter 4 to better rank the faulty program elements. We employ data fusion methods introduced in the domain of information retrieval to improve the effectiveness of the ranking results by ranking the faulty program elements in the top ranks. When there is no available test case for localizing faults, Chapter 6 presents a ranking approach to rank clone anomaly reports by incorporating user feedback so that the anomaly reports that contain true bugs can be ranked in the top positions for inspection by developers. Finally, Chapter 7 summarizes the contributions of this dissertation and possible future work.

Chapter 2

Literature Review

In this section, we discuss studies that are closely related to the contributions of this dissertation. The survey here is by no means a complete list of all related studies.

2.1 Empirical Studies on Bugs

Many researches have investigated the characteristics of bugs to obtain better understanding to automatically locate and fix bugs. Pan et al. analyze patterns of bug fixes [105]. They find that bug fixes can be classified into different families including: addition of pre-condition checks, removal of an if predicate, method call with different number of parameters, etc. Similarly, Ostrand and Weyuker [104], Perry and Stieg [107], and Leszak et al. [80] also study the characteristics of bugs. In this study, we perform an orthogonal study investigating the localizability of bugs.

Parnin and Orso perform user studies on the effectiveness of fault localization techniques and their results imply that the current techniques may not always be sufficient [106]. Our work as presented in Chapter 3 analyzes the bugs themselves without user evaluation, but the results also bear a similar implication.

There are also studies that investigate cases where multiple faults exist at the same time (e.g., [67]). Herzig and Zeller "untangle" changes in a commit that are unrelated to each other [56]. Our study in Chapter 3 assumes changes in one commit deal with one bug only, and empirically analyzes how widespread or localized the failure-relevant changes are.

2.2 Fault Localization

Recently, there have been many studies on fault localization and automated debugging. There are different ways to categorize these studies. Based on the data analyzed by the approaches, fault localization techniques can be classified into *spectrum-based* and *model-based*.

2.2.1 Spectrum-Based Fault Localization.

Spectrum-based fault localization techniques often use program spectra (i.e., program traces or abstractions of program traces) to represent program runtime behaviors in certain ways. Statistical analysis is often used to correlate program elements (e.g., statements, basic blocks, functions, and components) with program failures.

Many spectrum-based fault localization techniques [3, 31, 65, 66, 83, 116, 122, 141] take two sets of spectra as inputs, one for correct executions and the other for failed executions, and report potential locations where root causes of program failures (i.e., faults) may reside. Given a failed program spectrum and a set of correct spectra, Renieris and Reiss present a fault localization tool WHITHER [116] that compares the failed execution to the nearest correct execution and reports the most suspicious locations in the program. Liblit et al. propose a technique to search for predicates whose true evaluation correlates with failures [83]. Chao et al. extend the work by incorporating information on the outcomes of multiple predicate evaluations in a program run in their tool called SOBER [31]. Wong et al., [141] propose a method called DStar that modifies a similarity-based coefficient (i.e., Kulczynski coefficient) to better localize the bugs as compared to Taratula [65, 66], Ochiai [3], and 12 similarity-based coefficients. All of these techniques need to compare spectra of failed and correct executions in some way. Evaluating the effectiveness of various association measures can complement all of these techniques by helping to quickly locate the most failure-relevant program elements.

Artzi et al. propose a tool named Apollo that can generate test cases to expose failures in web applications, then localize bugs that cause the failures [13]. Their fault localization technique uses a spectrum-based fault localization technique namely, Tarantula and a technique that keep information about which program statements are potentially responsible for producing a particular part of an output (e.g., a table in an HTML document). This approach is possible for web applications but may not be applicable to other applications as the output is often not decomposable into parts (it could be a single number) and the number of program statements that are potentially responsible for producing an output is often large. Artzi et al. extend their work by evaluating the effectiveness of several test generation techniques in generating enough test cases for localizing faults in web applications with the help of a spectrum-based fault localization technique, namely Ochiai [12].

Bandyopadhyay and Ghosh study how the properties of faults affect the effectiveness of Tarantula [18]. Three properties, namely accessibility, original state failure condition, and impact, are investigated. Our study presented in Chapter 4 also investigates the effectiveness of various fault localization techniques for different kinds of bugs. However, we consider a different categorization of faults – our categorization is based on the empirical study performed by Pan et al. on bug fixes [105].

Other spectrum-based techniques [48,61,130,152] only use failed executions as the input and systematically alter the program structure or program runtime states to locate faults. Zhang et al. [152] search for faulty program predicates by switching the states of program predicates at runtime. Sterling and Olsson use the concept of program chipping [130] to automatically remove parts of a program so that the part that contributes to the failure becomes more apparent.Gupta et al. [48] work on program dependency graphs and use the intersection of forward and backward program slices to reduce the sizes of failure-relevant code for further inspection. Jeffrey et

Dataset	LOC	Papers	Dataset	LOC	Papers
SumPowers	27	[92,94]	Power	763	[83]
BinSearch	29	[92,94]	Compress	1590	[83]
BubbleSort	29	[92,94]	Bh	2053	[83]
Hamming	48	[92,94]	Webchess v.0.9.0	2226	[12,13]
Adder	49	[92,94]	Tetris	2403	[106]
Permutation	54	[92,94]	Schoolmate v.1.5.4	4263	[12,13]
Binomia	80	[92,94]	Gzip	5680	[5,21]
Polynom	105	[92,94]	Space	6218	[4,5,65,66]
Tcas	141	[5,6,18,21,31,61,65,91,92,94,116]	NanoXML	7646	[106,122]
Schedule	292	[3,5,6,18,21,31,48,61,65,116,122]	Phpsysinfo v.2.5.3	7745	[12,13]
Schedule2	301	[3,5,6,18,21,31,48,61,65,116,122]	Li	7761	[83]
Treeadd	385	[83]	Grep	10068	[21,152]
Perimeter	395	[83]	Flex	10459	[21,152]
Print_token2	399	[3,5,6,18,21,31,48,61,65,116,122]	Sed	14427	[5,21]
Tot_info	440	[3,5,6,18,21,31,61,65,116,122]	XMLsecurity	16800	[122]
Print_token	478	[3,5,6,18,21,31,48,61,65,116,122]	Bc-1.06	17042	[152]
Replace	512	[3,5,6,18,21,31,48,61,65,116]	Tar-1.13.25	27137	[152]
Emad	557	[83]	Go	29315	[83]
Rest	617	[83]	Ijpeg	31371	[83]
Bisort	707	[83]	Make	35545	[152]
Health	725	[83]	JABA	37966	[122]
Faqforge v.1.3.2	734	[12,13]			

Table 2.1: Subject programs used in past fault localization studies

al. use a value profile based approach to rank program statements according to their likelihood of being faulty [61]. These fault localization techniques do not compare the spectra of failed and correct executions. Thus, association measures are generally not applicable to them.

Instead of empirically studying the effectiveness of a fault localization technique on various faults, Naish et al. theoretically analyze a number of formulas that can be used to compute suspiciousness scores of program elements in "idealized conditions" [97]. Their study is later extended by Xie et al. which theoretically analyze 30 formulas and group them into equivalence classes [143]. They prove that, under some conditions, two families (i.e., ER1 and and ER5) outperform the rest. These families include five formulas: Naish1, Naish2, Wong1, Russel & Rao, and Binary.

Instead of manually designing fault localization formulas, Yoo generates a number of formulas using Genetic Programming [147]. The effectiveness of these formulas are then theoretically studied by Xie et al. [144]. They find that GP13 and three other formulas, i.e., GP02, GP03, and GP19, are the best ones for fault localization. Instead of generating new formulas, we combine multiple formulas to improve their effectiveness in localizing faults.

The above techniques [3, 31, 65, 66, 83, 97, 116, 122, 141, 143, 144, 147] analyze a single type of program spectra to assign suspiciousness scores to program elements. Differently, Santelices et al. collect *a number of program spectra types* (i.e., statement, branches, and du-pair spectra), where for each program spectra, *a single fault localization technique* (i.e., Ochiai) is used to assign a suspiciousness score for each program element [122]. For each program element, they combine its suspiciousness scores produced by analyzing different types of program spectra. Different from Santelices et al.'s technique, our approach presented in Chapter 5 analyzes *a single program spectra type* (i.e., block spectra) using *a number of fault localization techniques*. Thus, we are investigating orthogonal directions and it is possible to combine our approach and Santelices et al.'s approach in a future work.

Our approach presented in Chapter 5 is closely related to the technique proposed by Wang et al. [137]. For every program element, their technique linearly combines the scores of a number of association measures where each measure obtains a weight which is computed based on a set of training data using a Genetic Algorithm. Different from their technique, our approach does not require any training data which makes our technique applicable to new programs or when program spectra of relevant past program failures are unavailable. Also, their technique applies the same set of measures and the same set of weights to localize bugs for different buggy programs. Different from Wang et al.'s technique, our technique selects a different set of techniques or set of weights to localize faults for each buggy program based on its program spectra. While Wang et al.'s approach is one-size-fits-all, our approach is bug specific. Table 2.1 shows the list of programs that have been used by those past fault localization studies.

2.2.2 Model-Based Fault Localization.

Compared with spectrum-based techniques, model-based debugging techniques [39, 91–94] are often more accurate, but heavyweight since they are based on more expensive logic reasoning over formal models of programs. Many static and dynamic analysis techniques [82,89,131] can be classified as model-based debugging.

Recently, there have been a number of techniques proposed to combine spectrum-based with model-based techniques to better localize the bugs. Abreu et al. propose a framework called BARINEL that combines spectrum-based fault localization and model-based debugging to localize single and multiple bugs in programs, and found that the approach is more accurate and heavyweight than spectrum-based fault localization [5]. Baah et al. extend spectrum-based fault localization model by incorporating data and control dependencies to narrow down the potential faulty elements, hence to improve the effectiveness of the techniques in localizing bugs [16]. Gopinath et al. combine spectrum-based technique with specification-based analysis to better localize faults [47]. Digiuseppe and Jones generate description of faults by combining spectrum-based techniques with natural language techniques that can extract features from the source code [98].

Although few model-based techniques have employed the concept of failure association, incorporating association measures investigated in this study into program models can be a future direction to improve the performance of model-based debugging techniques. In this dissertation, we focus on comparisons with two well-known spectrum-based fault localization techniques, namely Tarantula [65,66] and Ochiai [3,5,6]. We evaluate 40 association measures and find promising ones for fault localization. Our work presented in Chapter 5 aims to improve the effectiveness of spectrum-based fault localization which could benefit the techniques that incorporate spectrum-based fault localization.

2.3 Code Clone Analysis and Clone-Based Bug Detection

Code clones have been widely studied in the literature. Some studies focus on detection of code clones, based on similarities among strings, tokens, syntax trees, dependency graphs, and even functionalities [17,19,69,76,81,110,135]. Clones are traditionally thought of as harmful, and techniques have been proposed to reduce clones [60, 114]. On the other hand, some studies show that clones can be useful and necessary [70, 72]. Then, instead of reducing clones, some studies investigate techniques to track and manage code clones [36,59,101].

One important use of code clones is to detect bugs. A number of studies detect bugs by detecting inconsistencies among clones [43, 53, 64, 68, 81]. Such inconsistency-based detection of clone-related bugs often produces many false positives, and uses various filtering rules to reduce false positives. However, even with the most recent filtering techniques, such as ones based on textual similarity and sequence alignment [43], false positive rates remain high. Compared with these studies that use filtering-based approaches to remove reports, which may cause false negatives, our approach actively and incrementally refines and re-ranks anomaly reports based on user feedback without removing any report. The code features used in our re-ranking are also different from those used in other papers. Our work is not an alternative, but rather a complement of others. Filtering-based approaches (which still leave many false positives behind) may be applied first, then our work refines the filtered reports as users take actions, e.g., to fix an anomaly if it is a true positive.

2.4 Bug Prediction and Triage

Many studies aim to predict whether certain code changes or files contains faults. Kim *et al.* [73,74] use bug history to predict faults. Ruthruff *et al.* [119] use logistic regression models from historical data to predict whether a warning generated by FindBugs is actionable. Zimmermann *et al.* [155] have studied the accuracies of bug prediction models that may be used across various projects in various domains.

Other studies aim to reduce similar bug reports or prioritize bug reports. Podgurski *et al.* [109] group software failures with similar symptoms together. Kremenek and Engler [77] propose *z-ranking* to order bug reports produced by a static program checking analysis tool. Heckman and William [55] propose FAULTBENCH, a benchmark for evaluating alert prioritization and classification techniques. These ranking models only perform reordering of bug reports once.

Our approach presented in Chapter 6 is different from the aforementioned studies in several aspects. We focus on anomaly reports generated by a *clone-based anomaly* detection *tool*, instead of reports from *users*. We *reorder* anomaly reports, while most other studies *filter* reports. Filtering anomaly reports carries a risk of removing true positives. Filtering and reordering are complementary as we could first filter and then re-order anomaly reports. Some studies leverage *historical* data to prioritize anomaly reports, while we leverage *immediate user feedback* to iteratively prioritize clone-based anomaly reports.

Chapter 3

Empirical Study on Fault Locations

In this chapter, we study a certain characteristic of faults in programs in order to have better idea about the capability that is expected from automatic software fault localization techniques. In particular, we perform an empirical study on whether real faults are localizable (i.e., whether it spans a few lines of code, methods, or classes).

We organize this chapter as follows. Section 3.1 presents the motivation and main contribution of this chapter. We present subject programs that we evaluate for this study and the setup of our empirical study in Section 3.2. Section 3.3 presents our findings and discusses their implications. Section 3.4 concludes the study.

3.1 Are Faults Localizable?

Bugs are one of the major contributors to high software cost. Many automated debugging techniques have been proposed to reduce the cost of debugging. Fault localization techniques aim to pinpoint program elements responsible for a fault. Many of these techniques analyze program spectra (i.e., a set of profiles of correct executions and failures), with the goal of highlighting likely faulty program elements [30, 31, 65, 83, 116, 150].

Fault localization techniques often assume that faults are localizable, i.e., a fault is often confined to one or a few lines of code that are close to each other in a software system. Most fault localization techniques would rank program elements according to their suspiciousness (i.e., the likelihood that a program element is the root cause of a program failure) and expect developers to traverse this list of program elements and be able to decide whether an element contains a fault by just inspecting that element. Past studies on fault localization often use faults that are injected to only one or a few locations, making the evaluation of the techniques biased.

The question is whether it is indeed the case that faults are confined to a few lines of code in real systems. *Are faults localizable?* This research question has important implications. If faults turn out to be non-localizable, we may then need to re-consider the applicability of fault localization and design new approaches to aid developers in debugging non-localizable faults.

In this chapter, we perform an empirical study by analyzing software from a public bug repository–iBugs [33] and the JIRA repository of Lucene. We consider hundreds of real bugs in three real systems: AspectJ, Rhino, and Lucene, and investigate how localized or spread-out the locations of buggy program elements are.

The contributions of this work are as follows:

- We highlight an important research question on whether bugs are localizable in real software.
- We present an empirical study on three Java programs and note that many faults are not localized.
- We analyze whether severe faults are localizable.

3.2 Empirical Study Setup

3.2.1 Evaluation Dataset

We analyze the locality of bugs in two Java programs (Rhino and AspectJ) from iBugs repository [33] and a third Java program Lucene. Rhino is a Javascript interpreter written in Java with code size of about 49 kLOC. There are 32 buggy versions of Rhino in iBugs. AspectJ is a compiler for aspect-oriented programming in Java with code size of about 75 kLOC, and iBugs contains 350 of its buggy versions. Lucene is a text engine library with code size of about 88 kLOC (version 2.9).

The iBugs repository stores both pre-fix versions that contain bugs (buggy versions) and the corresponding post-fix versions that fix the bugs. Each of the buggy versions is assumed to contain one bug that may span across multiple lines in multiple files. Information about each fix is also provided, e.g., the numbers of changed lines, changed methods, and changed files, and the severity level of the bug. We also know which lines or files are changed based on the differences between the pre-fix and the post-fix versions. Similar information about bugs and corresponding fixes in Lucene has been collected from JIRA [2] by another research team at UC Davis.

3.2.2 Extracting Faults from Changes

In this work, we are concerned with program elements that are *responsible*, or are the root causes of a bug (i.e., faults). The datasets from iBugs and JIRA include bug fixes and enhancements. We exclude those corresponding to enhancements, those that do not contain any severity information, and bug fixes that only fix test code and comments. We end up with 374 bugs and their fixes: 32 for Rhino, 290 for AspectJ, and 52 for Lucene.

For these bug fixes, the iBugs and JIRA datasets include lines added, deleted, and modified in the commit that *fixes the bug*. Not all these lines correspond to the root causes of the bug and thus we could not simply count them as the number of faulty lines. First, these lines are the *treatment* of the fault and we need to recover the fault from them. Second, previous studies, e.g., [71], show that not all changes are essential – many are simple refactorings that do not change the behavior of a program. To recover faults from their treatments, we perform a manual inspection on the iBugs and JIRA data.

3.2.3 Research Questions on Locality of Faults

First, we define the *locality* of a bug based on faulty program elements. We consider program elements at three levels of granularity, including lines, methods, and files. We also consider the spatial distances among the program elements. The following is a list of locality definitions that we use in our empirical evaluation.

D1 Considering a line of code as a program element, we define the locality L of a buggy version v as follows:
$L_{D1}(v) =$ the number of faulty lines.

D2 Considering a method as a program element, we define the locality L of a buggy version v as follows:
$L_{D2}(v) =$ the number of faulty methods.

D3 Considering a file as a program element, we define the locality L of a buggy version v as follows:
$L_{D3}(v) =$ the number of faulty files.

D4 Considering the spatial distances among the faulty lines, we define the locality L of a buggy version v based on the number of faulty files n_f and the gaps among the faulty lines in every file $G_{line}(\mathit{file})$:
$L_{D4}(v) = (\Sigma_{file} G_{line}(\mathit{file}) + n_f - 1) \times n_f$, where $G_{line}(\mathit{file})$ is the distance between the first and last faulty lines in the file.

Based on the locality definitions, we consider the following research questions:

RQ1 How wide or localized is the locality of bugs? How many bugs could be localized to a few program elements?

RQ2 Are severe bugs localizable?

The following section evaluates the locality of the 374 real bugs. We also discuss the answers for the above research questions.

3.3 Results and Discussion

3.3.1 RQ1: Are Faults Localizable?

We evaluate how localized the faults are in term of the number of faulty lines, methods, and files. Figures 3.1, 3.2, and 3.3 show the proportion of faults that are localizable up to a certain number of faulty lines, methods, and files respectively. Each figure shows the results for Rhino, AspectJ, Lucene, and the overall dataset. The detailed results are given in Tables 3.1, 3.2, and 3.3.

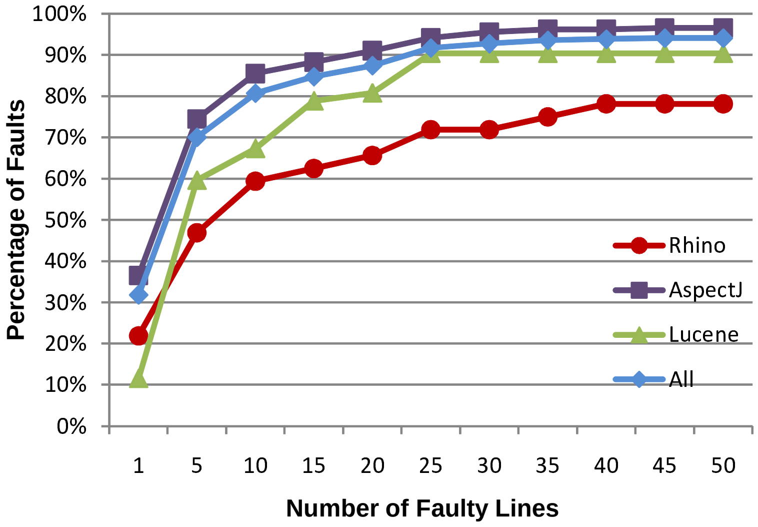


Figure 3.1: Proportion of faults versus line locality.

Considering the number of faulty lines, faults in Rhino, AspectJ, and Lucene could span up to 957, 103, and 772 lines respectively. However, there are not many versions that have faults residing in more than 50 lines – only 22%, 3%, and 10%

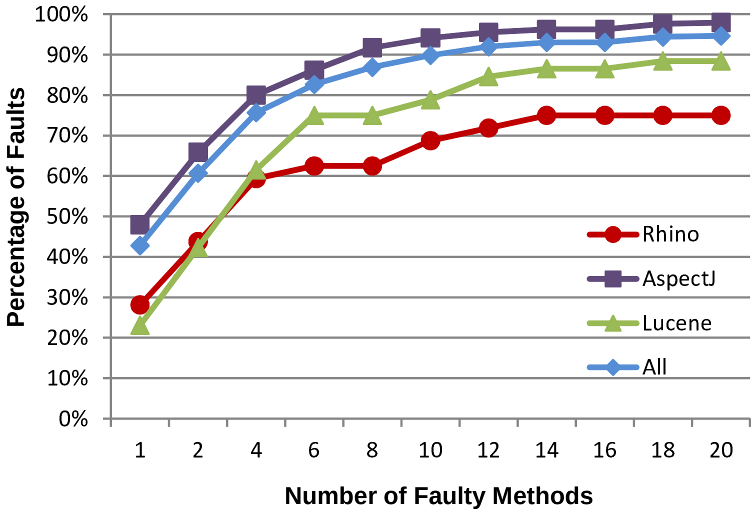


Figure 3.2: Proportion of faults versus method locality.

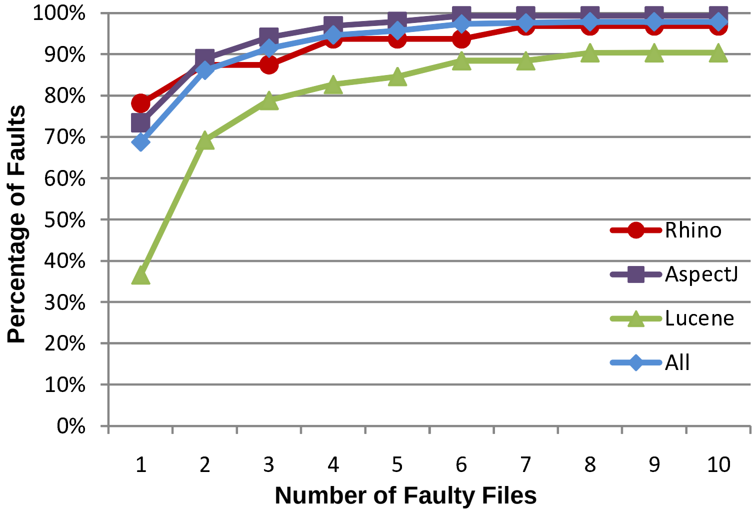


Figure 3.3: Proportion of faults versus file locality.

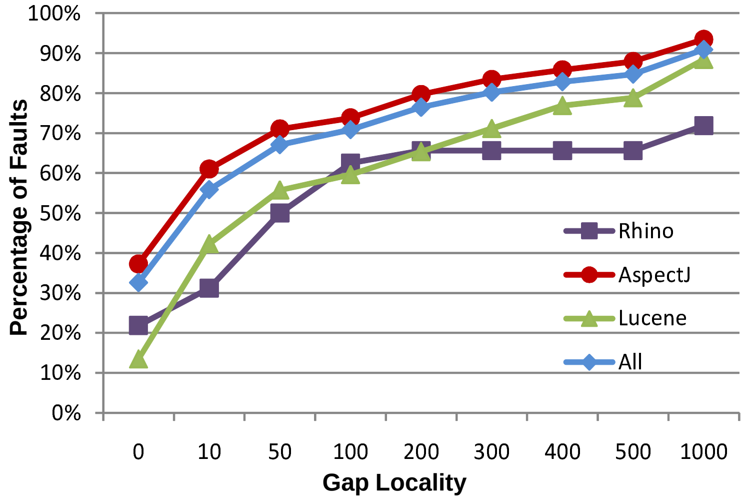


Figure 3.4: Proportion of faults versus gap locality.

of Rhino's, AspectJ's, and Lucene's faults respectively. Figure 3.1 shows the proportion of faults that span across 1 to 50 lines of code. Among the faults, 22% of Rhino's faults, 37% of AspectJ's faults, and 12% of Lucene's faults involve one line of code. Overall, 32% of the faults involve at most one line of code. More than 50% of the faults involve at most 10 lines of codes i.e., 59% of Rhino's faults, 86% of AspectJ's faults, 67% of Lucene's faults, and 81% of all faults. Therefore, most faults are localized within 10 lines of code.

Considering the number of faulty methods, faults in Rhino, AspectJ, and Lucene could span up to 296, 78, and 778 methods respectively. However, only few versions have faults residing in more than 20 methods – only 25%, 2%, 12% of Rhino's, AspectJ's, and Lucene's faults respectively. Figure 3.2 shows the proportion of faults that span across 1 to 20 methods. Among the faults, 28% of Rhino's faults, 49% of AspectJ's faults, and 23% of Lucene's faults involve one method. Overall 44% and 80% of the faults involve at most one method and six methods respectively.

Considering the number of faulty files, faults in Rhino, AspectJ, and Lucene could span up to 14, 56, and 57 files respectively. However, only few versions have faults residing in more than 10 files – only 3%, 1%, 10% of Rhino's, AspectJ's, and Lucene's faults respectively. Figure 3.3 shows the proportion of faults that span across 1 to 10 files. Among the faults, 76% of Rhino's faults, 77% of AspectJ's faults, and 37% of Lucene's faults involve one file. Overall 69% of the faults involve at most one file and 86% of the faults involve at most two files.

Figure 3.4 shows the proportion of faults that have 1 to 1,000 gap locality scores. We notice that most faults have rather big gap locality in the three programs.

3.3.2 RQ2: Are Severe Faults Localizable?

We evaluate the relationship between localizable bugs and their severity levels. There are six severity levels in the AspectJ bugs that we analyze: blocker, critical, major, minor, normal, and trivial (from high to low). As for Rhino, there are only two levels: major and normal. For Lucene, there are four levels: blocker, ma-

Locality	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**
1 Line	7 (22%)	1 Method	9 (28%)	1 File	25 (76%)
2 Lines	3 (9%)	2 Methods	5 (16%)	2 Files	3 (10%)
3 Lines	2 (6%)	3 Methods	4 (13%)	3 Files	0 (0%)
4 Lines	0 (0%)	4 Methods	1 (3%)	4 Files	2 (7%)
5 Lines	3 (9%)	5 Methods	1 (3%)	5 Files	0 (0%)
6 Lines	0 (0%)	6 Methods	0 (0%)	6 Files	0 (0%)
7 Lines	2 (6%)	7 Methods	0 (0%)	7 Files	1 (3%)
8 Lines	1 (3%)	8 Methods	0 (0%)	8 Files	0 (0%)
9 Lines	1 (3%)	9 Methods	1 (3%)	9 Files	0 (0%)
10 Lines	0 (0%)	10 Methods	1 (3%)	10 Files	0 (0%)

Table 3.1: Number & percentage of faults (in parentheses) covering different number of faulty lines, methods, and files for Rhino

Locality	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**
1 Line	106 (37%)	1 Method	139 (48%)	1 File	213 (77%)
2 Lines	49 (17%)	2 Methods	49 (17%)	2 Files	45 (16%)
3 Lines	32 (11%)	3 Methods	26 (9%)	3 Files	15 (5%)
4 Lines	13 (4%)	4 Methods	15 (5%)	4 Files	8 (3%)
5 Lines	16 (6%)	5 Methods	8 (3%)	5 Files	3 (1%)
6 Lines	8 (3%)	6 Methods	10 (3%)	6 Files	4 (1%)
7 Lines	9 (3%)	7 Methods	11 (4%)	7 Files	0 (0%)
8 Lines	4 (1%)	8 Methods	8 (2%)	8 Files	0 (0%)
9 Lines	5 (2%)	9 Methods	3 (1%)	9 Files	0 (0%)
10 Lines	6 (2%)	10 Methods	4 (1%)	10 Files	0 (0%)

Table 3.2: Number and percentage of faults (in parentheses) covering different number of lines, methods, and Java files for AspectJ

Locality	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**	**Locality**	**Num. of Faults (% of Faults)**
1 Line	6 (12%)	1 Method	12 (23%)	1 File	19 (37%)
2 Lines	10 (19%)	2 Methods	10 (19%)	2 Files	17 (33%)
3 Lines	8 (15%)	3 Methods	5 (10%)	3 Files	5 (10%)
4 Lines	5 (10%)	4 Methods	5 (10%)	4 Files	2 (4%)
5 Lines	2 (4%)	5 Methods	3 (6%)	5 Files	1 (2%)
6 Lines	3 (6%)	6 Methods	4 (8%)	6 Files	2 (4%)
7 Lines	1 (2%)	7 Methods	0 (0%)	7 Files	0 (0%)
8 Lines	0 (0%)	8 Methods	0 (0%)	8 Files	1 (2%)
9 Lines	0 (0%)	9 Methods	1 (2%)	9 Files	0 (0%)
10 Lines	0 (0%)	10 Methods	1 (2%)	10 Files	0 (0%)

Table 3.3: Number and percentage of faults (in parentheses) covering different number of lines, methods, and Java files for Lucene

Bug severity	1 Line	10 Lines	1 Method	1 File
Major	0 (0%)	1 (50%)	1 (50%)	2 (100%)
Normal	7 (23%)	18 (60%)	8 (30%)	23 (85%)

Table 3.4: Numbers and percentages of faults for different severity levels in Rhino when faults are within 1 line, 10 lines, 1 method, or 1 file

Bug severity	1 Line	10 Lines	1 Method	1 File
Blocker	3 (43%)	7 (100%)	4 (57%)	6 (86%)
Critical	8 (33%)	19 (79%)	11 (46%)	16 (67%)
Major	12 (40%)	26 (87%)	15 (50%)	22 (73%)
Minor	6 (43%)	11 (79%)	5 (36%)	11 (79%)
Normal	75 (35%)	183 (86%)	102 (48%)	156 (73%)
Trivial	2 (100%)	2 (100%)	2 (100%)	2 (100%)

Table 3.5: Numbers & percentages of faults for different severity levels in AspectJ when faults are within 1 line, 10 lines, 1 method, or 1 file

jor, minor, trivial. Table 3.4, 3.5, and 3.6 show the severity levels when faults reside in one line, less than ten lines, one method, and one file for Rhino, AspectJ, and Lucene respectively. For Rhino, 0%, 50%, 50%, and 100% of the major faults are localizable to one line of code, ten lines of code, one method, and one file, respectively. For AspectJ, 43%, 100%, 57%, and 86% of the blocker faults are localizable to one line of code, ten lines of code, one method, and one file respectively. For Lucene, 0%, 67%, 33%, and 33% of the blocker faults are localizable to one line of code, ten lines of code, one method, and one file respectively. Overall, considering the most severe bugs in the category (i.e., blocker for Lucene and AspectJ, and major for Rhino), 25%, 83%, 50%, and 75% of these severe bugs are localizable to one line of code, ten lines of code, one method, and one file respectively. The results show that many severe faults are localizable. A significant percentage of severe faults are however not localizable.

Bug severity	1 Line	10 Lines	1 Method	1 File
Blocker	0 (0%)	2 (67%)	1 (33%)	1 (33%)
Major	4 (16%)	17 (68%)	7 (28%)	9 (36%)
Minor	2 (10%)	14 (67%)	4 (19%)	9 (43%)
Trivial	0 (0%)	2 (100%)	0 (0%)	0 (0%)

Table 3.6: Numbers and percentages of faults for different severity levels in Lucene when faults are within 1 line, 10 lines, 1 method, or 1 file

3.3.3 Threats to Validity

Threat to external validity refers to the generalizability of our findings. In this study, we only analyze three real Java programs: AspectJ, Rhino, and Lucene. The findings may not be generalizable to programs written in different languages. Threat to internal validity is mostly related to the manual investigation of what lines of code are responsible for each bug. There may be errors in the labeling process.

3.4 Conclusion

In this chapter, we have performed a preliminary study for the question whether bugs are localizable. We have analyzed hundreds of bugs and their fixes from three software systems, AspectJ, Rhino, and Lucene, and manually extract faults from changes. We have found that (1) a substantial proportion of faults (i.e., 32%) still manifest in a single line of code, and 56% of the faults are localizable within a few lines of code or a few methods and (2) only 25% and 50% severe faults (i.e., categorized as blocker in AspectJ and Lucene, and major in Rhino) are localizable within one line and one method respectively.

Chapter 4

Employing Association Measures for Fault Localization

In this chapter, we present our work that automatically locates faulty program elements to help developers locate bugs in programs. For a given buggy program, our approach quantifies the likelihood that a program element is the faulty program element, then presents a list of most suspicious program elements to developers for inspection. The idea of our approach is to model the fault localization problem as the association between the executions of a program element with program failures. We employ association measures that have been introduced in the statistics and data mining domains to measure the likelihood that a program element is the faulty program element.

We organize this chapter as follows. Section 4.1 presents the motivation and main contribution of this chapter. Section 4.2 explains the concept and definition used in this work. Section 4.3 presents our approach that employs association measures for localizing faults. We evaluate our approach to localize faults in Section 4.4 and discuss our results in Section 4.5. Section 4.7 concludes our findings.

4.1 Association Measures for Localizing Faults

Program traces or abstractions of traces (called *program spectra*) can be used to represent program behaviors. The spectra of correct and failed executions can be compared to identify potentially faulty program elements (e.g., statements, basic blocks, functions, and components). Spectrum-based fault localization is one family of the fault localization techniques that utilize program spectra [3, 27, 31, 65, 84]. They often employ statistical analysis to compare the program traces. Program elements that are observed more often in failed executions than in correct executions (or statistically correlate with failures) may be identified and presented to developers for further inspection.

Spectrum-based fault localization techniques are promising as they are lightweight and have good accuracy. Among the existing spectrum-based fault localization techniques, two well-known techniques are Tarantula [65, 66] and Ochiai [3, 5, 6]. Both approaches compute a suspiciousness score for each program element based on the execution frequencies of the element in correct and failed executions. Then, all program elements are ranked according to their scores and presented to developers for inspection. Presenting the buggy element earlier in the ranking helps developers to effectively and efficiently localize bugs.

Unfortunately, the effectiveness of current state-of-the-art fault localization techniques is not yet perfect. Faulty program elements are often ranked low in the returned ranked list. Thus, there is a need to improve the state-of-the-art so that fault localization techniques can provide better assistance for developers while minimizing the cost of performing bug fixing tasks.

Fault localization boils down to the problem of computing good suspiciousness scores to differentiate faulty and correct program elements. The computation of the suspiciousness scores in effect answers the following question:

What is the strength of association between the executions of a particular program element with failures?

Various association measures in the data mining and statistics communities, such as odds ratio [9], Yule's Q [148], and Yule's Y [149] have been proposed to measure the strength of association between two variables of interest. For example, one might be interested in the association between an application of a particular medical treatment with recovery from an illness, or in the association between an execution of a business strategy with a revenue change. There are several studies that evaluate the effectiveness of some similarity coefficients, e.g., Jaccard [3, 4], Sorensen-Dice [4], Anderberg [4], Simple Matching [4], Rogers and Tanimoto [4], and Ochiai II [4].

Thus, a general solution for fault localization can naturally emerge based on the proliferation of association measures in the literature: measure the strength of association between a program element's executions and failures, and the stronger the association is, the more likely the program element is a fault.

Besides Tarantula and Ochiai which have been used for fault localization, there are rich varieties of association measures that have not been studied for fault localization. This chapter aims to fill this gap by investigating the effectiveness of 40 well-known association measures as methods to rank program elements for the purpose of fault localization and compare them with Tarantula and Ochiai. In particular, we are interested in answering the following research questions (RQs):

RQ 1. Are vanilla or off-the-shelf association measures accurate enough in localizing faults?

RQ 2. Among 40 association measures, which of the association measures are the most accurate for localizing faults?

RQ 3. What is the relative performance of off-the-shelf association measures as compared to well-known suspiciousness measures (in particular, Tarantula and Ochiai) for fault localization?

RQ 4. Is the accuracy of off-the-shelf association measures, of Tarantula, and of Ochiai, in localizing faults for programs written in C, different from their accuracy in localizing faults for programs written in Java?

RQ 5. What is the effectiveness of off-the-shelf association measures, Tarantula, and Ochiai in localizing different types of bugs?

RQ 6. What is the accuracy of off-the-shelf association measures, Tarantula, and Ochiai in localizing bugs in multiple-bug programs?

We use association measures as measurements to rank program elements in a program for the purpose of locating faults. We investigate and compare the accuracies of Tarantula, Ochiai, and the additional 40 association measures on programs from the Siemens test suite [125], and three larger programs from Software Infrastructure Repository (SIR) [35]. The latter includes Space which is written in C, and NanoXml and XmlSecurity, which are written in Java. The programs come along with seeded bugs, test cases, and test oracles to decide between failures and non-failures. We show that a few association measures could better localize faults than Ochiai and that many are better than Tarantula.

The contributions of this work are as follows:

1. We comprehensively investigate the effectiveness of 40 association measures for fault localization.

2. We highlight a few promising association measures that can outperform two well-known spectrum-based fault localization approaches (i.e., Tarantula and Ochiai) and those that are comparable with these two approaches.

3. We provide a partial order of association measures in terms of their accuracy for fault localization.

4. We characterize the effectiveness of the association measures, Ochiai, and Tarantula on programs written in different programming languages.

5. We analyze different kinds of bugs and investigate the effectiveness of the 40 association measures, Ochiai, and Tarantula in localizing each of these categories of bugs.

6. We investigate the accuracy of the 40 association measures, Ochiai, and Tarantula in localizing faulty programs containing multiple bugs.

4.2 Concepts & Definitions

In this section we formally introduce the problem of spectrum-based fault localization as the computation of association strengths between the executions of various program elements and failures. Also, we describe the concept of dichotomy matrix that is used in the calculation of these association strengths.

4.2.1 Spectrum-Based Fault Localization

This problem starts with a faulty program, a set of test cases, and a test oracle. The test cases are run over the faulty program and observations on how the program runs on each of the test cases are recorded as program spectra. A program spectrum represents certain characteristics of an execution of a program, providing a behavior signature of the execution [117]. The signature of a behavior could be a set of counters, each of which indicates the number of times each program element (e.g., statement, basic block, path, etc.) is executed in one execution [52]. The counters could also simply be 0-1 flags that indicate whether an element is executed. A test oracle is available to label whether a particular output or execution of a test case is correct or wrong. Wrong executions are classified as program failures. The task of a fault localization tool is to find the program elements that are responsible for the failures (i.e., the faults or the root causes) based on the program spectra of both correct and wrong executions.

Various spectra have been proposed in the literature [3, 52]. Different spectra may have different effects on the effectiveness of fault localization. The block-hit spectra are at a suitable profiling granularity because all code in the same basic block has the same execution pattern and there is no need to instrument individual instructions in a granularity finer than blocks. Also, it has been shown that the instrumentation costs to obtain such spectra are relatively low and it can be used for effective fault localization [3–6, 52]. The granularity has a balance between reducing instrumentation costs and having sufficient bug-revealing capabilities.

In this work, we use *block-hit program spectra*, each of which consists of a set of flags to indicate whether each basic block is executed or not in each test case. An example of block-hit program spectra is shown in Figure 4.1. The first column contains identifiers of basic blocks. The second column contains the statements in the corresponding basic blocks. The other columns indicate whether each basic block is executed in test cases T15, T16, T17, and T18 along with the information whether each of the test cases passes or fails. In this example, • denotes that a basic block is executed by a test case and an empty cell denotes that the block is not executed by the test case. In the code snippet, a bug lies in the condition of the `if` statement in Block 3, causing Blocks 4–5 to be skipped when the variable `count` is 1. Note that in test cases T16-T18, execution of Block 2 that contains return statement is followed by the execution of Block 3. Normally, Block 3 should not be executed, but since the function containing this code snippet is being called inside a loop, thus this block is executed.

Based on these spectra, we want to compute the suspiciousness score of each program element following Definition 4.2.1.

Definition 4.2.1 (Suspiciousness Score) *Consider a program $P = \{e_1, ..., e_n\}$ and a set of program spectra $T = T_s \cup T_f$ for P, where P comprises of n elements*

Block ID	Program Elements	T15	T16	T17	T18

1	int count;
	int n;
	Ele *proc;
	List *src_queue, *dest_queue;
	if (prio >= MAXPRIO) /*maxprio=3*/
		●	●	●	●

2	{return;}		●	●	●

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 1) /* Bug*//* expected : count>0*/ {
		●	●	●	●

4	n = (int) (count*ratio + 1);
	proc = find_nth(src_queue, n);
	if (proc) {
			●	●	

5	src_queue = del_ele(src_queue, proc);
	proc->priority = prio;
	dest_queue = append_ele(dest_queue, proc); } }}
			●	●	

	Status of Test Case Execution :	Pass	Pass	Pass	Fail

Figure 4.1: Example of block-hit program spectra

Symbol	**Definition**
n	Total number of test cases in the test suite
$n(e)$	Number of test cases that execute a program element e
n_s	Total number of correct test cases
n_f	Total number of failed test cases
$n_s(e)$	Number of correct test cases that execute a program element e
$n_f(e)$	Number of failed test cases that execute a program element e

Table 4.1: Some common notations

$e_1, ..., e_n$ *and* T *comprises the spectra for correct executions* T_s *and the spectra for wrong executions* T_f*. We would like to measure the strength of the association between the* executions of each e_i *and program failures and assign this strength as the* suspiciousness score *of* e_i *denoted as* $suspiciousness(e_i)$.

Tarantula

Jones and Harrold propose Tarantula [65] to rank program elements based on their suspiciousness scores. Intuitively, a program element is more suspicious if it appears in failed executions more frequently than in correct executions. Considering a program P and a test suite T, Table 4.1 introduces some common notations that are used in the rest of this work.

Tarantula computes suspiciousness score for a program element e as follows:

$$suspiciousness(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_s(e)}{n_s} + \frac{n_f(e)}{n_f}}$$

Based on block-hit program spectra shown in Figure 4.1, the suspiciousness score of Block 3 that contains the bug is $\frac{1/1}{3/3+1/1} = 0.5$. Block 1 has the same suspiciousness score. Interestingly, Block 2 receives the highest suspiciousness score: $\frac{1/1}{2/3+1/1} = 0.6$. Following the same calculation, Blocks 4 and 5 are not suspicious because there is no failure that executes these blocks. Assuming developers inspect program elements one by one from the most suspicious to the least suspicious, the bug in Block 3 can be found after inspecting at most 3 blocks.

Ochiai

Abreu et al. [6] propose Ochiai to assign suspicious scores to a program element. Given a program element, Ochiai analyzes the number of failed test cases that execute the program element (i.e., $n_f(e)$), the total number of failed test cases (i.e., n_f), and the number of test cases that execute the program element (i.e., $n(e)$). Similar to Tarantula, Ochiai considers an element more suspicious if it occurs more frequently in failed executions than in correct executions (computed as $\sqrt{\frac{n_f(e)}{n(e)}}$). Ochiai calculates the suspiciousness score of a program element as follows:

$$suspiciousness(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_s(e))}} = \frac{n_f(e)}{\sqrt{n_f n(e)}}$$

Using the same example shown in Figure 4.1, Blocks 1 and 3 receive a suspiciousness score of: $1/\sqrt{1 * (1 + 3)} = 0.50$. Similar to Tarantula, Ochiai also returns Block 2 as the most suspicious block: $1/\sqrt{1 * (1 + 2)} = 0.58$, while the remaining blocks are assigned suspiciousness scores of 0. Both Tarantula and Ochiai can find the bug after inspecting 3 blocks. In this study, we are interested in investigating other association measures that can possibly localize the bug earlier.

4.2.2 Dichotomous Association

A common characteristic of the association measures evaluated in this work is that they are all defined based on dichotomy matrices. The following are the necessary definitions.

Definition 4.2.2 (Dichotomy) *A* dichotomous outcome *is an outcome whose values could be split into two categories, e.g., wrong or correct, executed or skipped, married or unmarried, etc. A* dichotomous variable *is a variable having a dichotomous outcome. A* dichotomy matrix *is a 2×2 matrix that tries to associate two dichotomous variables in the form of a 2×2 contingency table which records the bivariate frequency distribution of the two variables.*

An example of a dichotomy matrix $D(A, B)$ relating variables A and B is shown in Table 4.2. The value c_{00} corresponds to the number of observations in which the value of variable A equals to A_0 and the value of variable B equals to B_0. The values of the other three entries in the dichotomy matrix are similarly defined.

	$A = A_0$	$A = A_1$
$B = B_0$	c_{00}	c_{01}
$B = B_1$	c_{10}	c_{11}

Table 4.2: An example of a dichotomy matrix. We refer to it as $D(A, B)$.

Based on the concept of dichotomy matrix, we introduce dichotomous association in Definition 4.2.3.

Definition 4.2.3 (Dichotomous Association) *A* dichotomous association *is a special form of bivariate association [54] which measures the strength of association between two dichotomous variables, e.g., application of a medical treatment and recovery from the disease, job satisfaction and productivity, and program element execution and program failure. The formulae for calculating dichotomous associations depend on the four entries in dichotomy matrices.*

Given a dichotomy matrix relating two variables, two questions are often asked:

1. Is there a (dichotomous) association between the two variables?

2. How strong is the association between the two variables?

A common way to answer these two questions is to define a formula, referred as an *association measure*, to calculate a score based on the four entries in a dichotomy matrix and consider that the association exists (or is strong) if the score is beyond a particular threshold. We define association measure in Definition 4.2.4.

Definition 4.2.4 (Association Measure) *An association measure M of two variables A and B is a mathematical function of the four entries of a dichotomy matrix $D(A, B)$, and is denoted as $M(A, B, D(A, B))$ or simply $M(A, B, D)$ if it is clear from the context.*

In fault localization, we could produce a dichotomy matrix that relates the executions of a program element with program failures. Consider a program element e. For each test case, a trace is generated when a subject program is executed on the test case. Some traces execute e, others do not. Some traces correspond to failures, others correspond to correct executions. After the test cases are run, a dichotomy matrix as shown in Table 4.3 is produced for every program element e.

	e Executed	e Not Executed
Test Passed	$n_s(e)$	$n_s(\overline{e})$
Test Failed	$n_f(e)$	$n_f(\overline{e})$

Table 4.3: Dichotomy matrix for fault localization.

The notation $\overline{e}$ means e is not executed, while other notations are defined in Table 4.1. Thus, $n_s(\overline{e})$ is the number of test cases that do not execute e and pass, while $n_f(\overline{e})$ is the number of test cases that do not execute e and fail.

Considering variables E and F to represent the execution of a program element and the occurrence of a program failure, respectively, we are interested in computing $M(E, F, D_e(E, F))$ (i.e., the association between the execution of e and a failure), where $D_e(E, F)$ represents the dichotomy matrix of the two variables E and F for a program element e. The formulae of the 40 association measures are given in Section 4.3.

4.3 Association Measures

In this section, we first describe how a dichotomy matrix can be constructed. Next, we present how the 40 association measures can be computed from a dichotomy matrix. Finally, we give an example that sorts program elements for inspection using the association measures.

4.3.1 Constructing a Dichotomy Matrix

Constructing a dichotomy matrix requires program instrumentation that supports collection of program execution traces and a test oracle. In this work, we instrument all the buggy programs in our dataset at the basic block level. We manually instrument C programs and use Cobertura[1] to instrument Java programs. In order to construct a dichotomy matrix, we collect program execution traces that contain information about basic blocks that are executed by each test case. The test oracle then gives us information on whether a test case fails or passes.

Next, for each basic block, we construct a dichotomy matrix by counting the number of times the basic block is executed by the passing test cases (denoted as $n_s(e)$), the number of times the basic block is not executed by the passing test cases (denoted as $n_s(\overline{e})$), the number of times the basic block is executed by failing test cases which is denoted as $n_f(e)$, and the number of times the basic block is not executed by failing test cases which is denoted as $n_f(\overline{e})$. The information in this dichotomy matrix is then used as an input to the association measures to calculate how likely it is that the corresponding basic block contains a bug.

[1] http://cobertura.sourceforge.net/

4.3.2 Association Measures

There have been a number of studies proposed in the statistics and data mining community on measures of association between variables since the early 20th century. These include measures such as Yule's Q and Yule's Y [148, 149]. Other measures, such as the Odds Ratio [9], are also commonly considered and utilized in various domains, such as medicine [10] and social science [54]. In the data mining community, Agrawal and Srikant have proposed association rule mining to infer associations from two itemsets in a transaction dataset in the early 90s [8]. They propose the metrics of support and confidence for measuring the strength of an association. Various other metrics, such as interest and collective strength, were proposed later.

Various association measures have been studied by comparing their properties, and investigating the benefits and limitations of each from a computational point of view [132]. The measures are revisited by Geng and Hamilton by including measures for aggregated data summaries [44]. Some of the association measures have been studied for fault localizations, e.g. Jaccard [3, 4], Sorensen-Dice [4], Anderberg [4], Simple Matching [4], Rogers and Tanimoto [4], and Ochiai II [4]. In this work, we revisit the effectiveness of the measures that have been studied before in localizing faults, because their effectiveness on localizing various kinds of bugs and localizing bugs in multiple-bug programs have not been evaluated.

We investigate the effectiveness of 40 association measures in localizing various kinds of bugs that reside in single-bug and multiple-bug programs. The 40 association measures that we consider are as follows: ϕ-coefficient [54], odds ratio [9], Yule's Q [148], Yule's Y [149], Kappa [32], J-Measure [127], gini index [45], support [8], confidence [8], Clark and Boswell's Laplace accuracy [29], conviction [24], interest [24], cosine [132], Piatetsky-Shapiro's Leverage [108], certainty factor [124], added value [132], collective strength [7], Jaccard [51], Klosgen [75], information gain [26, 112], Coverage [44, 103], Accuracy [44, 103], Leverage [44, 103], Relative Risk [44, 103], Interestingness Weighting Depen-

$P(A) = \frac{n_f(e)+n_s(e)}{n}$	$P(A, B) = \frac{n_f(e)}{n}$	$P(B\|A) = \frac{P(A,B)}{P(A)}$
$P(\overline{A}) = \frac{n_f(\overline{e})+n_s(\overline{e})}{n}$	$P(\overline{A}, B) = \frac{n_f(\overline{e})}{n}$	$P(A\|B) = \frac{P(A,B)}{P(B)}$
$P(B) = \frac{n_f(e)+n_f(\overline{e})}{n}$	$P(A, \overline{B}) = \frac{n_s(e)}{n}$	
$P(\overline{B}) = \frac{n_s(e)+n_s(\overline{e})}{n}$	$P(\overline{A}, \overline{B}) = \frac{n_s(\overline{e})}{n}$	

Table 4.4: Preliminary definitions and equations.

dency [44, 103], Goodman and Kruskal [44, 46, 103, 132], Normalized Mutual Information [44, 103, 132], One-Way Support [44, 103], Two-Way Support [44, 103], Two-Way Support Variation [44, 103], Loevinger [44], Sebag-Schoenauer [44], Least Contradiction [44], Odd Multiplier [44], Example and Counterexample Rate [44], Zhang [44], Sorensen-Dice [34, 129], Anderberg [11], Simple Matching [128], Rogers and Tanimoto [118], and Ochiai II [102].

The mathematical formulae of the association measures are defined in terms of probabilities, instead of frequencies, but we can derive the probabilities from the frequencies recorded in dichotomous matrices. The mathematical formulae for calculating the association measures, including Tarantula and Ochiai, are given in Tables 4.5 and 4.6. The ranges of values that these association measures can take are given in Tables 4.7 and 4.8.

4.3.3 From Association to Suspiciousness

From Section 4.2, the suspiciousness score for a program element e with dichotomy matrix D_e is defined as the strength of the association between the executions of e with failures (i.e., $M(E, F, D_e)$). M refers to one of the 40 association measures presented in Section 4.3.2.

Next, we illustrate how an association measure could be used to rank program elements for inspection. Using the example in Figure 4.1, we observe that there are 5 blocks: Blocks 1, 2, 3, 4, and 5. The bug resides at the `if` statement in Block 3. The suspiciousness score of Block 3 is determined by the association

Name	**Formula**
ϕ-Coefficient (M_1)	$\frac{P(A,B)-P(A)P(B)}{\sqrt{P(A)P(B)(1-P(A))(1-P(B))}}$
Odds ratio (M_2)	$\frac{P(A,B)P(\overline{A},\overline{B})}{P(A,\overline{B})P(\overline{A},B)}$
Yule's Q (M_3)	$\frac{P(A,B)P(\overline{A,B})-P(A,\overline{B})P(\overline{A},B)}{P(A,B)P(\overline{A,B})+P(A,\overline{B})P(\overline{A},B)} = \frac{\alpha-1}{\alpha+1}$
Yule's Y (M_4)	$\frac{\sqrt{P(A,B)P(\overline{A,B})}-\sqrt{P(A,\overline{B})P(\overline{A},B)}}{\sqrt{P(A,B)P(\overline{A,B})}+\sqrt{P(A,\overline{B})P(\overline{A},B)}} = \frac{\sqrt{\alpha}-1}{\sqrt{\alpha}+1}$
Kappa (M_5)	$\frac{P(A,B)+P(\overline{A},\overline{B})-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A)P(B)-P(\overline{A})P(\overline{B})}$
J-Measure (M_6)	$\max(P(A,B)\log(\frac{P(B\|A)}{P(B)}) + P(A\overline{B})\log(\frac{P(\overline{B}\|A)}{P(\overline{B})}), P(A,B)\log(\frac{P(A\|B)}{P(A)}) + P(\overline{A}B)\log(\frac{P(\overline{A}\|B)}{P(\overline{A})}))$
Gini Index (M_7)	$\max(P(A)[P(B\|A)^2 + P(\overline{B}\|A)^2] + P(\overline{A})[P(B\|\overline{A})^2 + P(\overline{B}\|\overline{A})^2] - P(B)^2 - P(\overline{B})^2, P(B)[P(A\|B)^2 + P(\overline{A}\|B)^2] + P(\overline{B})[P(A\|\overline{B})^2 + P(\overline{A}\|\overline{B})^2] - P(A)^2 - P(\overline{A})^2)$
Support (M_8)	$P(A,B)$
Confidence (M_9)	$\max(P(B\|A), P(A\|B))$
Laplace (M_{10})	$\max(\frac{P(A,B)+1}{P(A)+2}, \frac{P(A,B)+1}{P(B)+2})$
Conviction (M_{11})	$\max(\frac{P(A)P(\overline{B})}{P(A\overline{B})}, \frac{P(B)P(\overline{A})}{P(B\overline{A})})$
Interest (M_{12})	$\frac{P(A,B)}{P(A)P(B)}$
Piatetsky-Shapiro's (M_{13})	$P(A,B) - P(A)P(B)$
Certainty Factor (M_{14})	$\max(\frac{P(B\|A)-P(B)}{1-P(B)}, \frac{P(A\|B)-P(A)}{1-P(A)})$
Added Value (M_{15})	$\max(P(B\|A) - P(B), P(A\|B) - P(A))$
Collective Strength (M_{16})	$\frac{P(A,B)+P(\overline{A,B})}{P(A)P(B)+P(\overline{A})P(\overline{B})} \times \frac{1-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A,B)-P(\overline{AB})}$
Jaccard (M_{17})	$\frac{P(A,B)}{P(A)+P(B)-P(A,B)}$
Klosgen (M_{18})	$\sqrt{P(A,B)}\max(P(B\|A) - P(B), P(A\|B) - P(A))$
Information Gain (M_{19})	$(-P(B)\log P(B) - P(\overline{B})\log P(\overline{B})) - (P(A) \times (-P(B\|A)\log P(B\|A)) - P(\overline{B}\|A)\log P(\overline{B}\|A) - P(\overline{A}) \times (-P(B\|\overline{A})\log P(B\|\overline{A})) - P(\overline{B}\|\overline{A})\log P(\overline{B}\|\overline{A})))$

Table 4.5: Definitions of association measures, Tarantula, and Ochiai [Part I].

Name	**Formula**
Coverage (M_{20})	$P(A)$
Accuracy (M_{21})	$P(A,B) + P(\overline{A},\overline{B})$
Leverage (M_{22})	$P(B\|A) - P(A)P(B)$
Relative Risk (M_{23})	$P(B\|A)/P(B\|\overline{A})$
Interestingness Weighting Dependency (M_{24})	$((\frac{P(A,B)}{P(A)P(B)})^k - 1)P(A,B)^m$ where k,m are coefficients of dependency and generality, respectively, weighting the relative importance of the two factors.
Goodman and Kruskal (M_{25})	$\frac{\sum_i max_j P(A_i,B_j)+\sum_j max_i P(A_i,B_j)-max_i P(A_i)-max_j P(B_j}{2-max_i P(A_i)-max_j P(B_j))}$
Normalized Mutual Information (M_{26})	$\sum_i \sum_j P(A_i,B_j)log_2 \frac{P(A_i,B_j)}{P(A_i)P(B_j)}/\{-\sum_i P(A_i)log_2 P(A_i)\}$
One-Way Support (M_{27})	$P(B\|A)log_2 \frac{P(A,B)}{(P(A)P(B))}$
Two-Way Support (M_{28})	$P(A,B)log_2 \frac{P(A,B)}{(P(A)P(B))}$
Two-Way Support Variation (M_{29})	$P(A,B)log_2 \frac{P(A,B)}{(P(A)P(B))} + P(A,\overline{B})log_2 \frac{P(A,\overline{B})}{(P(A)P(\overline{B}))} + P(\overline{A},B)log_2 \frac{P(\overline{A},B)}{(P(\overline{A})P(B))} + P(\overline{A},\overline{B})log_2 \frac{P(\overline{A},\overline{B})}{(P(\overline{A})P(\overline{B}))}$
Loevinger (M_{30})	$1 - \frac{P(A)P(\overline{B})}{P(A,\overline{B})}$
Sebag-Schoenauer (M_{31})	$\frac{P(A,B)}{P(A,\overline{B})}$
Least Contradiction (M_{32})	$\frac{P(A,B)-P(A,\overline{B})}{P(B)}$
Odd Multiplier (M_{33})	$\frac{P(A,B)P(\overline{B})}{P(B)P(A,\overline{B})}$
Example and Counterexample Rate (M_{34})	$1 - \frac{P(A,\overline{B})}{P(A,B)}$
Zhang (M_{35})	$\frac{P(A,B)-P(A)P(B)}{max(P(A,B)P(\overline{B}),P(B)P(A,\overline{B})}$
Sorensen-Dice (M_{36})	$\frac{2P(A,B)}{2P(A,B)+P(\overline{A},B)+P(A,\overline{B})}$
Anderberg (M_{37})	$\frac{P(A,B)}{P(A,B)+2(P(\overline{A},B)+P(A,\overline{B}))}$
Simple-Matching (M_{38})	$P(A,B) + P(\overline{A},\overline{B})$
Rogers and Tanimoto (M_{39})	$\frac{P(A,B)+P(\overline{A},\overline{B})}{P(A,B)+P(\overline{A},\overline{B})+2(P(\overline{A},B)+P(A,\overline{B}))}$
Ochiai II (M_{40})	$\frac{P(A,B)+P(\overline{A},\overline{B})}{\sqrt{(P(A,B)+P(\overline{A},B))(P(A,B)+P(A,\overline{B}))(P(\overline{A},\overline{B})+P(\overline{A},B))(P(\overline{A},\overline{B})+P(A,\overline{B}))}}$
Tarantula	$\frac{\frac{P(A,B)}{P(B)}}{\frac{P(A,\overline{B})}{P(\overline{B})}+\frac{P(A,B)}{P(B)}}$
Ochiai	$\frac{P(A,B)}{\sqrt{P(A)P(B)}}$

Table 4.6: Definitions of association measures, Tarantula, and Ochiai [Part II].

Name	**Range**	**No**	**Perfect**
ϕ-Coefficient (M_1)	$-1 \ldots 0 \ldots 1$	0	1
Odds ratio (M_2)	$0 \ldots 1 \ldots \infty$	1	∞
Yule's Q (M_3)	$-1 \ldots 0 \ldots 1$	0	1
Yule's Y (M_4)	$-1 \ldots 0 \ldots 1$	0	1
Kappa (M_5)	$-1 \ldots 0 \ldots 1$	0	1
J_Measure (M_6)	$0 \ldots 1$	0	1
Gini index (M_7)	$0 \ldots 1$	0	1
Support (M_8)	$0 \ldots 1$	0	1
Confidence (M_9)	$0 \ldots 1$	0	1
Laplace (M_{10})	$0 \ldots 1$	0	1
Conviction (M_{11})	$0.5 \ldots 1 \ldots \infty$	1	∞
Interest (M_{12})	$0 \ldots 1 \ldots \infty$	1	∞
Piatetsky-Shapiro's (M_{13})	$-0.25 \ldots 0 \ldots 0.25$	0	0.25
Certainty factor (M_{14})	$-1 \ldots 0 \ldots 1$	0	1
Added Value (M_{15})	$-0.5 \ldots 0 \ldots 1$	0	1
Collective strength (M_{16})	$0 \ldots 1 \ldots \infty$	1	∞
Jaccard (M_{17})	$0 \ldots 1$	0	1
Klosgen (M_{18})	$(\frac{2}{\sqrt{3}} - 1)^{1/2}[2 - \sqrt{3} - \frac{1}{\sqrt{3}}] \ldots 0 \ldots \frac{2}{3\sqrt{3}}$	0	$\frac{2}{3\sqrt{3}}$
Information Gain (M_{19})	$0 \ldots 1$	0	1

Table 4.7: Value ranges of the association measures, Tarantula, and Ochiai [Part I]. The third and fourth columns give the values corresponding to no association and perfect association respectively. Values of measures with ranges in the format of $a \ldots b$ (e.g., J-Measure $(0 \ldots 1)$, etc.) indicate positive associations with failures. Values of measures with ranges in the format of $a \ldots b \ldots c$ (e.g., ϕ-coefficient (-1..0..1), etc.) indicate positive associations with failures (if they are between b and c), or negative associations with failures (if they are between a and b). Values closer to a imply stronger associations with passing executions.

Name	**Range**	**No**	**Perfect**
Coverage (M_{20})	$0 \dots 1$	0	1
Accuracy (M_{21})	$0 \dots 1$	0	1
Leverage (M_{22})	$-1 \dots 0 \dots 1$	0	1
Relative Risk (M_{23})	$0 \dots 1 \dots \infty$	1	∞
Interestingness Weighting Dependency (M_{24})	$-1 \dots 0 \dots 1$	0	1
Goodman and Kruskal (M_{25})	$0 \dots 1$	0	1
Normalized Mutual Information (M_{26})	$0 \dots 1$	0	1
One-Way Support (M_{27})	$-1 \dots 0 \dots 1$	0	1
Two-Way Support (M_{28})	$-1 \dots 0 \dots 1$	0	1
Two-Way Support Variation (M_{29})	$0 \dots 1$	0	1
Loevinger (M_{30})	$-1 \dots 0 \dots 1$	0	1
Sebag-Schoenauer (M_{31})	$0 \dots 1 \dots \infty$	1	∞
Least Contradiction (M_{32})	$0 \dots 1$	0	1
Odd Multiplier (M_{33})	$0 \dots 1 \dots \infty$	1	∞
Example and Counterexample Rate (M_{34})	-$\infty \dots 0 \dots 1$	0	1
Zhang (M_{35})	$-1 \dots 0 \dots 1$	0	1
Sorensen-Dice (M_{36})	$0 \dots 1$	0	1
Anderberg (M_{37})	$0 \dots 1$	0	1
Simple-Matching (M_{38})	$0 \dots 1$	0	1
Rogers and Tanimoto (M_{39})	$0 \dots 1$	0	1
Ochiai II (M_{40})	$0 \dots 1$	0	1
Tarantula	$0 \dots 1$	0	1
Ochiai	$0 \dots \sqrt{P(A,B)} \dots 1$	$\sqrt{P(A,B)}$	1

Table 4.8: Value ranges of the association measures, Tarantula, and Ochiai [Part II].

strength between the executions of Block 3 and failures. For example, by using one of the association measures, e.g., Coverage, Blocks 3 and 1 receive the highest suspiciousness score, i.e., 1, followed by Block 2 whose suspiciousness score is 0.75. The suspiciousness scores of Blocks 4 and 5 are 0.5. This particular measure can rank the block containing the bug such that no other blocks receive a higher score. However, Tarantula and Ochiai are unable to do so—see Section 4.2 for details.

4.4 Empirical Evaluation

In this section we describe our datasets, our evaluation metrics, and evaluation results.

4.4.1 Datasets

Based on the availability of the programs listed in Table 2.1, which have been used in previous studies, we choose our subject programs as follows. We analyze different programs from Siemens Test Suite [58]. Siemens programs are injected with realistic bugs and are often analyzed for fault localization studies [3, 5, 6, 18, 21, 31, 48, 61, 65, 91, 92, 94, 116, 122]. We also analyze other three real programs from Software-artifact Infrastructure Repository (SIR) [125] namely: Space [4, 5, 65, 66], NanoXML [122], and XML-Security [122]. Space is written in C, while NanoXML and XML-Security are written in Java. The average number of lines of code for the various versions of Space, NanoXML and XML-Security are 6,218, 4,223, and 21,275 respectively.

The Siemens test suite was originally used for research in test coverage adequacy and was developed by Siemens Corporation Research. We use the variant provided at www.cc.gatech.edu/aristotle/Tools/subjects/. Each program contains many different versions where each version has one bug. These bugs comprise a wide array of realistic bugs. The Siemens Test Suite comes with 7 programs:

print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. The total number of buggy versions is 132, as shown in Table 4.9. We manually instrumented the buggy versions at the basic block level. Since our instrumentation cannot reach the bugs that reside in variable declarations, we exclude versions that contain this type of bugs, i.e., versions 6, 10, 19, 21 of tot_info dataset, version 12 of replace dataset, and versions 13, 14, 15, 36, 38 of tcas dataset. We exclude versions 4 and 6 of print_token because they are identical to the original version. We also exclude version 9 of schedule2 because there is no test case for it that results in a failure. Thus, in total, we use 119 buggy versions from the Siemens test suite.

Space is an interpreter for Array Definition Language (ADL) used by European Space Agency. We analyze all 35 faulty versions of Space downloaded from SIR. NanoXML is a utility for parsing XML. SIR contains 5 versions of NanoXML. We exclude NanoXML_v4 because there is no test case for this program that results in a failure. For each version, SIR provides a few bugs. In total there are 32 buggy versions for NanoXML_v1, NanoXML_v2, NanoXML_v3, and NanoXML_v5, and we analyze 30 of them. We exclude two buggy versions because there is no test case that results in a failure. XML-Security is a Java library that supports digital signature and encryption. SIR contains 3 versions of XML-Security. Again for each version, SIR provides a few bugs. In total there are 32 buggy versions for XMLSec_v1, XMLSec_v2, and XMLSec_v3. We exclude 16 buggy versions because there is no test case that results in a failure. Thus, the total number of buggy versions that we analyze for the 3 programs are 81.

Table 4.9 provides the number of lines, the programming language in which the program is written, the number of faulty versions, and the number of test cases for each subject program.

Dataset	LOC	Language	Num. of Faulty Version	Num. of Test Cases
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6,218	C	35	13,585
NanoXML v1	3,497	Java	6	214
NanoXML v2	4,007	Java	7	214
NanoXML v3	4,608	Java	9	216
NanoXML v5	4,782	Java	8	216
XML security v1	21,613	Java	6	92
XML security v2	22,318	Java	6	94
XML security v3	19,895	Java	4	84

Table 4.9: Dataset descriptions

4.4.2 Evaluation Metrics

We use 40 association measures, Tarantula, and Ochiai to rank program elements based on their suspiciousness. Developers could then use this list to investigate the more suspicious program elements first. We assume that developers would be able to identify the buggy program element when they inspect it.

We consider two commonly used evaluation metrics: average percentage of code inspected to find all bugs, and proportion of bugs found when a given proportion of the code is inspected. We describe these two metrics in the following paragraphs.

Percentage of Code Inspected.

We evaluate the performance of the measures by the number of elements that are ranked as high or higher than the program element containing the fault/bug. For a suspiciousness score to be effective, buggy program elements should have a relatively larger value of suspiciousness scores than the non-buggy elements.

When a buggy program element has the same suspiciousness score as several other elements, the lowest rank of the elements that has this suspiciousness score is used as the rank of the buggy element. For example, consider the case where the two highest suspiciousness scores are 0.92 and 0.91, two elements (e.g., e_1 and e_2) have suspiciousness scores of 0.92, and 3 elements (e.g., e_3, e_4 and e_5) have suspiciousness scores of 0.91. Since we do not know how the programmer will traverse elements that have the same suspiciousness score, we use the worst case scenario where the programmer inspects all elements having the same score. In this case, e_1 and e_2 have a rank of 2, while e_3 to e_5 have a rank of 5. In the situation where a buggy version contains multiple bugs or one single bug that involves multiple program elements, we use the lowest rank among the buggy elements as it is the worst case rank that represents scenarios when programmers would like to find all the buggy elements by using the given list of most suspicious program elements.

Ranking program elements by using suspiciousness score is useful to evaluate the accuracy of a fault-localization approach. However, it may not be representative enough to know how programmers really locate the bugs with their expertise, as this ranking approach does not provide the context of the bugs to help programmers in understanding the root cause of the bugs, as the study by Parnin and Orso has shown [106]. An interesting future study would be to augment suspiciousness ranking with some additional information to more effectively guide programmers to locate all buggy program elements.

Given a list of most suspicious program elements produced by a suspiciousness measure, in this work, we assume that programmers would inspect the program elements from the most to the least suspicious ones in the list. Hence, an effective suspiciousness measure should rank buggy elements first. We then use this rank to compute the percentage of program elements that need to be inspected to find the buggy elements by the formula:

$$\frac{largest\ rank\ among\ the\ buggy\ elements}{total\ elements}$$

In our experiment, we thus choose to use basic block as the granularity of the elements and the above percentage is applied as the first accuracy criterion of the association measures. The accuracy of a particular measure to localize bug in all buggy versions of our datasets is then evaluated by calculating the overall mean of all percentages of code inspected for the measure which is the average of the percentages of code inspected of the measure for all buggy versions in our datasets. The smaller the overall mean, the more accurate the measure in localizing bug. However, the overall mean might not necessarily indicate that the measure would localize bugs with the same accuracy for all buggy versions. A measure could have a good accuracy in localizing one bug, but might not have a good accuracy in localizing other bugs. Thus, we also calculate the measure's overall standard deviation to evaluate its variance of percentages of code inspected for all buggy versions. The smaller the overall standard deviation, the better the overall mean reflects the accuracy of the measure because of less variation of percentage of code inspected for all buggy versions.

Proportion of Bugs Localized.

Next, we calculate the percentage of bugs that could be localized assuming that the developers are only willing to investigate a given proportion of code. To compute this measure we vary the proportion of code that the developers are willing to inspect and for each proportion we compute the percentage of bugs that can be localized.

4.4.3 Evaluation Results

We describe the accuracy of the association measures in comparison with Ochiai and Tarantula in the following paragraphs.

Percentage of Code Inspected.

We calculate the overall means and standard deviations of percentage of code inspected for the 40 association measures, Ochiai and Tarantula for all subject programs, as shown in Table 4.10. Among the 40 measures, Klosgen (M_{18}) achieves the smallest average percentage of code inspected i.e., 25.24%, which is better than Ochiai (i.e., 25.45%) and Tarantula (i.e., 27.13%). Intuitively, Klosgen also considers that a program element is more suspicious when it is executed more often by the failed test cases. It measures the suspiciousness of a program element by considering two main conditions. First, it measures the difference between the probability that a test case would fail, given that the element was executed, and the probability that a test case would fail during its execution (i.e., $P(B|A) - P(B)$). Second, it measures the difference between the probability that a program element would be executed, given that a test case was failed during its execution, and the probability that that program element would be executed (i.e., $P(A|B) - P(A)$). Then, Klosgen takes the maximum likelihood among these two conditions. The comparison between these two conditions can enrich the information to infer the faulty elements.

In addition, a number of the association measures has similar accuracy with Klosgen (M_{18}) and Ochiai (i.e., having a mean in the range of 25.66% and 26.24%): Collective Strength (M_16), Normalized Mutual Information (M_{26}), ϕ-Coefficient (M_1), Added Value (M_{15}), Two-Way Support (M_{28}), and Interestingness Weighting Dependency (M_{24}). We notice that 11 other measures can achieve similar accuracy as Tarantula (i.e., having a mean between 26.60% to 27.12%) i.e., J-Measure (M_6), Information Gain (M_{19}), Two-Way Support Variation (M_{29}), Example and Counterexample Rate (M_{34}), Confidence (M_9), Kappa (M_5), Sebag (M_{31}), Odd Multiplier (M_{33}), Interest (M_{12}), Zhang (M_{35}), and One-Way Support (M_{27}).

In our dataset, we have 15 subject programs where each subject program has a number of buggy versions. Since the accuracy of association measures in localizing bugs may vary for different programs, we then evaluate the accuracy of the measures in localizing bugs for each program. Tables 4.12 and 4.13 shows the mean and standard deviation of each measure in localizing faults for each subject program. Based on the results, almost half of the measures could localize bugs in buggy versions of $schedule$ when less than 10% of code is inspected. The measures generally require

Association Measures	Mean	StdDev	Association Measures	Mean	StdDev
Klosgen(M_{18})	25.24%	27.13%	Anderberg(M_{37})	27.60%	27.30%
Ochiai	25.45%	26.54%	Sorensen-Dice(M_{36})	27.61%	27.28%
Collective Strength(M_{16})	25.66%	26.86%	Ochiai II(M_{40})	28.23%	28.88%
Normalized Mutual Information(M_{26})	25.68%	26.38%	Gini Index(M_7)	28.23%	28.43%
ϕ-Coefficient(M_1)	25.73%	27.07%	Leverage(M_{22})	29.46%	28.12%
Added Value(M_{15})	25.74%	27.50%	Least Contradiction(M_{32})	33.42%	28.82%
Two-Way Support(M_{28})	26.03%	27.25%	Rogers and Tanimoto(M_{39})	33.47%	28.85%
Interestingness Weighting Dependency(M_{24})	26.24%	27.42%	Accuracy(M_{21})	33.68%	28.81%
J-Measure(M_6)	26.60%	27.51%	Simple-Matching(M_{38})	33.68%	28.81%
Information Gain(M_{19})	26.73%	27.53%	Odds Ratio(M_2)	41.61%	21.34%
Two-Way Support Variation(M_{29})	26.73%	27.53%	Yule's Q (M_3)	41.68%	21.45%
Example and Counterexample Rate(M_{34})	26.96%	27.12%	Yule's Y (M_4)	41.69%	21.44%
Confidence(M_9)	26.99%	27.25%	Certainty Factor(M_{14})	41.73%	21.26%
Kappa(M_5)	27.04%	27.21%	Conviction(M_{11})	41.73%	21.27%
Sebag(M_{31})	27.09%	27.18%	Relative Risk(M_{23})	44.82%	22.01%
Odd Multiplier(M_{33})	27.09%	27.20%	Laplace(M_{10})	45.34%	21.63%
Interest(M_{12})	27.10%	27.19%	Support(M_8)	45.44%	21.65%
Zhang(M_{35})	27.11%	27.17%	Goodman and Kruskal(M_{25})	45.74%	37.82%
One-Way Support(M_{27})	27.12%	27.26%	Coverage(M_{20})	47.02%	24.37%
Tarantula	27.13%	27.17%	Piatetsky-Shapiro's(M_{13})	57.37%	24.50%
Jaccard(M_{17})	27.47%	27.36%	Loevinger(M_{30})	57.57%	25.35%

Table 4.10: Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better)

more than 10% of the code to be inspected in order to localize bugs for the other five C programs (i.e., print_tokens, schedule2, replace, tcas, and tot_info) and the Java programs.

We observe that the average percentages of code inspected for all measures in localizing bugs for XML security programs are the same. This is due to the fact that program elements in those programs are either executed by both failed and passed test cases, or are not executed at all. Given the above fact, each element only has two possible dichotomy matrices (i.e., a matrix that is used by association measures to calculate the suspiciousness score of a program element). Table 4.11 shows two possible dichotomy matrices for buggy programs of XML Security version 1. As each possible dichotomy matrix has only two non-zero values, not much information can be inferred from the program spectra to rank the program elements, hence all measures perform similarly.

Hence, not much information can be inferred from the program spectra to rank the program elements and all measures perform similarly.

We also perform statistical tests for each pair of measures including Tarantula and Ochiai using Wilcoxon signed rank test [139] to evaluate if some measures are

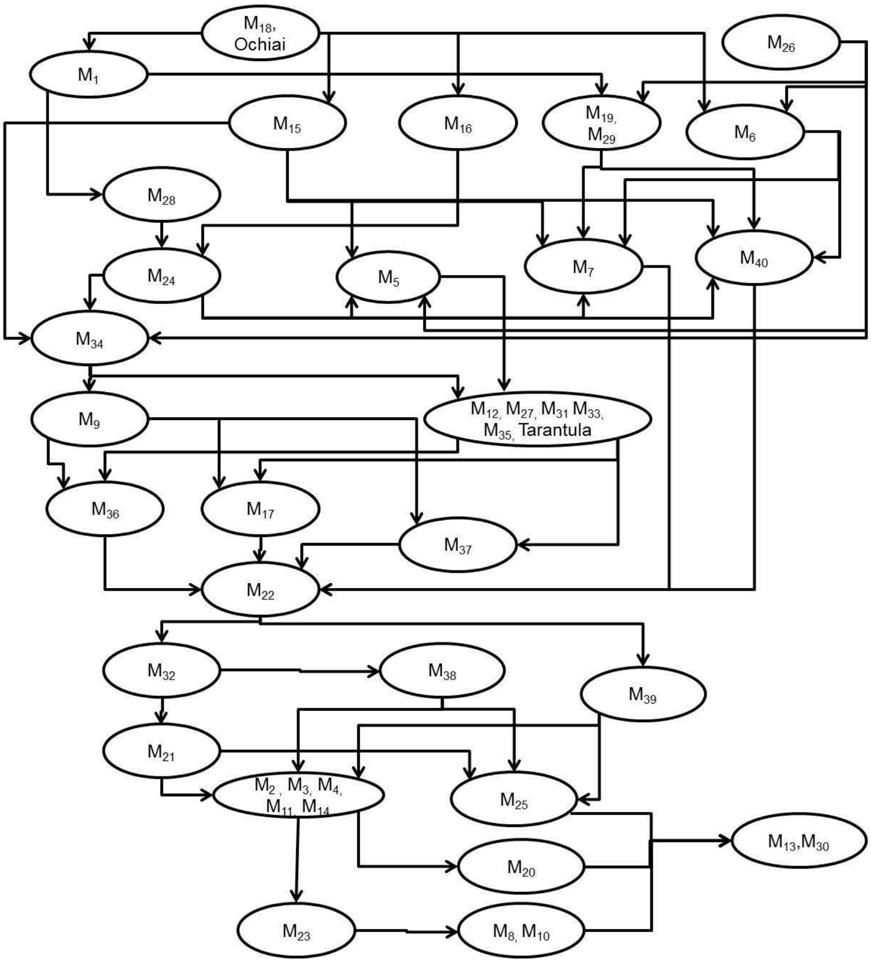


Figure 4.2: Accuracy partial order

	$_e$ Executed	$_e$ Not Executed
Test Passed	91	0
Test Failed	1	0

	$_e$ Executed	$_e$ Not Executed
Test Passed	0	91
Test Failed	0	1

Table 4.11: Two possible dichotomy matrices for buggy programs of XML Security version 1: [left] a program element is executed by all test cases; [right] a program element is not executed at all.

statistically significantly better than others (i.e., p-value is smaller than 0.05). The Wilcoxon test does not assume that the data should follow certain distribution.

We visualize the statistical significance relationship as a partial order as presented in Figure 4.2. A link from A to B in the partial order denotes that A is statistically significantly better than B. The relationships expressed in the partial order are transitive: if A outperforms B, and B outperforms C, then A outperforms C. To reduce the number of links in the partial order, we omit links that could be captured by this transitivity property.

Based on the partial order, Klosgen (M_{18}) and Normalized Mutual Information (M_{26}) have comparable performance as Ochiai. Klosgen (M_{18}), Normalized Mutual Information (M_{26}), and Ochiai are significantly better than Tarantula. We also notice that 7 other measures also perform significantly better than Tarantula i.e., ϕ-coefficient (M_1), Added Value (M_{15}), Collective Strength (M_{16}), Two-Way Support (M_{28}), Interestingness Weighting Dependency (M_{24}), Example and Counterexample Rate (M_{34}), and Kappa (M_5). Measures that perform comparably with Tarantula are Interest (M_{12}), One-Way Support (M_{27}), Sebag (M_{31}), Odds Multiplier (M_{33}), and Zhang (M_{35}). On the other hand, Piatetsky-Shapiro's (M_{13}), and Loevinger (M_{30}) perform worse than other measures for fault localization.

Proportion of Bugs Localized.

For each measure, we plot a curve showing the proportion of code that are investigated (x-axis) vs. the proportion of bugs localized (y-axis) for all dataset. We split

Association Measures	**Programs**							
	print_token	**print_token2**	**schedule**	**schedule2**	**replace**	**tcas**	**tot_info**	**space**
ϕ-Coefficient	21%(25%)	13%(17%)	9%(15%)	47%(24%)	13%(19%)	56%(27%)	19%(15%)	*10%(24%)
OddsRatio	43%(19%)	44%(2%)	49%(28%)	53%(7%)	34%(16%)	64%(15%)	49%(20%)	23%(20%)
Yule's Q	43%(19%)	44%(3%)	48%(28%)	53%(7%)	34%(16%)	64%(15%)	49%(20%)	23%(20%)
Yule's Y	44%(18%)	44%(3%)	48%(28%)	53%(7%)	34%(16%)	64%(15%)	49%(20%)	23%(20%)
Kappa	28%(26%)	16%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	21%(16%)	*10%(24%)
J-Measure	22%(27%)	11%(13%)	14%(19%)	49%(23%)	*12%(18%)	56%(29%)	18%(15%)	12%(25%)
Gini Index	26%(32%)	16%(22%)	12%(12%)	58%(26%)	13%(19%)	58%(29%)	24%(19%)	12%(25%)
Support	43%(17%)	45%(3%)	50%(30%)	56%(8%)	34%(16%)	67%(14%)	51%(20%)	26%(21%)
Confidence	29%(26%)	13%(18%)	7%(6%)	51%(24%)	16%(21%)	57%(27%)	23%(16%)	11%(24%)
Laplace	43%(17%)	45%(3%)	50%(29%)	56%(8%)	35%(16%)	67%(14%)	50%(20%)	25%(20%)
Conviction	42%(16%)	44%(3%)	48%(28%)	53%(7%)	34%(16%)	64%(15%)	49%(20%)	24%(20%)
Interest	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Piatetsky-Shapiro's	61%(32%)	77%(16%)	71%(29%)	68%(28%)	64%(22%)	66%(26%)	63%(28%)	50%(18%)
Certainty Factor	42%(16%)	44%(3%)	48%(28%)	53%(7%)	34%(16%)	64%(15%)	49%(20%)	24%(20%)
Added Value	23%(28%)	14%(19%)	7%(10%)	50%(25%)	13%(20%)	56%(28%)	18%(15%)	11%(24%)
Collective Strength	24%(31%)	13%(16%)	11%(20%)	44%(23%)	13%(18%)	55%(27%)	18%(15%)	*10%(24%)
Jaccard	26%(26%)	12%(18%)	7%(6%)	50%(24%)	15%(21%)	56%(27%)	21%(15%)	11%(24%)
Klosgen	21%(25%)	11%(15%)	8%(12%)	48%(24%)	*12%(19%)	55%(28%)	*17%(13%)	11%(24%)
Information Gain	22%(27%)	11%(14%)	14%(19%)	49%(23%)	*12%(18%)	56%(29%)	19%(16%)	12%(25%)
Coverage	63%(26%)	63%(20%)	69%(30%)	*36%(26%)	53%(23%)	*47%(25%)	51%(28%)	40%(24%)
Accuracy	38%(26%)	28%(22%)	11%(5%)	60%(24%)	27%(22%)	60%(29%)	37%(29%)	15%(26%)
Leverage	30%(28%)	20%(22%)	7%(4%)	55%(26%)	17%(20%)	59%(28%)	28%(23%)	14%(25%)
Relative Risk	44%(19%)	45%(3%)	50%(28%)	56%(8%)	35%(16%)	68%(14%)	50%(20%)	23%(20%)
Int. Weighting Dependency	24%(28%)	15%(20%)	8%(9%)	50%(24%)	14%(20%)	56%(27%)	21%(16%)	*10%(24%)
Goodman and Kruskal	35%(44%)	30%(44%)	67%(38%)	83%(9%)	40%(40%)	70%(30%)	66%(41%)	19%(31%)
Normalized Mutual Info.	23%(31%)	*9%(11%)	16%(27%)	41%(20%)	*12%(17%)	54%(27%)	*17%(13%)	11%(24%)
One-Way Support	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(20%)	57%(27%)	22%(16%)	11%(24%)
Two-Way Support	23%(28%)	14%(19%)	8%(10%)	50%(25%)	13%(18%)	56%(28%)	20%(15%)	*10%(24%)
Two-Way Support Variation	22%(27%)	11%(14%)	14%(19%)	49%(23%)	*12%(18%)	56%(29%)	19%(16%)	12%(25%)
Loevinger	71%(26%)	78%(21%)	74%(30%)	44%(26%)	65%(22%)	53%(23%)	70%(26%)	61%(25%)
Sebag	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Least Contradiction	34%(30%)	28%(22%)	11%(5%)	59%(24%)	26%(21%)	60%(29%)	36%(29%)	15%(26%)
Odd Multiplier	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Example and Counter.	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Zhang	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Sorensen-Dice	27%(26%)	16%(20%)	7%(6%)	50%(24%)	15%(21%)	56%(27%)	20%(14%)	11%(24%)
Anderberg	26%(26%)	16%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	20%(14%)	11%(25%)
Simple-Matching	38%(26%)	28%(22%)	11%(5%)	60%(24%)	27%(22%)	60%(29%)	37%(29%)	15%(26%)
Rogers and Tanimoto	34%(30%)	28%(22%)	11%(5%)	60%(24%)	27%(22%)	60%(29%)	36%(29%)	15%(26%)
Ochiai II	27%(32%)	16%(21%)	*6%(4%)	57%(26%)	15%(21%)	57%(28%)	28%(27%)	11%(24%)
Tarantula	29%(26%)	17%(20%)	7%(6%)	51%(24%)	16%(21%)	56%(27%)	22%(16%)	11%(24%)
Ochiai	*20%(29%)	*9%(10%)	13%(28%)	42%(22%)	*12%(19%)	53%(26%)	*17%(12%)	11%(24%)

Table 4.12: Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the C programs. The star (*) marks the measure(s) with the lowest mean and the underline marks measures that have a mean below 10%.

Association Measures	**Programs**						
	Nano_v1	**Nano_v2**	**Nano_v3**	**Nano_v5**	**XML-sec_v1**	**XML-sec_v2**	**XML-sec_v3**
ϕ-Coefficient	*21%(28%)	32%(25%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
OddsRatio	45%(18%)	43%(18%)	44%(19%)	23%(10%)	29%(0%)	31%(0%)	40%(0%)
Yule's Q	44%(18%)	43%(18%)	44%(19%)	25%(14%)	29%(0%)	31%(0%)	40%(0%)
Yule's Y	44%(18%)	43%(18%)	44%(19%)	25%(14%)	29%(0%)	31%(0%)	40%(0%)
Kappa	*21%(28%)	32%(25%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
J-Measure	25%(30%)	31%(26%)	*31%(30%)	26%(17%)	29%(0%)	31%(0%)	40%(0%)
Gini Index	25%(30%)	32%(25%)	*31%(30%)	26%(19%)	29%(0%)	31%(0%)	40%(0%)
Support	67%(11%)	51%(12%)	55%(10%)	41%(18%)	29%(0%)	31%(0%)	40%(0%)
Confidence	*21%(28%)	26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Laplace	67%(11%)	51%(12%)	55%(10%)	41%(18%)	29%(0%)	31%(0%)	40%(0%)
Conviction	44%(18%)	42%(18%)	44%(19%)	25%(14%)	29%(0%)	31%(0%)	40%(0%)
Interest	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Piatetsky-Shapiro's	52%(20%)	46%(16%)	36%(13%)	46%(22%)	29%(0%)	31%(0%)	40%(0%)
Certainty Factor	44%(18%)	42%(18%)	44%(19%)	25%(14%)	29%(0%)	31%(0%)	40%(0%)
Added Value	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Collective Strength	*21%(28%)	31%(26%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Jaccard	*21%(28%)	53%(9%)	*31%(30%)	19%(15%)	29%(0%)	31%(0%)	40%(0%)
Klosgen	*21%(28%)	28%(28%)	*31%(30%)	19%(14%)	29%(0%)	31%(0%)	40%(0%)
Information Gain	25%(30%)	32%(25%)	*31%(30%)	26%(18%)	29%(0%)	31%(0%)	40%(0%)
Coverage	59%(21%)	47%(19%)	34%(15%)	40%(19%)	29%(0%)	31%(0%)	40%(0%)
Accuracy	25%(30%)	38%(27%)	32%(30%)	24%(21%)	29%(0%)	31%(0%)	40%(0%)
Leverage	25%(30%)	28%(28%)	*31%(30%)	24%(17%)	29%(0%)	31%(0%)	40%(0%)
Relative Risk	67%(11%)	51%(12%)	55%(10%)	40%(16%)	29%(0%)	31%(0%)	40%(0%)
Int. Weighting Dependency	*21%(28%)	27%(28%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Goodman and Kruskal	29%(35%)	38%(27%)	40%(30%)	42%(30%)	29%(0%)	31%(0%)	40%(0%)
Normalized Mutual Info.	*21%(28%)	43%(18%)	*31%(30%)	21%(14%)	29%(0%)	31%(0%)	40%(0%)
One-Way Support	*21%(28%)	*26%(29%)	*31%(30%)	21%(18%)	29%(0%)	31%(0%)	40%(0%)
Two-Way Support	*21%(28%)	30%(27%)	*31%(30%)	20%(13%)	29%(0%)	31%(0%)	40%(0%)
Two-Way Support Variation	25%(30%)	32%(25%)	31%(30%)	26%(18%)	29%(0%)	31%(0%)	40%(0%)
Loevinger	63%(20%)	50%(21%)	35%(15%)	45%(21%)	29%(0%)	31%(0%)	40%(0%)
Sebag	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Least Contradiction	25%(30%)	38%(27%)	32%(30%)	24%(21%)	29%(0%)	31%(0%)	40%(0%)
Odd Multiplier	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Example and counter.	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Zhang	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Sorensen-Dice	*21%(28%)	53%(9%)	42%(35%)	19%(15%)	29%(0%)	31%(0%)	40%(0%)
Anderberg	*21%(28%)	53%(9%)	42%(35%)	19%(15%)	29%(0%)	31%(0%)	40%(0%)
Simple-Matching	25%(30%)	38%(27%)	43%(35%)	24%(21%)	29%(0%)	31%(0%)	40%(0%)
Rogers and Tanimoto	25%(30%)	38%(27%)	43%(35%)	24%(21%)	29%(0%)	31%(0%)	40%(0%)
Ochiai II	25%(30%)	32%(25%)	42%(35%)	24%(18%)	29%(0%)	31%(0%)	40%(0%)
Tarantula	*21%(28%)	*26%(29%)	*31%(30%)	20%(15%)	29%(0%)	31%(0%)	40%(0%)
Ochiai	*21%(28%)	53%(10%)	*31%(30%)	*18%(15%)	29%(0%)	31%(0%)	40%(0%)

Table 4.13: Detailed means and standard deviations (in parentheses) of percentages of code inspected to find all bugs in the Java programs.

the large graphs into several smaller graphs so that measures that have similar accuracies are grouped together, as shown in Figures 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8. For each graph, we compare the percentage of bugs localized by a number of association measures with Tarantula and Ochiai.

Figure 4.3 shows three association measures that perform better than Ochiai and Tarantula or as good as Ochiai. When only 10% of program elements are inspected, Klosgen (M_{18}) and Added Value (M_{15}) could localize more bugs than Tarantula and Ochiai—they could localize 47% and 48% of the bugs respectively. Tarantula and Ochiai could localize 39% and 45% of the bugs respectively. Two-Way Support (M_{28}) could localize the same proportion of bugs as Ochiai.

When only 10% of program elements are inspected, a number of association measures could perform better than Tarantula even though they are not as good as Ochiai, as shown in Figures 4.5 and 4.6. In addition, a number of association measures could have similar accuracy with Tarantula, as shown in Figure 4.4. When 10% of program elements are inspected, these measures could localize 39 to 40% of the bugs, while the rest of the measures could not perform as good as Tarantula and Ochiai. See Figures 4.7 and 4.8 for the proportion of bugs localized for the measures that perform worse than Tarantula and Ochiai.

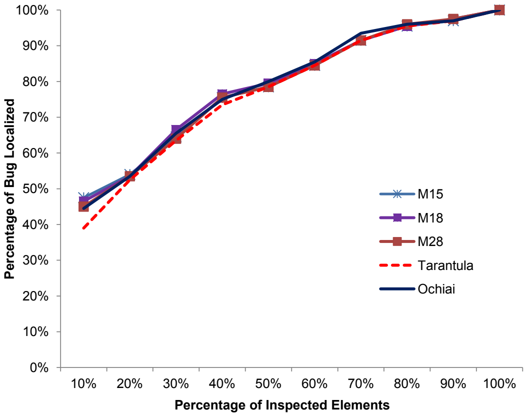


Figure 4.3: Comparing M_{15}, M_{18}, and M_{28} with Ochiai and Tarantula for all datasets

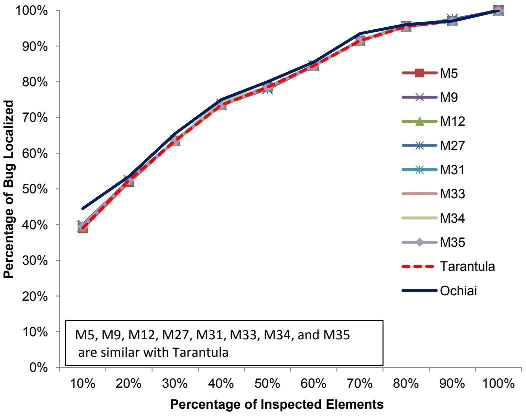


Figure 4.4: Comparing M_5, M_9, M_{12}, M_{27}, M_{31}, M_{33} - M_{35} with Ochiai and Tarantula for all datasets

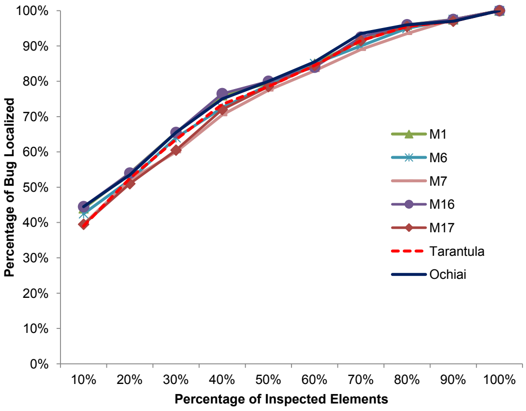


Figure 4.5: Comparing M_1, M_6, M_7, M_{16}, M_{17} with Ochiai and Tarantula for all datasets

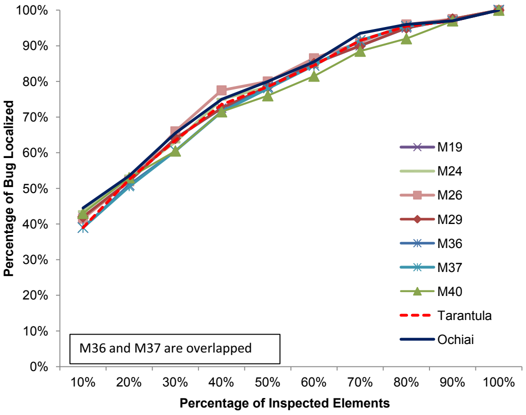


Figure 4.6: Comparing M_{19}, M_{24}, M_{26}, M_{29}, M_{36}, M_{37}, M_{40} with Ochiai and Tarantula for all datasets

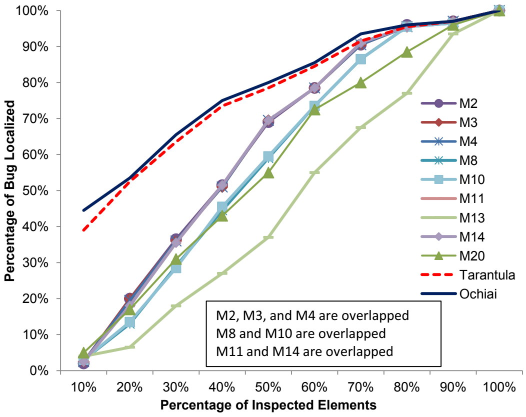


Figure 4.7: Comparing M_2 - M_4, M_8, M_{10}, M_{11}, M_{13}, M_{14}, M_{20} with Ochiai and Tarantula for all datasets

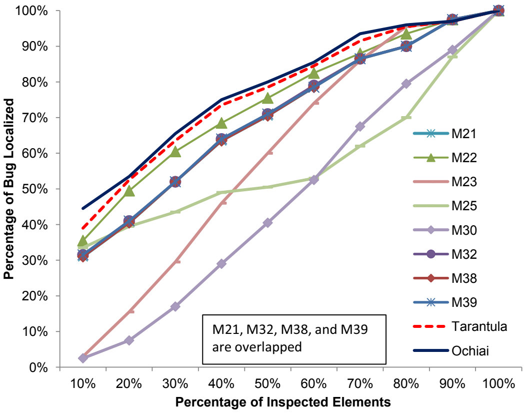


Figure 4.8: Comparing M_{21} - M_{23}, M_{25}, M_{30}, M_{32}, M_{38}, M_{39} with Ochiai and Tarantula for all datasets

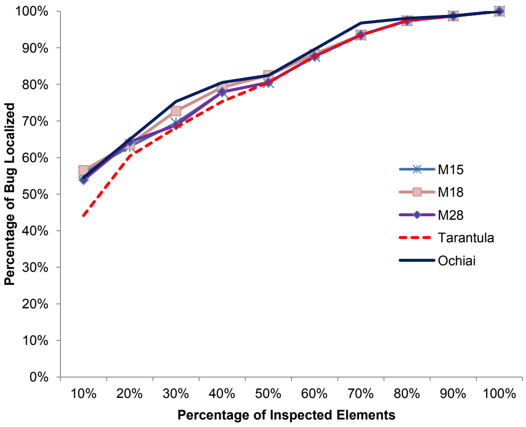


Figure 4.9: Comparing M_{15}, M_{18}, and M_{28} with Ochiai and Tarantula for C programs

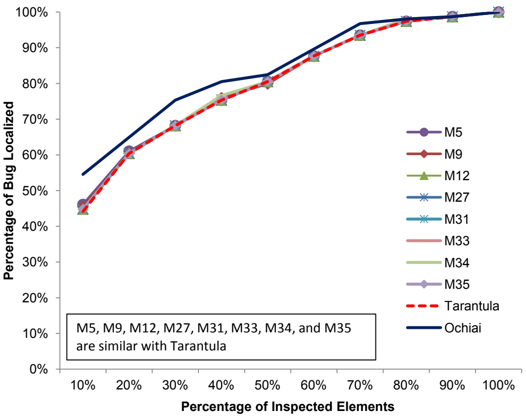


Figure 4.10: Comparing M_5, M_9, M_{12}, M_{27}, M_{31}, M_{33} - M_{35} with Ochiai and Tarantula for C programs

4.4.4 Effectiveness for Various Programming Languages.

In this section, we evaluate the accuracy of measures in localizing faults for different programming languages (C and Java) in terms of the percentage of code inspected and the proportion of bugs localized. Based on the percentage of code inspected, we generate two partial orders separately for the C and Java programs to evaluate the measures that perform better in each of the programming languages. We highlight measures that are at the top of the partial orders (i.e., no other measures perform statistically significantly better than them).

We find that for both C and Java programs, Ochiai and Klosgen (M_{18}) are at the top of the partial order. Besides these two measures, a number of other measures are also at the top of the partial order in localizing bugs for Java programs, namely Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Tarantula, Interestingness Weighting Dependency (M_{24}), and Normalized Mutual Information (M_{26}).

Localizing Bugs in C Programs.

Based on the proportion of bugs localized, Added Value (M_{15}) and Klosgen (M_{18}) could localize more bugs in C programs as compared to Ochiai and Tarantula when 10% of code inspected. Added Value (M_{15}) and Klosgen (M_{18}) could

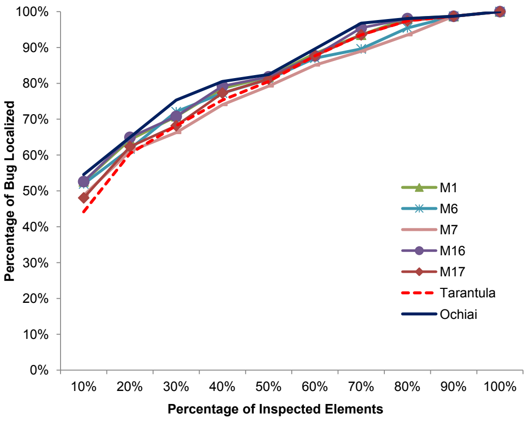


Figure 4.11: Comparing M_1, M_6, M_7, M_{16}, M_{17} with Ochiai and Tarantula for C programs

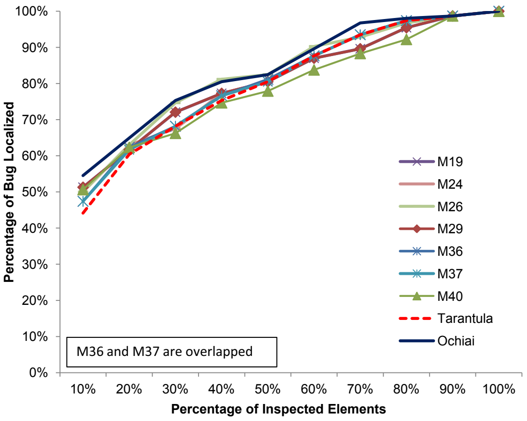


Figure 4.12: Comparing M_{19}, M_{24}, M_{26}, M_{29}, M_{36}, M_{37}, M_{40} with Ochiai and Tarantula for C programs

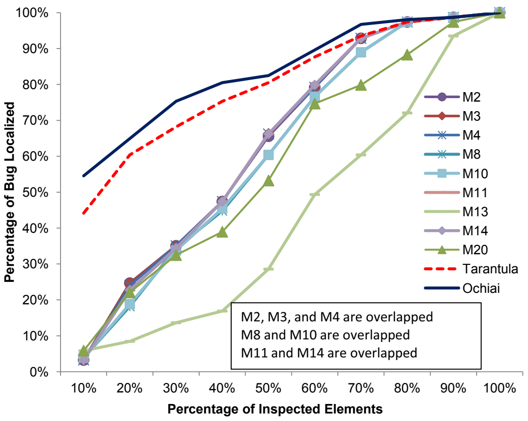


Figure 4.13: Comparing M_2 - M_4, M_8, M_{10}, M_{11}, M_{13}, M_{14}, M_{20} with Ochiai and Tarantula for C programs

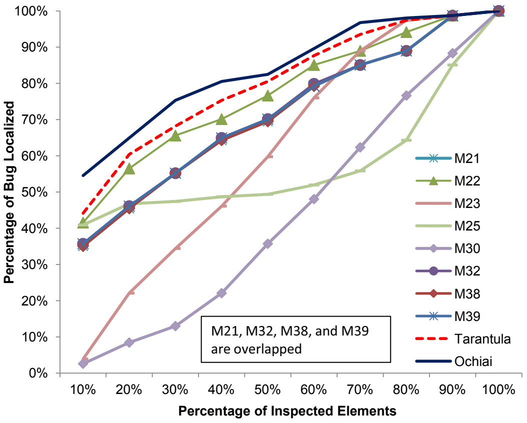


Figure 4.14: Comparing M_{21} - M_{23}, M_{25}, M_{30}, M_{32}, M_{38}, M_{39} with Ochiai and Tarantula for C programs

localize 54% and 53% of the bugs respectively, while Tarantula and Ochiai could localize 43% and 52% of total bugs in C programs. Two-Way Support (M_{28}) could localize 51% of the bugs, which is slightly lower than Ochiai. See Figure 4.9 for the proportion of bugs localized for different percentage of code inspected of these measures.

Figures 4.11 and 4.12 show measures that perform better than Tarantula but not as good as Ochiai. When only 10% of program elements are inspected, they could localized 45% to 51% of the bugs. The measures are ϕ-Coefficient (M_1), Collective Strength (M_{16}), J-Measure (M_6), Information Gain (M_{19}), Interestingness Weighting Dependency (M_{24}), Normalized Mutual Information (M_{26}), Two-Way Support Variant(M_{29}), Ochiai II (M_{40}), Gini Index (M_7), Jaccard (M_{17}), Sorensen-Dice (M_{36}), and Anderberg (M_{37}).

On the other hand, a number of the measures have similar performance with Tarantula, as presented in Figure 4.10. They localize 43-44% of the bugs when 10% of the program elements are inspected. The rest of the association measures could not outperform Tarantula and Ochiai. See Figures 4.13 and 4.14 for the proportion of bugs localized when we use these measures.

Localizing Bugs in Java Programs.

For Java programs, Tarantula can have better accuracy in localizing bugs as compared to Ochiai. When 10% of program elements are inspected, Tarantula could localize 26% of the bugs, while Ochiai could only localize 20% of the bugs. Figure 4.15 shows measures that have similar performance as Tarantula. They could localize 26% of the bugs within 10% of code inspected i.e., Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), and Zhang (M_{35}).

A number of association measures could also have performance between arantula and Ochiai when 10% of program elements are inspected, as shown in Figures 4.16, 4.17, and 4.18. The association measures that could not perform as good as compare to Ochiai and Tarantula are shown in Figures 4.19 and 4.20.

Based on the above results, a number of association measures could perform as good as Tarantula and Ochiai to localize the bugs that reside in programs written in different programming languages. The differences in accuracies for localizing bugs in C and Java programs possibly imply that there could be a specific characteristic of spectra produced by different program languages that could benefit or disadvantage spectrum-based fault localization. For example in Java, due to the object oriented model, more program elements (e.g., class constructors) could be executing together with the faulty ones, thus obscuring the suspiciousness scores for the actual faults.

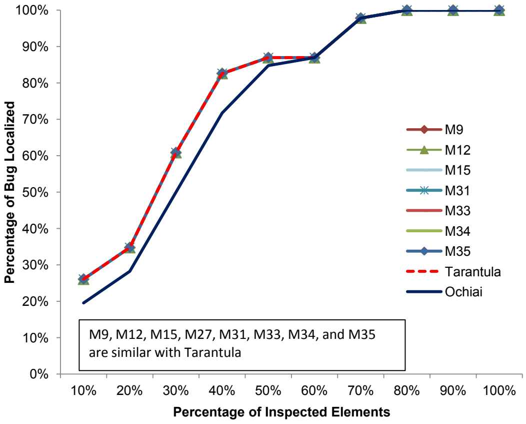


Figure 4.15: Comparing M_9, M_{12}, M_{15}, M_{31}, M_{33}, M_{34}, and M_{35} with Ochiai and Tarantula for Java programs

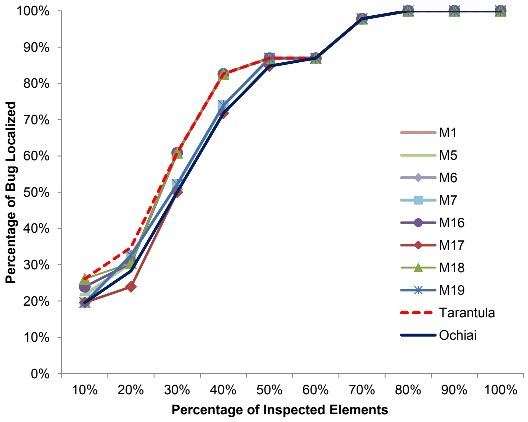


Figure 4.16: Comparing M_1, M_5, M_6, M_7, M_{16}, M_{17}, M_{18}, M_{19} with Ochiai and Tarantula for Java programs

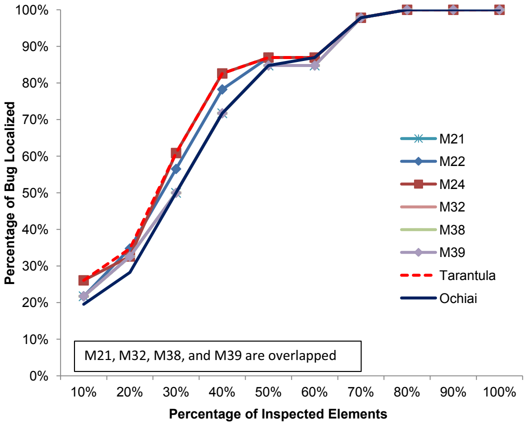


Figure 4.17: Comparing M_{21}, M_{22}, M_{24}, M_{32}, M_{38}, M_{39} with Ochiai and Tarantula for Java programs

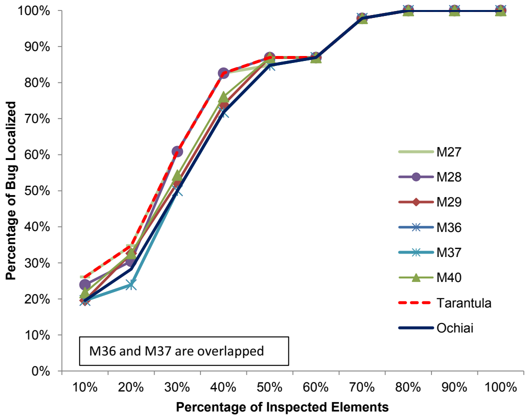


Figure 4.18: Comparing M_{27} - M_{29}, M_{35} - M_{37}, M_{40} with Ochiai and Tarantula for Java programs

4.4.5 Effectiveness for Various Kinds of Bugs.

In this section, we categorize the bugs into several groups based on the bug-fix categorization by Pan et al. [105]. The following paragraphs describe our bug categories and present the effectiveness of the various association measures, including Tarantula, and Ochiai on different bug categories.

Bug Categories.

Pan et al. [105] describe a number of bug categories based on the way bugs are fixed. In this work, we refer to their categories to analyze the bugs in our programs. We

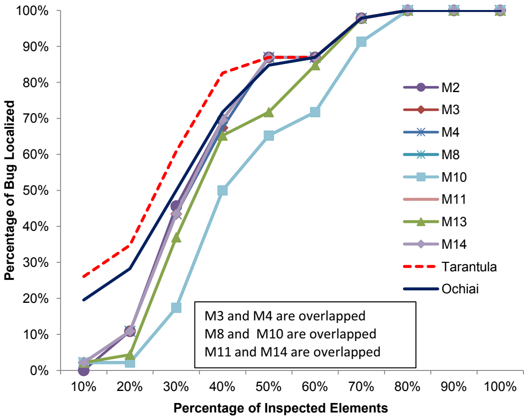


Figure 4.19: Comparing M_2 - M_4, M_8, M_{10} ,M_{11} - M_{14} with Ochiai and Tarantula for Java programs

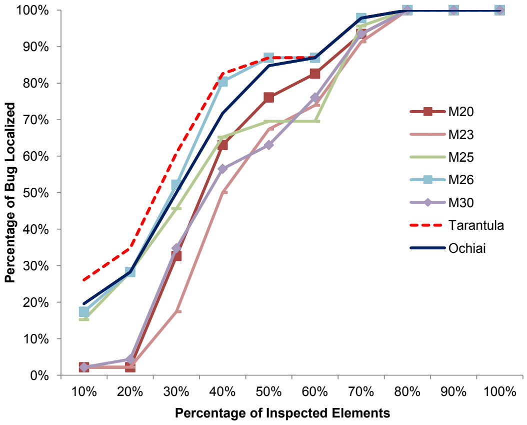


Figure 4.20: Comparing M_{20}, M_{23}, M_{25}, M_{26}, M_{30} with Ochiai and Tarantula for Java programs

Bug category	**Total version**
Addition/removal of conditional statement (CH-CS)	28
Addition/removal of non-conditional statement (CH-NCS)	10
Change in method calls (CH-MC)	26
Change of assignment expression (AS-CE)	60
Change of if condition expression (IF-CC)	44
Change of loop predicate (LP-CC)	11
Change of return expression (CH-RET)	18
Others (OTH)	3

Table 4.14: Bug categories

categorize the bug in each buggy version based on the bug fix. Buggy versions for which there is no category are put in the category ”other” (OTH). In addition to bug categories by Pan et al. [105], we add the category change of return expression (CH-RET). Thus, we categorize the bug in our programs into 8 categories. Table 4.14 shows the categories and the number of buggy versions for each type of bugs.

We categorize a bug into addition or removal of conditional statement category (CH-CS) when a bug could be fixed by adding a conditional check statement (e.g. if statement) or removing an inappropriate check statement. This type of bug occurs when there is a missing precondition or postcondition check of some variables, or an extraneous conditional check. Bugs that could be fixed by adding or by removing statement that is not a conditional check statement are categorized into addition/removal of non-conditional statement (CH-NCS) category. An example of this type of bug is missing or extraneous assignment statements.

The Change in method calls (CH-MC) category includes bugs that can be fixed by adding or removing a method call in a program, or by changing the parameter values of a method call. Bugs that could be fixed by changing the right hand side of an assignment expression are categorized into change of assignment expression (AS-CE) category. When a bug could be fixed by modifying a conditional expression within an if statement, we categorize the bug into change of if condition expression (IF-CC) category. Similarly, when a bug could be fixed by modifying a conditional expression in a looping statement, then we put this bug into change of loop predicate (LP-CC) category. When a bug could be fixed by changing the value or the expression in a return statement, then we categorize the bug into change of return expression (CH-RET) category.

We create a category named other (OTH) to include other bugs that are not covered by the above categories, e.g., a bug is fixed by changing an if statement to a for loop statement, etc. As very few bugs (i.e., three bugs) in our dataset are categorized as other (OTH), we exclude the bugs that belong to this category from our evaluation.

Effectiveness.

We first evaluate the effectiveness of the various measures on each category by the first accuracy criterion (i.e., percentage of code inspected). Based on this criterion, we create several partial orders based on statistically significantly better relationships among the measures. We highlight measures that are at the top of the partial orders as shown in Table 4.15.

Based on the partial orders, Klosgen (M_{18}) is the only measure that is at the top of the partial orders for all bug categories. Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}) are at the top of the partial orders of six bug categories.

Next, we evaluate the effectiveness of the measures on each bug category by the second accuracy criterion (i.e., proportion of bugs localized). We compute the

Bug category	**Top Measures**
Addition/removal of conditional statement (CH-CS)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Tarantula, ϕ-Coefficient (M_1), Kappa (M_5), J-Measure (M_6), Confidence (M_9), Added Value (M_{15}), Collective Strength (M_{16}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_{31}), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Sorensen-Dice (M_{36}), Anderberg (M_{37}), Interest(M_{12}), Leverage(M_{22}), and Jaccard(M_{17})
Addition/removal of non-conditional statement (CH-NCS)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), ϕ-Coefficient (M_1), Kappa (M_5), Gini Index (M_7), J-Measure (M_6), Added Value (M_{15}), Collective Strength (M_{16}), Jaccard (M_{17}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{28}), Anderberg (M_{37}), and Ochiai II (M_{40})
Change in method calls (CH-MC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Tarantula, ϕ-Coefficient (M_1), Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Collective Strength (M_{16}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_{31}), Odd Multiplier (M_{33}), Example and Counterexample Rate (M_{34}), and Zhang (M_{35})
Change of assignment expression (AS-CE)	Klosgen (M_{18}), Ochiai, and Normalized Mutual Information (M_{26})
Change of if condition expression (IF-CC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), and J-Measure (M_6)
Change of loop predicate (LP-CC)	Klosgen (M_{18}), Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{26}), Two-Way Support Variation (M_{29}), Tarantula, ϕ-Coefficient (M_1), Kappa (M_5), Collective Strength (M_{16}), J-Measure (M_6), Gini Index (M_7), Confidence (M_9), Added Value (M_{15}), Jaccard (M_{17}), Interestingness Weighting Dependency (M_{24}), Goodman Kruskal (M_{25}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Sorensen-Dice (M_{36}), Anderberg (M_{37}), Interest(M_{12}), Leverage(M_{22}), Odd Multiplier (M_{33}), Support(M_8), and Ochiai II (M_{40})
Change of return expression (CH-RET)	Klosgen (M_{18}), Information Gain (M_{19}), Two-Way Support Variation (M_{29}), J-Measure (M_6), Gini Index (M_7), Added Value (M_{15}), Interestingness Weighting Dependency (M_{24}), Two-Way Support (M_{28}), Accuracy(M_{21}), Least Contradiction(M_{32}), Simple-Matching (M_{38}), and Rogers and Tanimoto (M_{39})

Table 4.15: Measures that are at the top of the partial orders for each bug category. The underline marks measures that are at the top of the partial orders of more than five bug categories.

percentage of bugs localized when up to 10% of program elements are inspected, as shown in Table 4.16.

For each category shown in Table 4.16, different measures have different effectiveness in localizing bugs when 10% of the code are inspected. The star (*) marks the measures that could localize the largest number of bugs in each category. We notice that Added Value (M_{15}) could localize the largest number of bugs in six categories, while Klosgen (M_{15}) and Two-Way Support (M_{28}) could localize the largest number of bugs in four bug categories. Ochiai and Tarantula could localize the largest number of bugs in 3 and 1 categories respectively.

By only inspecting up to 10% of the program elements, bugs in change in method calls (CH-MC) category are not easy to be localized. The best measure for this category could only localize 35% of these bugs. Our initial investigation shows that the bugs in this category often involve more than one faulty line. Hence multiple program elements must be inspected in order to localize the bugs that belong to this category.

On the other hand, the measures can better localize three types of bugs i.e., the bugs in addition or removal of non-conditional statement (CH-NCS), change of loop predicate (LP-CC), and change of return expression (CH-RET) categories. The best measures for each of the three categories can localize up to 70%, 64%, and 61% of the bugs respectively. We observe that the bugs in these categories often involve one faulty line which is more localizable than the bugs in the change in method calls (CH-MC) category. As for the bugs in change of assignment expression (AS-CE) category and bugs in addition or removal of conditional statement (CH-CS) category, the best measures of each of the two categories can only localize 48% and 43% of the bugs.

In general, most of the measures could better localize bugs that can be categorized into addition or removal of non-conditional statement (CH-NCS), change of loop predicate (LP-CC), and change of return statement (CH-RET) categories. They could localize at least 50% of these types of bugs. Only 11 measures that

Measures	**CH-CS**	**CH-NCS**	**CH-MC**	**AS-CE**	**IF-CC**	**LP-CC**	**CH-RET**
ϕ-Coefficient(M_1)	*43%	*70%	31%	47%	41%	55%	50%
Odds Ratio(M_2)	0%	0%	0%	3%	5%	0%	0%
Yule's Q(M_3)	0%	0%	4%	3%	5%	0%	0%
Yule's Y(M_4)	0%	0%	4%	3%	5%	0%	0%
Kappa(M_5)	39%	60%	27%	43%	30%	55%	50%
J-Measure(M_6)	*43%	60%	23%	45%	*48%	36%	50%
Gini Index(M_7)	39%	60%	23%	45%	36%	36%	50%
Support(M_8)	0%	0%	4%	5%	5%	0%	0%
Confidence(M_9)	39%	60%	*35%	40%	32%	55%	56%
Laplace(M_{10})	0%	0%	4%	5%	5%	0%	0%
Conviction(M_{11})	0%	0%	4%	3%	5%	0%	0%
Interest(M_{12})	39%	60%	*35%	40%	30%	55%	56%
Pietatsky-Shapiro(M_{13})	4%	0%	4%	8%	2%	0%	0%
Certainty Factor(M_{14})	0%	0%	4%	3%	5%	0%	0%
Added Value(M_{15})	*43%	*70%	*35%	*48%	*48%	*64%	56%
Collective Strength(M_{16})	*43%	*70%	31%	*48%	39%	*64%	50%
Jaccard(M_{17})	39%	60%	27%	43%	32%	55%	50%
Klosgen(M_{18})	39%	*70%	31%	*48%	*48%	55%	*61%
Information Gain(M_{19})	*43%	60%	23%	45%	45%	36%	50%
Coverage(M_{20})	4%	0%	8%	5%	7%	9%	0%
Accuracy(M_{21})	36%	40%	*35%	37%	18%	18%	39%
Leverage(M_{22})	36%	40%	31%	35%	27%	55%	56%
Relative Risk(M_{23})	0%	0%	4%	5%	5%	0%	0%
Int. Weighting Dependency(M_{24})	39%	*70%	*35%	43%	39%	*64%	56%
GoodMan and Kruskal(M_{25})	29%	30%	23%	42%	25%	45%	50%
Normalized Mutual Info.(M_{26})	*43%	60%	27%	43%	43%	36%	50%
One-Way Support(M_{27})	39%	60%	*35%	40%	30%	55%	56%
Two-Way Support(M_{28})	*43%	*70%	31%	47%	43%	*64%	50%
Two-Way Support Variation(M_{29})	*43%	60%	23%	45%	45%	36%	50%
Loevinger(M_{30})	0%	0%	4%	5%	2%	0%	0%
Sebag(M_{31})	39%	60%	*35%	40%	30%	55%	56%
Least Contradiction(M_{32})	36%	40%	*35%	37%	20%	18%	39%
Odd Multiplier(M_{33})	39%	60%	*35%	40%	30%	55%	56%
Example and Counter.(M_{34})	39%	60%	*35%	40%	30%	55%	56%
Zhang(M_{35})	39%	60%	*35%	40%	30%	55%	56%
Sorensen-Dice (M_{36})	39%	60%	27%	43%	30%	55%	50%
Anderberg (M_{37})	39%	60%	27%	43%	30%	55%	50%
Simple-Matching (M_{38})	36%	40%	*35%	37%	18%	18%	39%
Rogers and Tanimoto (M_{39})	36%	40%	*35%	37%	20%	18%	39%
Ochiai II (M_{40})	39%	*70%	31%	45%	39%	*64%	50%
Tarantula	39%	60%	*35%	40%	27%	55%	56%
Ochiai	*43%	*70%	27%	45%	*48%	55%	50%

Table 4.16: Effectiveness of bug localization for (M_1) to (M_{40}) for each bug category when up to 10% program elements are inspected. The star (*) marks the measures that localize bugs the most for each category.

could only localize a small number of bugs for each bug category i.e., Coverage (M_{20}), Pietatsky-Shapiro (M_{13}), Support (M_8), Laplace (M_{10}), Relative Risk (M_{23}), Yule's Q (M_3), Yule's Y (M_4), Conviction (M_{11}), Certainty Factor (M_{14}), Loevinger (M_{30}), and Odds Ratio (M_2).

4.4.6 Effectiveness on Multiple-bug Versions

We evaluate the effectiveness of association measures, Tarantula and Ochiai, in localizing multiple bugs in programs. We refer the programs in which more than one bug could reside as multiple-bug versions. A multiple-bug version of a program contains a number of bugs where each bug only involves one line in the program and different bugs affect different lines [5, 66].

We generate 173 multiple-bug versions of C programs as shown in Table 4.17. To generate these versions of a subject program, we first take the bugs that only involve one line in the single-bug versions of the subject program. Among these bugs, we randomly choose a number of bugs (i.e., two or five bugs) and concatenate the code fragments containing bugs, to make a new version. We refer this version to as multiple-bug version. We ensure that each bug appears in at least one of the generated multiple-bug versions. Also, the bugs that we choose to be concatenated should not involve the same line of code to maintain the effect of each bug.

As the print_token and schedule2 programs only have four and five bugs, respectively, that involve one line, the number of possible single-bug fragments that can be concatenated in each of their multiple-bug versions is limited. In this work, we only concatenate two possible single-bug fragments in every multiple-bug version of those two programs so that we could have more versions that contain more than one bug. For each of those two programs, we generate ten multiple-bug versions. Thus, we have 20 multiple-bug versions that contain two bugs (minimum number of multiple bugs).

Dataset	**Num. of Bug in a Version**	**Num. of Single Bug in Dataset**	**Language**	**Num. of Buggy Version**
print_token	2	4	C	10
schedule2	2	5	C	10
print_token2	5	9	C	10
replace	5	25	C	32
schedule	5	7	C	9
tcas	5	30	C	41
tot_info	5	19	C	23
space	5	19	C	38

Table 4.17: Multiple-bug datasets

For the other programs, we randomly choose five bugs and concatenate the code fragments containing bugs in every multiple-bug version of these programs. We generate the same number of multiple-bug versions as the number of its single-bug versions (including single-bug versions that contain bugs that involve multiple lines) so that we could have similar number of buggy versions for our evaluation. For example, there are 38 single-bug versions (including versions that contain bugs that involve multiple lines) for Space, so we randomly generate 38 multiple-bug versions for Space, each of which contains five bugs. Thus, we have 153 multiple-bug versions that contain five bugs (minimum number of multiple bugs).

We evaluate the measures' effectiveness on versions containing five and two bugs respectively, as well as their overall effectiveness in multiple-bug versions. Tables 4.18, 4.19, and 4.20 show the overall means and standard deviations of the percentage of code that must be inspected by the measures to localize bugs in all multiple-bug versions, versions containing five bugs, and versions containing two bugs, respectively.

Generally, the effectiveness of all measures in localizing multiple bugs is not as good as their effectiveness in localizing a single bug. The overall ranges of means of percentage of code inspected of the measures to localize all multiple-bug versions is between 74% and 91%. The measures require an inspection on average between 43% and 87% of the program elements in localizing two bugs in the programs, and

Association Measures	Mean	StdDev	Association Measures	Mean	StdDev
Added Value (M_{15})	73.88%	23.36%	Klosgen (M_{18})	74.92%	22.39%
One-Way Support (M_{27})	73.98%	23.50%	Ochiai II (M_{40})	75.07%	22.61%
Collective Strength (M_{16})	73.99%	23.36%	Yule's Q (M_3)	75.34%	20.43%
Ochiai	74.03%	22.95%	Yule's Y (M_4)	75.36%	20.38%
ϕ-Coefficient (M_1)	74.1%	22.80%	Leverage (M_{22})	75.76%	21.42%
Tarantula	74.14%	23.28%	Odds Ratio (M_2)	76.58%	19.99%
Odd Multiplier (M_{33})	74.16%	23.30%	Laplace (M_{10})	76.61%	19.87%
Zhang (M_{35})	74.17%	23.28%	Gini Index (M_7)	76.97%	21.48%
Interest (M_{12})	74.17%	23.29%	Rogers (M_{39})	77.37%	20.33%
Sebag (M_{31})	74.17%	23.28%	Accuracy (M_{21})	77.40%	20.34%
Confidence (M_9)	74.20%	23.29%	Simple-Matching (M_{38})	77.40%	20.34%
Interestingness Weighting Dependency (M_{24})	74.20%	23.39%	Least Contradiction (M_{32})	77.41%	20.45%
Kappa (M_5)	74.34%	22.99%	Information Gain (M_{19})	77.58%	20.39%
Two-Way Support (M_{28})	74.36%	22.96%	Two-Way Support Variation (M_{29})	77.58%	20.39%
Jaccard (M_{17})	74.40%	21.71%	J-Measure (M_6)	77.60%	20.33%
Relative Risk (M_{23})	74.43%	22.41%	Normalized Mutual Information (M_{26})	77.79%	20.52%
Certainty Factor (M_{14})	74.46%	21.09%	Support (M_8)	77.94%	19.34%
Conviction (M_{11})	74.46%	21.11%	Coverage (M_{20})	80.72%	17.40%
Example Rate (M_{34})	74.78%	23.28%	Loevinger (M_{30})	86.24%	15.32%
Sorensen-Dice (M_{36})	74.79%	21.74%	Piatetsky-Shapiro's (M_{13})	87.34%	12.69%
Anderberg (M_{37})	74.81%	21.68%	GoodMan Kruskal (M_{25})	91.31%	9.25%

Table 4.18: Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of all multiple-bug versions

the measures require more inspections (i.e., the ranges of 76% to 92% in localizing five bugs in the programs. As it is expected, the percentage of code inspected is more likely to be larger when there are more bugs manifest in programs. Among all of the measures, the measures that could localize all multiple-bug versions, five bugs, and two bugs in the programs with the smallest percentage of code inspected are Added Value (M_{15}), Relative Risk (M_{23}), and Two-Way Support (M_{28}) respectively.

We notice that performance of the measures in localizing five and two bugs in the programs are different. Tarantula performs better than Ochiai when localizing two bugs (i.e., 43.05% for Tarantula, 57.75% for Ochiai). In contrast Ochiai performs slightly better than Tarantula when localizing five bugs in the programs (i.e., 78.14% for Tarantula, 76.14% for Ochiai).

There are 3 measures (i.e., Added Value (M_{15}), One-Way Support (M_{16}), and Collective Strength (M_{16})) that perform better than Tarantula and Ochiai in localizing all multiple-bug versions in the programs. Among these measures, Added Value (M_{15}) and Collective Strength (M_{16}) also have good performance in localizing single bugs. The overall mean of percentage of code inspected for Added Value (M_{15}) and Collective Strength (M_{16}) are 25.74% and 25.66% respectively, which are only

Association Measures	Mean	StdDev	Association Measures	Mean	StdDev
Relative Risk (M_{23})	75.56%	23.17%	Confidence (M_9)	78.21%	20.59%
Certainty Factor (M_{14})	76.03%	21.34%	Laplace (M_{10})	78.24%	19.53%
Conviction (M_{11})	76.04%	21.36%	Kappa (M_5)	78.35%	20.20%
Ochiai	76.16%	21.33%	Example Rate (M_{34})	78.41%	20.76%
Yule's Q (M_3)	76.71%	20.85%	Two-Way Support (M_{28})	78.46%	20.06%
Yule's Y (M_4)	76.75%	20.81%	Ochiai II (M_{40})	78.52%	20.85%
Jaccard (M_{17})	77.34%	20.00%	Leverage (M_{22})	79.36%	19.08%
Coefficient (M_1)	77.52%	20.67%	Support (M_8)	79.67%	18.81%
Added Value (M_{15})	77.74%	21.08%	J-Measure (M_6)	80.10%	18.98%
Anderberg (M_{37})	77.78%	19.96%	Information Gain (M_{19})	80.11%	19.04%
Sorensen (M_{36})	77.78%	19.97%	Two-Way Support Variation (M_{29})	80.11%	19.04%
Collective Strength (M_{16})	77.92%	20.33%	Normalized Mutual Information (M_{26})	80.22%	18.92%
One-Way Support (M_{27})	77.97%	20.91%	Gini Index (M_7)	80.27%	19.66%
Klosgen (M_{18})	78.10%	20.45%	Rogers (M_{39})	80.74%	18.43%
Odds Ratio (M_2)	78.12%	20.27%	Accuracy (M_{21})	80.78%	18.43%
Tarantula	78.14%	20.60%	Simple-Matching (M_{38})	80.78%	18.43%
Odd Multiplier (M_{33})	78.17%	20.61%	Least Contradiction (M_{32})	80.78%	18.57%
Zhang (M_{35})	78.18%	20.59%	Coverage (M_{20})	82.64%	16.34%
Interest (M_{12})	78.18%	20.59%	Loevinger (M_{30})	88.58%	13.68%
Interestingness Weighting Dependency (M_{24})	78.18%	20.73%	Piatetsky-Shapiro's (M_{13})	89.52%	11.38%
Sebag (M_{31})	78.18%	20.59%	GoodMan Kruskal (M_{25})	91.82%	9.61%

Table 4.19: Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of versions containing five bugs

slightly higher than the smallest mean of percentage of code inspected to localize all single-bug programs (25.24%). In addition, Added Value (M_{15}) and Collective Strength (M_{16}) are in the top of partial order of 5 and 4 bug categories out of 7 categories.

We also perform statistical tests for each pair of measures including Tarantula and Ochiai using Wilcoxon signed rank test [139] at 0.05 statistical significance threshold to see if some measures are statistically significantly better than others in localizing multiple-bug versions. Table 4.21 shows the measures that are at the top of the partial order (no other measures that statistically significantly perform better than the measure) for localizing both all multiple-bug versions, versions containing five bugs, and versions containing two measures. We notice that Added Value (M_{15}), Collective Strength (M_{16}), Ochiai, Ochiai II (M_{40}), One-Way Support (M_{27}), Relative Risk (M_{23}), Two-Way Support (M_{28}), and ϕ-Coefficient (M_1) are at the top of the partial order of all multiple-bug versions and at least at the top order of one of partial order of two or five bugs versions.

We also plot the curve showing the proportion of code that is investigated (x-axis) versus the proportion of bugs localized (y-axis) for all multiple-bug versions.

Association Measures	Mean	StdDev	Association Measures	Mean	StdDev
Two-Way Support (M_{28})	42.95%	19.45%	Gini Index (M_7)	51.65%	17.95%
Confidence (M_9)	43.50%	19.90%	Jaccard (M_{17})	51.90%	21.55%
Interest (M_{12})	43.50%	19.90%	Sorensen-Dice (M_{36})	51.90%	21.55%
One-Way Support (M_{27})	43.50%	19.90%	Anderberg (M_{37})	52.10%	21.24%
Sebag (M_{31})	43.50%	19.90%	Ochiai	57.75%	28.46%
Odd Multiplier (M_{33})	43.50%	19.90%	Information Gain (M_{19})	58.25%	20.44%
Zhang (M_{35})	43.50%	19.90%	Two-Way Support Variation (M_{29})	58.25%	20.44%
Tarantula	43.50%	19.90%	J-Measure (M_6)	58.40%	20.54%
Kappa (M_5)	43.60%	19.92%	Normalized Mutual Information (M_{26})	59.25%	23.20%
Interestingness Weighting Dependency (M_{24})	43.75%	20.21%	Conviction (M_{11})	62.40%	14.45%
Collective Strength (M_{16})	43.90%	23.58%	Certainty Factor (M_{14})	62.40%	14.45%
Added Value (M_{15})	44.35%	18.65%	Laplace (M_{10})	64.15%	18.33%
Example Rate (M_{34})	47.00%	23.27%	Support (M_8)	64.70%	18.65%
ϕ-Coefficient (M_1)	47.95%	21.76%	Odds Ratio (M_2)	64.80%	12.85%
Leverage (M_{22})	48.20%	18.38%	Yule's Q (M_3)	64.80%	12.85%
Ochiai II (M_{40})	48.65%	17.84%	Yule's Y (M_4)	64.80%	12.85%
Klosgen (M_{18})	50.60%	22.10%	Relative Risk (M_{23})	65.80%	12.75%
Accuracy (M_{21})	51.60%	15.29%	Coverage (M_{20})	66.00%	18.64%
Least Contradiction (M_{32})	51.60%	15.29%	Loevinger (M_{30})	68.35%	15.69%
Simple-Matching (M_{38})	51.60%	15.29%	Piatetsky-Shapiro's (M_{13})	70.70%	9.55%
Rogers (M_{39})	51.60%	15.29%	GoodMan Kruskal (M_{25})	87.40%	4.17%

Table 4.20: Overall mean and standard deviation (in parentheses) of accuracy values (smaller the better) of versions containing two bugs

Num. of Bugs in a Version	Top Measures
Two and five bugs	Ochiai, Added Value (M_{15}), Relative Risk (M_{23}), Ochiai II (M_{40}), Collective Strength (M_{16}), One-Way Support (M_{27}), Two-Way Support (M_{28}), and ϕ-Coefficient (M_1)
Five bugs	Ochiai, Relative Risk (M_{23}), and Ochiai II (M_{40})
Two bugs	Ochiai, Added Value (M_{15}), Collective Strength (M_{16}), One-Way Support (M_{27}), Two-Way Support (M_{28}), Klosgen (M_{18}), Confidence (M_9), Interest (M_{12}), Interestingness Weighting Dependency (M_{24}), Kappa (M_5), Odd Multiplier (M_{33}), Sebag (M_{31}), Zhang (M_{35}), and Tarantula

Table 4.21: Measures that are at the top of the partial orders for different types of multi-bug versions. The underline marks measures that are at the top of the partial orders of two types of multiple-bug versions.

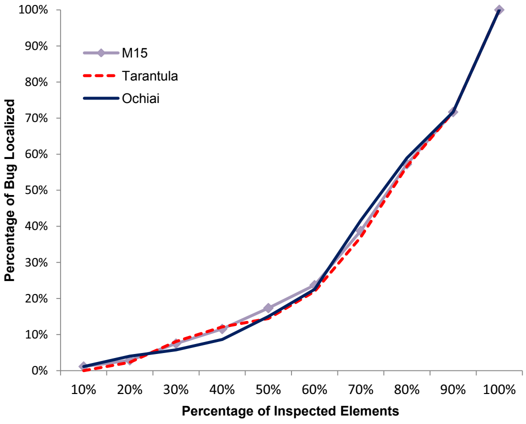


Figure 4.21: M_{15}, Ochiai, and Tarantula for all multiple-bug programs

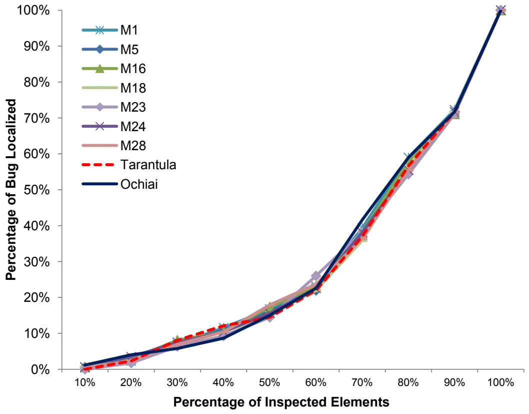


Figure 4.22: M_1, M_5, M_{16}, M_{18}, M_{23}, M_{24}, M_{28}, Ochiai, and Tarantula for all multiple-bug programs

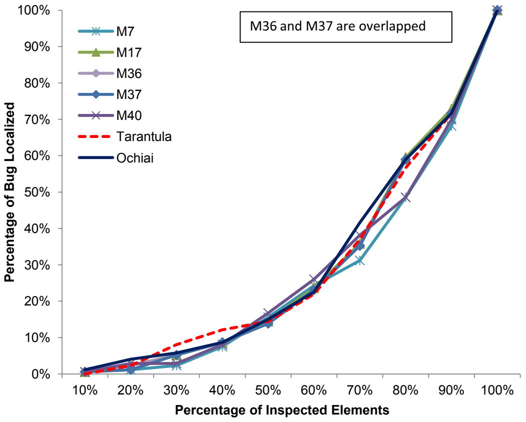


Figure 4.23: M_7, M_{17}, M_{36}, M_{37}, M_{40}, Ochiai, and Tarantula for all multiple-bug programs

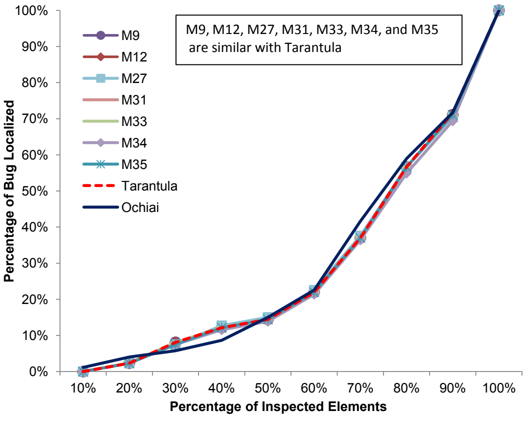


Figure 4.24: M_9, M_{12}, M_{27}, M_{31}, M_{33}, M_{34}, M_{35}, Ochiai, and Tarantula for all multiple-bug programs

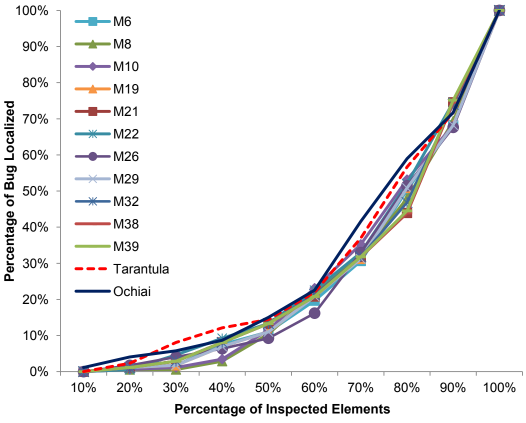


Figure 4.25: M_6, M_8, M_{10}, M_{19}, M_{21}, M_{22}, M_{26}, M_{29}, M_{32}, M_{38}, M_{39}, Ochiai, and Tarantula for all multi-bug programs

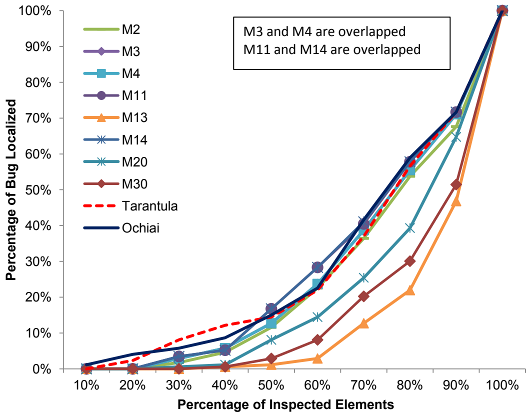


Figure 4.26: M_2, M_3, M_4, M_{11}, M_{13}, M_{14}, M_{20}, M_{30}, together with Ochiai and Tarantula for all multi-bug programs

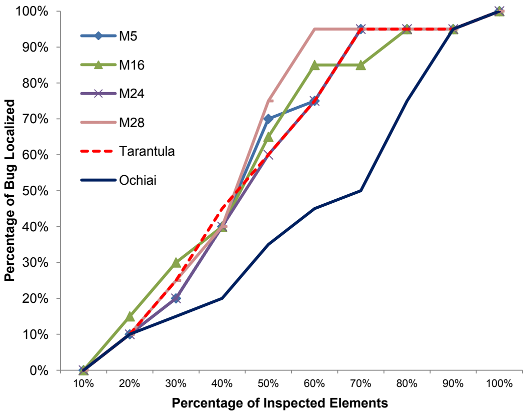


Figure 4.27: M_5, M_{16}, M_{24}, M_{28}, Ochiai, and Tarantula for versions containing two bugs

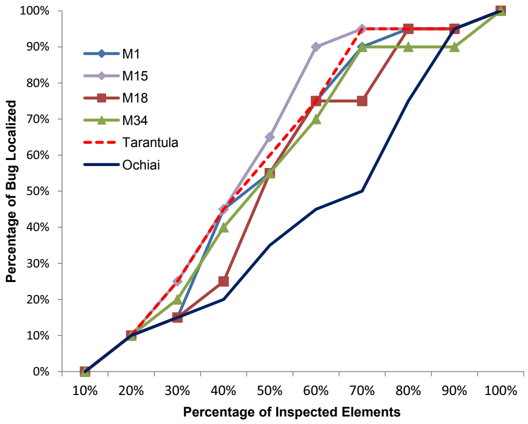
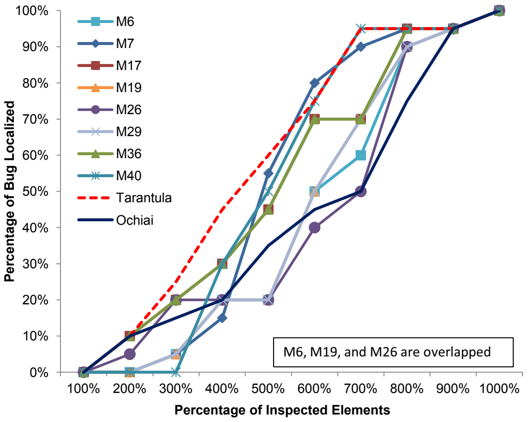
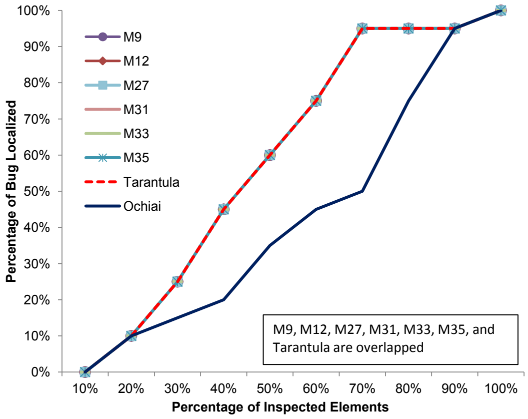


Figure 4.28: M_1, M_{15}, M_{18}, M_{34}, Ochiai, and Tarantula for two-bug versions



M6, M19, and M26 are overlapped

Figure 4.29: M_6, M_7, M_{17}, M_{19}, M_{26}, M_{29}, M_{36}, Ochiai, and Tarantula for two-bug versions



M9, M12, M27, M31, M33, M35, and Tarantula are overlapped

Figure 4.30: M_9, M_{12}, M_{27}, M_{31}, M_{33}, M_{35}, Ochiai, and Tarantula for versions containing two bugs

We split the large graphs into several smaller graphs by grouping measures that have similar accuracies, as shown in Figures 4.21, 4.24, 4.22, 4.23, 4.25, and 4.26. For each graph, we compare a number of association measures with Tarantula and Ochiai. Figure 4.21 shows a measure (i.e., Added Value (M_{15})) that could localize similar number of buggy versions as compared to Ochiai when less than 10% of the code is inspected. Added Value (M_{15}) as well as Ochiai could localize 1% of multiple-bug versions when less than 10% of the code is inspected. When more than 40% of the code is inspected, Added Value (M_{15}) could localize more buggy versions than Tarantula and Ochiai. Figures 4.22 and 4.23 show measures that could

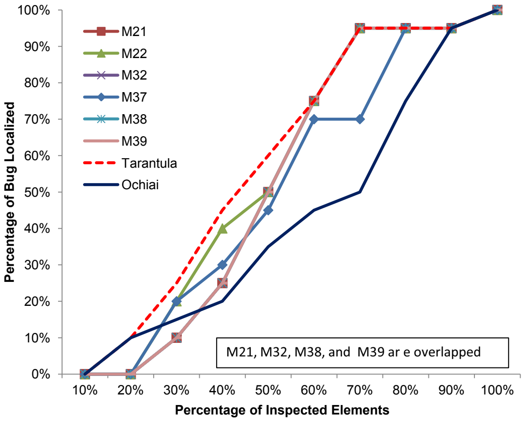


Figure 4.31: M_{21}, M_{22}, M_{32}, M_{37}, M_{38}, M_{39}, M_{40}, Ochiai, and Tarantula for two-bug versions

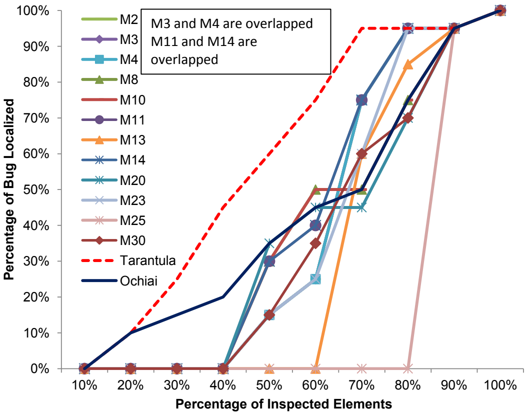


Figure 4.32: M_2–M_4, M_8, M_{10}, M_{11}, M_{13}, M_{14}, M_{20}, M_{23}, M_{25}, M_{30}, Ochiai and Tarantula for two-bug versions

localize similar numbers of buggy versions as compared to Ochiai when less than 10% of the code is inspected and they also have similar performance with Tarantula and Ochiai.

On the other hand, Tarantula could not localize any multi-bug version within 10% of code inspection. Measures that have similar performance with Tarantula are shown in Figure 4.24. Measures that perform worse than Tarantula and Ochiai are shown in Figures 4.25 and 4.26. In this work, we omit showing the curves for versions that contain five bugs because the curves are similar to the curves for all multiple-bug versions.

We also plot the curves showing the proportion of code that is investigated (x-axis) vs. the proportion of bugs localized (y-axis) for versions that contain two bugs, as shown in Figures 4.27, 4.30, 4.28, 4.29, 4.31, and 4.32. A number of measures could localize more bugs as compared to Tarantula and Ochiai when more than 10% of code is inspected, as shown in Figures 4.27, while several measures shown in Figure 4.30 have the same performance as Tarantula. Figure 4.28 shows measures that could localize more bugs as compared to Ochiai when more than 10% of code is inspected. The rest of the measures could not outperform Tarantula and Ochiai. See Figures 4.29, 4.31, and 4.32 for the details of these measures. We summarize the findings of this section in the answer for RQ6 in the next section.

4.5 Discussion

In this section, we summarize the answers to the research questions mentioned in Chapter 1.

RQ1. Are vanilla or off-the-shelf association measures accurate enough in localizing faults? We are interested to find if off-the-shelf association measures are powerful enough to locate bugs. Based on the mean accuracy values of the measures, the 40 association measures could localize all the bugs when an average of 25–58$\%$ of the program elements are inspected. Fifty percent of the association measures are able to help find bugs by inspecting an average of 25–27$\%$ of elements, while Tarantula and Ochiai require developers to inspect approximately 27$\%$ and 26$\%$ of program elements respectively.

RQ2. Among 40 association measures, which of the association measures are the most accurate for localizing faults? Next, we are interested to find which association measures are better than others in localizing single-bug programs. The answer to this research question is the partial order shown in Figure 4.2. Two off-the-shelf association measures are at the top of the partial order namely: Klosgen (M_{18}), and Normalized Mutual Information (M_{26}). They perform comparably to Ochiai. They are statistically significantly better than Tarantula and the other off-the-shelf association measures. We also highlight 7 other measures that perform statistically significantly better than Tarantula i.e., ϕ-coefficient (M_1), Added Value (M_{15}), Collective Strength (M_{16}), Two-Way Support (M_{28}), Interestingness Weighting Dependency (M_{24}), Example and Counterexample Rate (M_{34}), and Kappa (M_5).

RQ3. What is the relative performance of off-the-shelf association measures as compared to well-known suspiciousness measures (in particular, Tarantula and Ochiai) for fault localization? We also would like to know the relative accuracy of the association measures versus those of well-known suspiciousness measures for fault localization for single-bug versions. When only 10$\%$ of the code is inspected, Klosgen (M_{18}) and Added Value (M_{15}) could localize more bugs than Tarantula and

Ochiai. They could localize 47% and 48% of the bugs respectively, while Tarantula and Ochiai could localize 39% and 45% of the bugs respectively.

RQ4. Is the accuracy of off-the-shelf association measures, of Tarantula, and of Ochiai, in localizing faults for programs written in C, different from their accuracy in localizing faults for programs written in Java? We find that most measures perform better for the C programs than the Java programs. We compute two partial orders of the 40 association measures, Tarantula, and Ochiai, by performing statistical significance tests based on their percentages of code inspected in localizing bugs in C and Java programs. For the C programs, Ochiai and Klosgen (M_{18}) are the measures that are at the top of the partial orders (i.e., no measures are statistically significantly better than them). For the Java programs, a number of measures are at the top including Ochiai, Klosgen (M_{18}), Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), Zhang (M_{35}), Tarantula, Interestingness Weighting Dependency (M_{24}), and Normalized Mutual Information (M_{26}). When up to 10% of the code is inspected, Klosgen (M_{18}) and Added Value (M_{15}) could localize more bugs in C programs than Ochiai and Tarantula. Tarantula and Ochiai could localize 43% and 52% of the bugs respectively, while Klosgen (M_{18}) and Added Value (M_{15}) could localize 54% and 53% of the bugs. For the Java programs, Tarantula, Confidence (M_9), Interest (M_{12}), Added Value (M_{15}), Klosgen (M_{18}), Odd Multiplier (M_{33}), Interestingness Weighting Dependency (M_{24}), One-Way Support (M_{27}), Sebag (M_{31}), Example and Counterexample Rate (M_{34}), and Zhang (M_{35}) could localize the largest number of bugs as compared to Ochiai (i.e., 26% of the bugs could be localized within 10% of code inspected). Ochiai, on the other hand, could only localize 20% of the bugs.

RQ5. What is the effectiveness of off-the-shelf association measures, Tarantula, and Ochiai in localizing different types of bugs? In terms of the percentage of code inspected, we also compute several partial orders (one per bug category) by performing statistical significance tests. Tarantula is at the top of three partial orders. Ochiai, Information Gain (M_{19}), Normalized Mutual Information (M_{24}), and Two-Way Support Variation (M_{29}) are at the top of six partial orders. Klosgen (M_{18}) is at the top of seven partial orders. In terms of proportion of bugs localized, Klosgen (M_{18}) could also localize the most number of bugs for all categories as compared to other measures. The categories of bugs that could be better localized by most of the measures are addition or removal of non-conditional statements (CH-CS), change of loop predicates (LP-CC), and change of return expression (CH-RET).

RQ6. What is the accuracy of off-the-shelf association measures, Tarantula, and Ochiai in localizing bugs in multiple-bug programs? When localizing multiple bugs in programs, no measure performs better than in localizing a single bug in the programs. The percentages of code inspected to localize all bugs in all multi-bug versions, versions containing five bugs, and versions containing two bugs are 74% to 91%, 76% to 92%, and 43% to 84% respectively. Measures that are at the top of the partial order for localizing all multiple-bug versions are Added Value (M_{15}), Collective Strength (M_{16}), Ochiai, Ochiai II (M_{40}), One-Way Support (M_{27}), Relative Risk (M_{23}), Two-Way Support (M_{28}), and ϕ-Coefficient (M_1). We notice that Added Value (M_{15}) and Collective Strength (M_{16}) also have good accuracies in localizing single-bug programs. The overall means of the percentage of code inspected for Added Value (M_{15}) and Collective Strength (M_{16}) are 25.74% and 25.66%, respectively, which are only slightly larger than the smallest mean of percentage of code inspected to localize all single-bug programs (25.24%). Also, Added Value (M_{15}) and Collective Strength (M_{16}) are at the top of partial order of 5 and 4 bug categories out of 7 categories.

Based on our results, we find that there is no single best measure for all cases. For localizing C and Java programs containing a single bug, Klosgen (M_{18}) could localize the bug with the smallest average of percentage of code inspected. Also, when we evaluate the effectiveness of the measures to localize different bug categories, Klosgen (M_{18}) could outperform other measures for all bug categories. While in localizing multiple bugs, Added Value (M_{15}) could localize bugs with the smallest average of percentage of code inspected. We notice that Added Value also has good accuracy in localizing single bugs; its accuracy is only slightly lower than that of Ochiai and Klosgen.

We also find that the effectiveness of association measures in localizing faults can be different for different buggy programs. The formula of association measures and the program spectra of buggy programs are two important factors that greatly affect the effectiveness of the measures in localizing faults. The score assigned by an association measure to each program element depends on the values of its dichotomy matrix (i.e., a matrix that associates the executions of a program element with the occurrence of a program failure) which is derived from the program spectra. When some program elements have the same dichotomy matrix, the scores assigned to the elements are the same. Hence it is hard to differentiate which element is more likely to be the faulty element. Lack of variety in scores assigned to program elements may not benefit the association measures in localizing faults.

Based on our results, association measures as well as the two well-known measures require larger number of program elements to be inspected to localize bugs in XML security programs. We observe that many program elements in these buggy programs obtain the same scores. The values of their dichotomy matrices indicate that many program elements are either being executed or not executed at all. Therefore, having test cases that produce program spectra that can provide information about the number of times a program element is executed and not executed by failed test cases, as well as executed and not executed by correct test cases would be helpful to differentiate the likelihood of each program element to be the faulty element (as more unique score assigned to each program element).

Nevertheless, it remains challenging to determine under which case each measure can better localize the bugs. The formula of an association measures only indicates the association strength between the executions of a program element with the occurrence of a failure. However, the effectiveness of an association measure in localizing faults is affected by the scores of program elements collectively, that determines the relative ordering of the faulty elements. This chapter emphasizes on investigating whether or not there is a benefit of employing association measures for fault localization. Further study is required to investigate the characteristic of cases under which an association measure can successfully localize faults with minimum percentage of code inspected. Both formula of association measures and characteristic of program spectra or bugs are important factors to be studied.

In addition, localizing bugs in multiple-bug programs requires much more code to be inspected than localizing bugs in single-bug programs. Future research is required to improve the effectiveness of the technique in localizing bugs in multi-bug programs. Also, it can be interesting future work to explore ways to compose various measures together so that the combined *meta-measure* may perform better for all cases than individual measures.

4.6 Threats to Validity

Threats to construct validity refers to the suitability of our evaluation criteria. In this work, we use two criteria: percentage of program elements inspected to find all bugs, and proportion of bugs localized when at most a given percentage of program elements are inspected. We believe these two criteria reasonably measure the effectiveness of a fault localization approach. They have also been used before in prior studies on fault localization [6, 65].

Threats to internal validity include bias and human errors. The accuracy of a measure in localizing bugs is influenced by the granularity level considered during program instrumentation and trace generation (statement, basic block, or method levels). Different granularity levels may produce different accuracies since there would be a different total numbers of elements which would affect the percentages of inspected elements. We choose to use block-hit spectra in our evaluation since it has a suitable balance between instrumentation cost and bug-revealing power, and our study focuses on comparing the effectiveness of different association measures on the *same* spectra. We hypothesize that the relative performance of different association measures on other spectra may remain the same as that on block-hit spectra, but it remains an interesting future work for us to verify. Also, we manually instrument the C programs; we might miss instrumenting some blocks or add extraneous instrumentation code. For Java programs, we automatically instrument the programs ; there could be some errors in our implementation. We manually assign the bugs into categories; there might be some errors in our assignments. We carefully checked the instrumented programs and assigned bug category labels to minimize such errors.

Threats to external validity refers to the generalizability of our findings. We have tried to reduce this threat by considering a number of programs of various sizes written in two popular programming languages: C and Java.

4.7 Conclusion

In this work, we investigate the effectiveness of a comprehensive number of association measures for fault localization. These measures gauge the strength of association between two variables expressible as a dichotomy matrix. We consider and compare 40 association measures with two well-known fault localization measures, namely Tarantula and Ochiai.

For programs that contain a single bug, we find that Klosgen outperforms Ochiai and Tarantula in terms of the average percentage of code that must be inspected. Many measures, including Normalized Mutual Information, J-Measure, Information Gain, Two-Way Support Variation, Example and Counterexample Rate, Confidence, Kappa, Sebag, Odd Multiplier, Interest, Zhang, and One-Way Support outperform Tarantula. The percentages of code that must be inspected for different buggy program versions are different; such percentage values for each measure form a distribution, and we employ statistical significance tests to compare the accuracy of different measures. We find that three measures, Klosgen, Normalized Mutual Information, and Ochiai can be statistically significantly better than other measures when localizing single-bug program. In terms of proportions of bugs found when up to 10% of code is inspected, Klosgen outperforms Ochiai. Also, Klosgen, Ochiai, ϕ-coefficient, Added Value, Collective Strength, Two-Way Support, Interestingness Weighting Dependency, Example and Counterexample Rate, and Kappa outperform Tarantula. Thus, we can conclude that association measures are also promising to be used for fault localization.

We find that most measures perform better for the C programs than the Java programs. For the C programs, in terms of proportions of bugs localized, Added Value and Klosgen outperform Ochiai and Tarantula. For the Java programs, in terms of proportions of bugs localized, Tarantula, Confidence, Interest, Added Value, Klosgen, Relative Risk, Loevinger, Normalized Mutual Information, Odd Multiplier, Example and Counterexample Rate, and Least Contradiction measures, outperform the other measures and Ochiai.

We have also grouped the bugs into 7 categories and analyze the effectiveness of the measures to localize each category of bugs. The categories of bugs that could be better localized by most of the measures are addition or removal of non-conditional statements (CH-CS), change of loop predicates (LP-CC), and change of return expression (CH-RET). Klosgen is among the best measures for all bug categories.

The effectiveness of all measures in localizing multiple bugs in programs is not as good as localizing buggy programs that contain a single bug. Added Value could localize the bugs with the smallest percentage of inspected code. There are also other measures that have comparable performance as Added Value, i.e., Ochiai, Relative Risk, Ochiai II, Collective Strength, One-Way Support, Two-Way Support, and ϕ-Coefficient.

Based on our results, we find that there is no single best measure for all cases.

While there are several measures that perform statistically comparable for localizing single-bug programs, Klosgen could localize bugs with the smallest on average percentage of code inspected. As for localizing multi-bug programs, Added Value could localize bugs with the smallest on average percentage of code inspected, but a number of measures have comparable performance as well. Thus, solely based on fault localization accuracy, Klosgen and Added Value can be best measures to use.

Dataset. Our dataset and tool are made publicly available at: `http://www.mysmu.edu/phdis2009/lucia.2009/jsme/Dataset.htm`.

Chapter 5

Fusing Fault Localizers

As we have demonstrated in Chapter 4, for various bugs, the best techniques to localize the bugs may differ due to the characteristics of the buggy programs and their program spectra. In this chapter, we leverage the diversity of existing spectrum-based fault localization techniques to better localize bugs using data fusion methods. Our proposed approach requires no training data and is able to adapt itself for various kinds of spectra and bugs.

We organize this chapter as follows. Section 5.1 presents the motivation and main contribution of this chapter. Section 5.2 provides a motivating example to illustrate the benefit of our data fusion approach and Section 5.3 elaborates our approach. We present our experiment results that demonstrate the effectiveness of our approach in Section 5.4 and finally conclude our work in Section 5.5.

5.1 Variations in Effectiveness of Localizing Faults

Many fault localization techniques have been proposed to locate the root cause of a program failure by analyzing the program traces (i.e., the abstraction of program behaviors). Spectrum-based fault localization techniques [3,27,31,65,84,151] compare the spectra of correct and failed executions to identify program elements (i.e., statements, blocks, methods, and components) that are likely to be the root cause of a program failure. These techniques often use statistical analysis to assign a suspiciousness score to each program element based on its likelihood to be faulty. The higher the score, the more suspicious an element is. A list of the most suspicious program elements is then presented to developers for inspection. As discussed in Chapter 4, the best performing spectrum-based fault localization techniques vary for different buggy programs. Some techniques can rank faulty elements at the top positions for some buggy programs, while for other buggy programs, they rank faulty elements low in the list.

In this chapter, we aim to better localize the bugs by leveraging diversity of existing spectrum-based fault localization techniques (in particular 40 association measures used in Chapter 4, Tarantula [65], and Ochiai [3]). Since these techniques are lightweight, we could inexpensively obtain suspiciousness scores for program elements by using different techniques. Different from the approach proposed by Santelices et al. [122] that analyze several types of program spectra using a single technique, we combine many techniques that analyze a single type of program spectra (i.e., block hit spectra).

Data fusion methods have been proposed in the domain of information retrieval to rank the most relevant documents such that the relevant ones are in the top positions by combining the ranking information from different retrieval systems [142]. We incorporate data fusion methods with the goal of ranking the faulty program elements higher in the list by combining the scores or ranks assigned to program elements by different fault localization techniques. Our approach, referred to as `Fusion Localizer`, normalizes the suspiciousness scores of different fault localization techniques, selects fault localization techniques to be fused, and combines the selected techniques using a data fusion method. We propose 20 variants of `Fusion Localizer` that use different score normalization, technique selection, and data fusion methods.

Related to our work, Wang et al. [137] propose an approach that linearly combines the scores of a number of association measures to rank faulty program elements. Their approach generates a weight for each association measure using genetic algorithm based on a set of training data i.e., program traces of the past program failures and the corresponding faults of the past failures. However, such training data could be unavailable *e.g.*, if developers did not archive the program traces and the corresponding faults. Archiving program traces of past failures can consume a lot of storage. Even when traces are not archived but test cases to generate the traces are available, regenerating such program traces using the test cases may take time especially for large programs. Furthermore, the bugs and failures in the training set may not be representative enough to generalize to other buggy programs, which could limit the effectiveness of the approach. Different from this technique, our approach can compute the weight for each fault localization technique and select the techniques to be fused without requiring any training data. Furthermore, our approach is bug specific – it adaptively chooses a different set of techniques to be fused for different buggy programs based on their program spectra.

We have evaluated our approach on a commonly used benchmark dataset consisting of 10 small to medium sized programs written in C and Java, and our own dataset consisting of 30 real bugs collected from 3 larger programs – namely Rhino, Lucene, and Ant. We compare our approach against a number of state-of-the-art spectrum-based fault localization techniques that also do not require training data and analyze a single type of program spectra, i.e., Ochiai [3], theoretically best spectrum-based fault localization techniques [143], and theoretically best genetic programming (GP) based fault localization techniques [144]. Our experiment results show that our approach outperforms these techniques. The best performing variants of *Fusion Localizer* (i.e., $\mathcal{F}^{Zero-One,Bias}_{CombANZ}$ and $\mathcal{F}^{Zero-One,Overlap}_{CombANZ}$) statistically significantly outperform the state-of-the-art spectrum-based fault localization techniques. These best variants require statistically significantly smaller average percentage of code inspected to locate faulty elements as compared to the best performing state-of-the-art fault localization techniques (i.e., 21% vs. 24%). When developers only inspect 10% of the most suspicious program elements, these best variants could improve the percentage of bugs localized by the best performing state-of-the-art fault localization techniques by 11% to 26%. Furthermore when developers only inspect the top 10 most suspicious program blocks, these best variants could improve the number of bugs localized by the best performing state-of-the-art fault localization techniques by 23% to 26%.

The main contributions of this chapter are as follow:

1. We leverage diversity of 40 association measures evaluated in Chapter 4, Tarantula [65], and Ochiai [3] to better localize bugs using data fusion methods.
2. We provide a bug specific approach that adaptively selects a set of techniques to be fused for each buggy program.
3. We propose an approach that does not require any training data (e.g., program traces of the past program failures) to select which techniques to be fused to better localize bugs in programs.
4. We show that combining the lists of most suspicious program elements recommended by different fault localization techniques (i.e., association measures, Tarantula, and Ochiai) using data fusion methods can improve the effectiveness in localizing faults and significantly outperform the state-of-the-art spectrum-based fault localization techniques.

5.2 Motivating Example

This section provides an illustration of the benefit of the use of data fusion to leverage different techniques to help ranking faulty program elements such that the faulty ones are in the top positions. Figure 5.1 contains a sample program code, excerpted from one of our subject programs, i.e., the method $gser$ of the program tot_info, version 8. In this example, we divide the code into five blocks. The bug is in Block 5 where the value assigned to a variable $temp$ is incorrectly calculated.

Various spectrum-based fault localization techniques, such as Ochiai [3], Klosgen [75], and Pietatsky Shapiro [108], can be used to assign suspiciousness scores to each program block. Based on these scores, developers could inspect the program blocks starting from the most suspicious blocks. Figure 5.1 shows the suspiciousness score of each program blocks as output by Ochiai, Klosgen, and Pietatsky Shapiro. These scores are computed by analyzing the program traces collected during the executions of the test cases that come with the program (i.e, tot_info).[1]

According to Ochiai and Klosgen, the most suspicious program blocks are Blocks 3 to 5, followed by Block 1, then Block 2. The ranks of Blocks 1 and 2 would be 4 and 5. According to Ochiai and Klosgen, Blocks 3 to 5 obtain the same score. Among the three blocks, we do not know which block developers would inspect first, thus we assign the worst case rank to these blocks. The worst case rank that either Blocks 3 to 5 is inspected by developers would be 3. According to Pietatsky Shapiro, Blocks 2 and 5 are the most suspicious program blocks, followed by Blocks 3, 4, and 1. Since Blocks 2 and 5 obtain the same score, the worst case rank that either Blocks 2 or 5 is inspected by developers using this measure would be 2. In this example, locating the faulty block based on the recommendation by Pietatsky Shapiro requires examining fewer blocks than when using the recommendations of Ochiai and Klosgen.

A simple data fusion method to combine a set of techniques is CombSUM [41, 42]. CombSUM can be used to combine the scores output by Ochiai, Klosgen, and Pietatsky Shapiro to produce a new score for each program block. For each program block, the new score is calculated by summing up the normalized scores given by Ochiai, Klosgen, and Pietatsky Shapiro. As the ranges of the scores produced by these techniques may differ one another, we may need to normalize the scores to make them comparable. In this example, we normalize the scores using zero-one score normalization.[2] From Figure 5.1, the CombSUM score for Block 5 is 1 + 1 + 1 = 3. We also compute the CombSUM score for the other blocks similarly. Finally,

[1] Please refer to [3] and Tables 4.5 for the formulas used to compute the suspiciousness scores.

[2] We elaborate the details of the normalization process in Section 5.3.

we find that Block 5 is the most suspicious block as it has the highest score among all of the blocks. Using the list of most suspicious blocks recommended by CombSUM, developers could locate the faulty block by only inspecting the first block, which is more effective than using the list of most suspicious blocks recommended by either Ochiai, Klosgen, or Pietatsky Shapiro. In this example, we show that a data fusion method can be used to boost the effectiveness of fault localization techniques.

5.3 Fusion Localizer

In this section, we present our approach that incorporates data fusion methods to locate the source of a program failure. The overview of this approach is presented in Section 5.3.1. We elaborate three main steps of our approach namely score normalization, technique selection, and data fusion in Sections 5.3.2, 5.3.3, and 5.3.4 respectively.

5.3.1 Overview

Our approach combines the scores produced by different spectrum-based fault localization techniques to produce a new ranking with the goal of improving fault localization effectiveness. In this work, we employ spectrum-based fault localization techniques because they are lightweight and have good accuracy. Nevertheless, it is also possible to use other fault localization techniques that can assign scores to program elements. The overall framework of our approach named *Fusion Localizer* is shown in Figure 5.1.

The input to our approach is a set of spectrum-based fault localization techniques which computes the suspiciousness scores of all program elements in a buggy program. In this work, we use two well-known spectrum-based fault localization techniques: Tarantula [65,66] and Ochiai [4,6], as well as the 40 association measures that have been studied for fault localization as in Chapter 4.

Block ID	Program Elements	Suspicious Scores			Normalized Scores			CombSUM
		Ochiai	Klosgen	Pietatsky Shapiro	Ochiai	Klosgen	Pietatsky Shapiro	

1	double a, x;
	double ap, del, sum;
	int n;
	double temp;
	if (x <= 0.0);
		0.82	0.31	-0.04	0.79	0.9	0.75	2.44

2	{return 0.0}	0.39	0.06	0	0	0	1	1

3	del = sum = 1.0 / (ap = a);
	for (n = 1; n <= ITMAX; ++n) {
		0.93	0.34	-0.15	1	1	0	2

4	sum += del *= x / ++ap;
	if (Abs(del) <Abs(sum) * EPS) {
		0.93	0.34	-0.15	1	1	0	2

5	/*BUGS: supposed to be:*/
	/*temp=sum*exp(-x+a*log(x)-LGamma(a))*/
	temp=sum*exp(x+a*log(x)-LGamma(a));
	return temp; }}
		0.93	0.34	0	1	1	1	3

Table 5.1: An example of using a data fusion method.

Different fault localization techniques use different formulas to calculate suspiciousness scores. Each formula has its own characteristics, especially in terms of the range and distribution of the suspiciousness scores computed using it. Thus, normalizing the scores is essential so that scores produced by different measures can be compared with one another. For every technique, we normalize the suspiciousness scores that are assigned to program elements into a new set of scores that falls in the range of between zero and one. Section 5.3.2 elaborates on the methods that we use in this work to normalize the scores.

After the scores are normalized, our approach then adaptively selects techniques to be fused together based on the spectra of the buggy program. Our goal is to select a set of techniques that complement one another well for a particular buggy program. Some techniques could perform worse when they are grouped together to localize a particular bug. As we demonstrate in Section 5.4, incorporating all 42 techniques does not result in the best performance. Section 5.3.3 elaborates on the methods that we use in this work to select the techniques.

Given a set of normalized scores from the selected techniques, we combine the scores using a number of existing data fusion methods. A new set of scores would then be assigned to program elements for the given buggy program. These new scores can then be used to create a new list for developer's inspection. Section 5.3.4 elaborates on the methods that we use in this work to fuse the normalized scores of selected fault localization techniques.

5.3.2 Step 1: Score Normalization

In this work, we apply two score normalization methods: Zero-One score normalization [142] and Reciprocal ranking normalization [85, 142]. Each of them can be used as the first step of *Fusion Localizer*. The following paragraphs discuss how these normalization methods work.

1) Zero-One Score Normalization: This method transforms scores from different

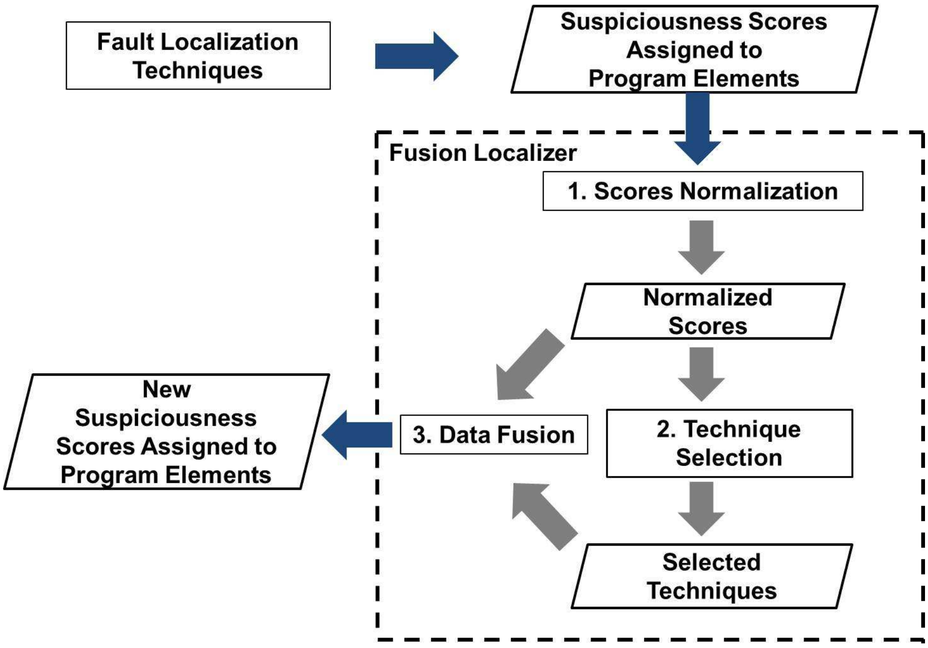

Fault Localization Techniques
Suspiciousness Scores Assigned to Program Elements
Fusion Localizer
1. Scores Normalization
Normalized Scores
2. Technique Selection
Selected Techniques
3. Data Fusion
New Suspiciousness Scores Assigned to Program Elements

Figure 5.1: Overview of Fusion Localizer

fault localization techniques into the same range, i.e., zero to one. The method works as follows.

Let n be the total number of techniques and m be the total number of program elements for a given buggy program, then $s_i(e_j)$ denotes the score of the j-th program element, assigned by the i-th technique, where $1 \leq j \leq m$ and 1$\leq i \leq n$. Furthermore, let max_s_i denote the maximum score produced by the i-th technique, and min_s_i denote the minimum score produced by the i-th technique. The normalized score of the j-th program element given by the i-th technique, is calculated as follows:

$$s\_norm_i(e_j) = \frac{s_i(e_j) - min\_s_i}{max\_s_i - min\_s_i}$$

2) Reciprocal Rank Normalization: Instead of directly normalizing a list of scores produced by different techniques, this method considers the ranks of program elements and transforms them into normalized scores. The method works as follows.

Let n be the total number of techniques and m be the total number of program elements in a buggy program. Also, let $r_i(e_j)$ denote the rank of the j-th program element, assigned by the i-th technique, where $1 \leq j \leq m$ and 1$\leq i \leq n.$ The normalized score of the j-th program element, ranked by the i-th technique is calculated as follows:

$$s\_norm_i(e_j) = \frac{1}{r_i(e_j)}$$

5.3.3 Step 2: Technique Selection

We adapt overlap-based selection [142] and bias-based selection [113] to select a subset of techniques to be fused together from a set of fault localization techniques.These methods are instance (or bug) specific since the selected techniques could be different for different programs. Also, they require no training data to select techniques. Each of them can be used as the second step of *Fusion Localizer*. We elaborate how we adapt these methods for fault localization as follows.

1) Overlap-Based Selection: This method selects techniques to be fused together based on the overlap between the list of top-K most suspicious program elements produced by a fault localization technique and the top-K lists produced by other techniques. By default, we set K to be 10% of the total number of program elements that the input buggy program has. If this number is less than 10, we set K to 10.

This method then measures the *overlap rate* of the results returned by each technique with the results of other techniques and favors the techniques for which the results overlap less than other techniques. Hence more possible failure-relevant program elements could be covered. The *overlap rate* of each technique with other techniques is computed as follows:

Definition 5.3.1 (Overlap Rate) *Let L_{all} be a set of program elements that appear in at least one of the top-K lists produced by the set of fault localization techniques. Also, let L_i be a set of program elements that appear in the top-K list of the i-th*

Technique	**Top 5 Most Suspicious Blocks**
T_1	Block 2, Block 3, Block 4, Block 7, Block 8
T_2	Block 4, Block 5, Block 6, Block 7, Block 9
T_3	Block 1, Block 4, Block 5, Block 6, Block 8
T_4	Block 4, Block 5, Block 6, Block 8, Block 10

Table 5.2: Example: Overlap-based selection

technique but not in the top-K lists of other techniques. The overlap rate of the i-th technique with the other techniques is calculated as follows:

$$o\_rate_i = \frac{|L_{all}| - |L_i|}{|L_{all}|}$$

Example: Given a buggy program, consider four techniques (i.e., T_1, T_2, T_3, and T_4) that return the top 5 most suspicious program elements (i.e., program blocks) as shown in Table 5.2.

Based on Table 5.2, the set of program blocks returned by at least one of the four techniques, denoted by L_{all}, is $\{$Block 1, Block 2, Block 3, Block 4, Block 5, Block 6, Block 7, Block 8, Block 9, Block 10$\}$. T_1 recommends two program blocks that are not recommended by the other techniques, i.e., L_1=$\{$Block 2, Block 3$\}$. T_2, T_3, and T_4 each recommends one block that is not recommended by other measures – L_2=$\{$Block 9$\}$, L_3=$\{$Block 1$\}$, and L_4=$\{$Block 10$\}$. Hence, the overlap rate of T_1 among the other techniques can be calculated as $\frac{(10-2)}{10} = 0.8$. The overlap rate of the other 3 techniques can be similarly computed – they are $\frac{(10-1)}{10} = 0.9$.

After calculating the overlap rate of each technique, we sort the techniques in ascending order of their overlap rates and select the top-N techniques. In this way, we aim to include techniques that have more unique results (i.e., top-K lists). By default, we set N to be 50% of all input techniques.

2) Bias-Based Selection: This method selects a subset of techniques to be fused based on the bias rate of each technique towards the *norm* considering the top-K lists returned by each technique. By default, we set K to be 10% of the total number of program elements that the input buggy program has. If this number is less than

10, we set K to 10.

This method represents each technique as a vector of zeroes and ones representing whether each of the program elements occurs in its top-K list. It also constructs the *norm* which is a vector containing the number of top-K lists each program element belongs to. This method then computes a bias rate for each technique based on a similarity coefficient (i.e., cosine similarity) and favors the techniques that are more biased towards the norm. The bias rate for each technique is computed as follows:

Definition 5.3.2 (Bias Rate) *Let n be the number of input techniques and m be the number of program elements that appear in a top-K list produced by at least one of the techniques. Let L_i be a vector of zeros and ones that represents whether each program element appears in the top-K list of the i-th technique, where $1 \leq i \leq n$. The norm L_{all} is a vector containing the number of top-K lists each program element appears in. Let $Sim(L_i, L_{all})$ be the similarity between the vector L_i and the vector L_{all}, computed based on cosine similarity [146]. Given $Sim(L_i, L_{all})$, the bias rate of the i-th technique is calculated as follows:*

$$Bias(L_i, L_{all}) = 1 - Sim(L_i, L_{all})$$

$$Sim(L_i, L_{all}) = \frac{\sum_{j=1}^{m} L_j \times L_{all_j}}{\sqrt{\sum_{j=1}^{m} L_j^2} \times \sqrt{\sum_{j=1}^{m} L_{all_j}^2}}$$

Example: Based on Table 5.2, the set of program blocks that appear in at least one top-5 lists is {Block 1, Block 2, Block 3, Block 4, Block 5, Block 6, Block 7, Block 8, Block 9, Block 10}. Block 4 is recommended by four techniques, while Blocks 5, 6, and 8 are recommended by three techniques. Block 7 is recommended by two techniques, and the rest of the blocks are recommended by only one technique. From these pieces of information, the norm L_{all} is $\{1, 1, 1, 4, 3, 3, 2, 3, 1, 1\}$. To calculate the bias rate of T_1 to the norm, i.e., Bias(L_1, L_{all}), we first calculate the similarity score of this technique with the norm L_{all}. The similarity score of L_1 with the norm L_{all} is 0.6822 which is calculated as follows:

$$Sim(L_1, L_{all}) = \frac{(1 + 1 + 4 + 2 + 3)}{\sqrt{5 \times 52}}$$

Based on the similarity score, the bias rate of T_1 is $1 - 0.6822 = 0.37$. The bias rate of the other techniques can be computed in a similar way. The bias rate of T_2 is $1 - \frac{13}{\sqrt{5 \times 52}} = 0.19$. T_3 has the same bias rate as T_4, i.e., $1 - \frac{14}{\sqrt{5 \times 52}} = 0.13$. For each technique, we calculate the bias rate of the results and sort them in decreasing order of their bias rates. We then select the top-N techniques, with the aim of including techniques that are less similar towards the norm. By default, we set N to be 50% of the input fault localization techniques.

5.3.4 Step 3: Data Fusion

Our approach adapts an unsupervised data fusion method proposed in the information retrieval domain to combine normalized scores from the selected fault localization techniques. In this work, we adapt five well-known unsupervised data fusion methods, namely CombSUM [41, 42], CombMNZ [41, 42], CombANZ [41, 42], correlation-based fusion methods [142], and Borda count [14]. Each of them can be used as the third step of *Fusion Localizer*.

The following paragraphs elaborate how the five popular fusion methods can be adapted for fault localization. We use the example shown in Figure 5.1 to illustrate how each method works.

1) CombSUM: This method combines the scores of different techniques, by simply summing up their scores. This method assumes that each technique is equally important.

Example. Based on the example in Figure 5.1, the set of new scores of Blocks 1 to 5 would be {2.44, 1, 2, 2, and 3}.

2) CombANZ: This method combines the scores from different techniques, by computing the average of the non-zero scores. Let the j-th program element be denoted as e_j, the i-th technique be denoted as T_i, and the score assigned to program element e_j by technique T_i be denoted as $T_i(e_j)$. Suppose there are n techniques and m_{e_j} denotes the number of techniques that assign a non-zero score to e_j, CombANZ calculates the new score for e_j as follows:

$$Score(e_j) = 1/m_{e_j} \times \sum_{i=1}^{n} T_i(e_j)$$

Example. Based on the example in Figure 5.1, the set of new scores of Blocks 1 to 5 are $\{\frac{2.44}{3}, \frac{1}{1}, \frac{2}{2}, \frac{2}{2}, \frac{3}{3}\}$={0.81, 1, 1, 1, 1}.

3) CombMNZ: CombMNZ is a variant of CombANZ; it multiplies the sum of all the scores for a given element with the number of techniques that assign a non-zero score to the element. Let the j-th program element be denoted as e_j, the i-th technique be denoted as T_i, and the score assigned to program element e_j by technique T_i be denoted as $T_i(e_j)$. Suppose there are n techniques and m_{e_j} denotes the number of systems that give a non-zero score to e_j, CombMNZ calculates the new score for e_j as follows:

$$Score(e_j) = m_{e_j} \times \sum_{i=1}^{n} T_i(e_j)$$

Example. Based on the example in Figure 5.1, the set of new scores of Blocks 1 to 5 is $\{2.44 \times 3, 1 \times 1, 2 \times 2, 2 \times 2, 3 \times 3\}$ = {0.732, 1, 4, 4, 9}.

4) Correlation-based methods: Different from the above methods, correlation-based methods assume each technique is not equally important. The importance of a technique is represented by its weight. Based on the weights of the techniques, these methods linearly combine (i.e., sum up) the scores assigned by different techniques multiplied by their weights. Let the j-th program element be denoted as e_j, and the score assigned to program element e_j by the i-th technique be denoted as $T_i(e_j)$. Then, let w_i denote the weight assigned to the i-th technique and n be the number of techniques. The new score of program element e_j is calculated as follows:

$$Score(e_j) = \sum_{i=1}^{n} w_i \times T_i(e_j) \tag{5.1}$$

The following paragraphs describe the weight calculation procedures of two correlation-based methods: CorrA and CorrB.

4.1) CorrA: This method computes the correlation among the techniques based on the overlap of the lists of top-N most suspicious program elements returned by the techniques. The method aims to minimize the domination of a certain group of techniques that tend to return similar results, by assigning a heavier weight to a technique that has less correlation with other techniques.

Let the sets of top-N most suspicious program elements returned by the i-th and j-th techniques be denoted as L_i and L_j respectively, where $i \neq j$. The *overlap ratio* between L_i and L_j is calculated as follows:

$$OverlapRatio_{ij} = 2 \times \frac{|L_i \bigcap L_j|}{2 \times N}$$

Suppose, there are n techniques, we calculate the weight of the i-th technique, based on the average of its overlap ratio with other techniques to be fused together as follows:

$$w_i = 1 - \frac{1}{n-1} \sum_{j=1,2,..,j \neq i}^{n} OverlapRatio_{ij}$$

Example. Based on the example in Table 5.1 and the top 3 most suspicious program elements returned by each technique, the overlap ratio between Ochiai and Klosgen is 1, while the ratio between Ochiai and Piatetsky Shapiro is 0.33. Thus, the weight assigned to Ochiai is 1-$\frac{1.33}{2}$ =0.335. In the same way, the weights of Klosgen and Piatetsky Shapiro are 0.335 and 0.67 , respectively. The set of new scores for these blocks can be calculated using these weights and Equation 5.1 – it is $\{1.07, 0.67, 0.67, 0.67, 1.34\}$.

In this work, we denote the variant of the CorrA that calculates the correlation among the measures based on the top 10% of the most suspicious program elements by $CorrA\_Top10\%$ and another variant that is based on the top 50% of the most suspicious program elements as $CorrA\_Top50\%$.

4.2) CorrB: CorrB is a correlation-based method that also computes the weight of a technique based on the overlap of its list of top-N most suspicious program elements with other lists produced by other techniques. Let the list of top-N most suspicious program elements returned by the i-th technique be denoted as L_i. For a program element e, let us denote the number top-N lists it belongs to as $list(e)$. Let L_{all} denote a set of program elements which appear in at least one of the lists produced by the techniques. Suppose there are n techniques to be fused, the weight of the i-th technique can be calculated as follows:

$$w_i = 1 - \frac{(\sum_{e \in L_i} list(e)) - |L_i|}{|L_i| \times |L_{all}|}$$

where $|L_i|$ denotes the number of elements returned by the i-th technique and $|L_{all}|$ denotes the number of elements that appear in at least one of the lists returned by the techniques.

Example. Based on the example in Table 5.1 and the top 3 most suspicious program elements returned by each technique, L_{Ochiai}, $L_{Klosgen}$, and $L_{Pietatsky\ Shapiro}$ are {Block 3, Block 4, Block 5}, {Block 3, Block 4, Block 5}, and {Block 2, Block

1, Block 5} respectively. Thus, L_{all} is {Block 1, Block 2, Block 3, Block 4, Block 5}. Also, $list(Block3)$, $list(Block4)$, and $list(Block5)$ are 2, 2, and 3 respectively. From the above, the weight assigned to Ochiai is 1-$\frac{7-3}{3\times5}$ = 0.73. In the same way, the weights of Klosgen and Piatetsky Shapiro are 0.73 and 0.87, respectively. Therefore, the set of new scores for these blocks calculated using Equation 5.1 is {1.87, 0.87, 1.46, 1.46, 2.33}.

In this work, we denote the variant of the CorrB method which calculates the correlation among the measures based on the top 10% of the most suspicious program elements by $CorrB\_Top10\%$ and another variant of the CorrB method which is based on the top 50% of the most suspicious program elements as $CorrB\_Top50\%$

5) Borda count: This method converts the normalized scores that are assigned to program elements by each selected technique into ranks – program elements with higher scores are given smaller ranks. For each element, this method sums up the *ranking points* of an element given by a set of techniques. The ranking point of an element given by a technique is defined as the substraction of the element's rank in the list produced by the technique from the total number of program elements in the input buggy program.

Let e_j denotes the j-th program element and $r_i(e_j)$ the rank assigned to program element e_j by the i-th technique. Also, let N_e denotes the number of program elements and n the number of techniques. The new score of program element e_j is then calculated using Borda count as follows:

$$Score(e_j) = \sum_{i=1}^{n}(N_e - r_i(e_j))$$

Example. Table 5.3 shows the ranking points for each program block in Figure 5.1 given by Ochiai, Klosgen, and Piatetsky Shapiro. The set of new scores for Blocks 1 to 5 based on the summation of their ranking points is {4, 3, 4, 4, 7}.

Block	**Ranks**	**Ranking Points**
Block 1	{4, 4, 3}	{1, 1, 2}
Block 2	{5, 5, 2}	{0, 0, 3}
Block 3	{3, 3, 5}	{2, 2, 0}
Block 4	{3, 3, 5}	{2, 2, 0}
Block 5	{3, 3, 2}	{2, 2, 3}

Table 5.3: Example of Ranks And Ranking Points Given By Ochiai, Klosgen, and Piatetsky Shapiro

5.4 Empirical Evaluation

In this section, we present the subject programs used in our evaluation, our evaluation metrics, and our experiment results. We also discuss some threats to validity.

5.4.1 Dataset

We evaluate the effectiveness of data fusion methods presented in Section 5.3.4 to localize faults in 230 buggy programs. Each buggy program contains a single bug that could span across multiple program elements. Table 5.4 briefly describes our subject programs.

In this work, we evaluate eight subject programs written in C: one real program, namely space, from the Software-artifact Infrastructure Repository (SIR) [125] and the seven Siemens programs [58, 125], namely print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Each of the eight programs has a number of buggy versions. We manually instrument these buggy programs at block level. After excluding buggy versions that could not be reached by our instrumentation, e.g., bugs in global variable declarations, we evaluate 154 buggy versions.

We also analyze two real Java programs from the SIR [125] namely: NanoXML [122] and XML-Security [122]. In this work, we analyze four versions of NanoXML (i.e., NanoXML v1, NanoXML v2, NanoXML v3, and NanoXML v5) and three versions of XML-Security (i.e., XML-Security v1, XML-Security v2, and XML-Security v3). Similarly, each of these programs has multiple buggy versions and we instrument them at block level. After excluding buggy versions that have no failed test case, we evaluate 46 buggy versions.

As previously shown in Table 2.1, the above subject programs have often been used to evaluate many fault localization techniques. Despite their popularity, the size of the above subject programs are relatively small. Most of the bugs in the above subject programs are also synthetic rather than real. Thus, we create another dataset consisting of 30 real bugs from three larger programs namely Rhino [33], Lucene [2], and Ant [1]. Rhino bugs and test suite are obtained from the iBugs repository which was created by Dallmeier and Zimmermann [33]. Lucene and Ant bugs and test suite are obtained from their bug tracking and version control systems following the procedure described by Dallmeier and Zimmermann [33]. In particular, we consider Lucene bugs that were reported for versions 2.0 to 3.1, and Ant bugs that were reported for versions 1.5.1 to 1.9.2. Kawrykow and Robillard observed that not all changes made to fix a bug are essential; many of them are cosmetic changes or simple refactorings that do not change the behavior of a program [71]. Thus, to find the root cause of a real bug (i.e., the buggy program blocks), we need to manually inspect the code that are changed to fix the bug. To help us in the manual inspection, we only include Rhino, Lucene, and Ant bugs whose fixes only change at most five lines of code. Among these bugs, we also only choose bugs that have at least one faulty test case covering the faulty lines.

5.4.2 Evaluation Criteria

A fault localization technique (either using data fusion or not) generates a list of most suspicious program elements for developers' inspection. We evaluate the effectiveness of a fault localization technique in localizing faults based on the following commonly used metrics:

1) Percentage of Code Inspected: For each bug, we measure the percentage of program blocks that a developer needs to inspect to locate all the faulty program

Dataset	LOC	Num. of Buggy Version	Num. of Test Cases
print_token	478	5	4,130
print_token2	399	10	4,115
replace	512	31	5,542
schedule	292	9	2,650
schedule2	301	9	2,710
tcas	141	36	1,608
tot_info	440	19	1,051
space	6,218	35	13,585
NanoXML v1	3,497	6	214
NanoXML v2	4,007	7	214
NanoXML v3	4,608	9	216
NanoXML v5	4,782	8	216
XML security v1	21,613	6	92
XML security v2	22,318	6	94
XML security v3	19,895	4	84
Rhino	49k	11	20-152
Lucene	88k	9	1,072-1,154
Ant	264k	10	1,024-1,555

Table 5.4: Dataset descriptions

elements. The percentage of program blocks that needs to be inspected depends on the rank of the faulty program elements in the list. We report the average percentage of code (i.e., program blocks) inspected over all the bugs.

2) Proportion of Bugs Localized: We compute the proportion of bugs that can be localized when developers inspect up to a certain percentage of program blocks.

3) Absolute Amount of Code Inspected: As studied by Panin and Orso [106], developers may only inspect a certain number of most suspicious program elements recommended by an automatic debugging tool. Thus, we also compute the number of bugs that can be localized when developers inspect up to E program blocks. In this study, we set E to 10.

5.4.3 Experiment Results

We evaluate the effectiveness of various variants of our approach to localize faults. We denote a variant of *Fusion Localizer* as $F_Z^{X,Y}$ where X specifies a score normalization method (i.e., Zero-One or Reciprocal), Y specifies a technique selection method (i.e., Overlap or Bias), and Z specifies a data fusion method such as CombANZ, CorrA_Top10%, CorrA_Top50%, CorrB_Top10%, CorrB_Top50%, CombSUM, CombMNZ, or Borda count.

Their effectiveness is compared with the effectiveness of the well-known spectrum-based fault localization techniques, namely Tarantula [65] and Ochiai [4, 6]. We also compare with the theoretically best spectrum-based fault localization techniques [143] namely Naish1, Naish2, Binary, Wong1, and Russel & Rao, as well as the theoretically best genetic programming (GP) based fault localization techniques namely GP02, GP03, GP13, and GP19 [144].

Percentage of Code Inspected

In terms of average percentage of code inspected to localize all bugs, twelve variants of *Fusion Localizer* perform better than the state-of-the-art fault localization techniques. These variants fuse the techniques using CombANZ, CorrA_Top10%, CorrA_Top50%, CorrB_Top10%, CorrB_Top50%, CombSUM, and CombMNZ by first normalizing the scores using Zero-one normalization and selecting the techniques using either the bias-based or overlap-based method. They could achieve smaller average percentage of code inspected (i.e., 21.36% to 23.78%) as compared to the best performing state-of-the-art fault localization technique (i.e., Naish2). Among the state-of-the-art fault localization techniques, Naish2 achieves the smallest average percentage of code to be inspected (i.e., 24.63%), followed by GP13 (i.e., 24.78%) and Ochiai (i.e., 25.29%). See Table 5.5 for the details of average percentage of code inspected for the above variants of *Fusion Localizer* and the state-of-the-art fault localization techniques.

To investigate whether the differences in the average percentage of code inspected between our twelve variants and the best performing state-of-the-art fault localization techniques (i.e., Naish2, GP13, and Ochiai) are significant or not, we perform a statistical significance test namely Wilcoxon signed rank test [139] at 5% significance level. This statistical test does not assume that the data should

Technique	**Average**	**Technique**	**Average**
$F^{Zero-One,Overlap}_{CombANZ}$	21.36%	Naish2	24.63%
$F^{Zero-One,Bias}_{CombANZ}$	21.39%	GP13	24.78%
$F^{Zero-One,Bias}_{CombSUM}$	22.94%	Ochiai	25.29%
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	23.11%	GP03	25.82%
$F^{Zero-One,Overlap}_{CombSUM}$	23.15%	Tarantula	26.77%
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	23.23%	GP19	31.60%
$F^{Zero-One,Overlap}_{CombMNZ}$	23.31%	Naish1	34.40%
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	23.33%	GP02	39.48%
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	23.38%	Russel&Rao	42.48%
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	23.56%	Binary	52.04%
$F^{Zero-One,Bias}_{CombMNZ}$	23.78%	Wong1	86.26%
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	23.78%		

Table 5.5: Average Percentage of Code Inspected to Localize All Bugs.

follow normal distribution. Based on the significance tests, $F^{Zero-One,Bias}_{CombANZ}$ and $F^{Zero-One,Overlap}_{CombANZ}$ significantly outperform Naish2 with p-values equal to 0.0135 and 0.02284, respectively. They also significantly outperform GP13 and Ochiai with p-values less than 0.05. The other ten variants could only significantly outperform Ochiai with p-values less than 0.05. Table 5.6 lists the p-values when we compare each of the best performing *Fusion Localizer* variants with each of the best performing state-of-the-art fault localization techniques using Wilcoxon signed rank test.

Furthermore, we plot the improvements of the best performing variant of *Fusion Localizer*, namely $F^{Zero-One,Bias}_{CombANZ}$ over the three best performing state-of-the-art fault localization techniques namely Naish2, GP13, and Ochiai for each of the buggy versions in Figures 5.2, 5.3, and 5.4 respectively. We plot the improvements made by $F^{Zero-One,Bias}_{CombANZ}$ because it achieves the most significant improvement over Naish2 (i.e., the lowest p-value over Naish2). The improvement is based on the difference in percentage of code inspected. The graphs only show the buggy versions where the improvements are either positive (our technique performs better) or negative (our technique performs worse). There are 87 buggy versions in which

Fusion Localizer Variant	**p-value over Ochiai**	**p-value over GP13**	**p-value over Naish2**
$F^{Zero-One,Overlap}_{CombANZ}$	1.70e-07	0.0182	0.0228
$F^{Zero-One,Bias}_{CombANZ}$	3.12e-05	0.0104	0.0135
$F^{Zero-One,Bias}_{CombSUM}$	4.30e-06	0.2464	0.2736
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	1.54e-06	0.2966	0.2985
$F^{Zero-One,Overlap}_{CombSUM}$	1.75e-06	0.5985	0.6189
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	1.21e-05	0.5495	0.5642
$F^{Zero-One,Overlap}_{CombMNZ}$	1.24e-05	0.6354	0.7402
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	4.09e-06	0.2047	0.2464
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	2.90e-06	0.2981	0.2948
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	1.00e-05	0.4842	0.5231
$F^{Zero-One,Bias}_{CombMNZ}$	0.0032	0.7860	0.8588
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	1.74e-05	0.3065	0.3065

Table 5.6: Statistical Significance Test Results

$F^{Zero-One,Bias}_{CombANZ}$ improves Naish2 and GP13. As compared to Ochiai, it has better performance for 91 buggy versions. In addition, there are 66, 65, and 88 buggy versions in which $F^{Zero-One,Bias}_{CombANZ}$ performs the same as Naish2, GP13, and Ochiai respectively. Based on the figures, our best variant outperforms the three best performing state-of-the-art fault localization techniques for most of the buggy versions.

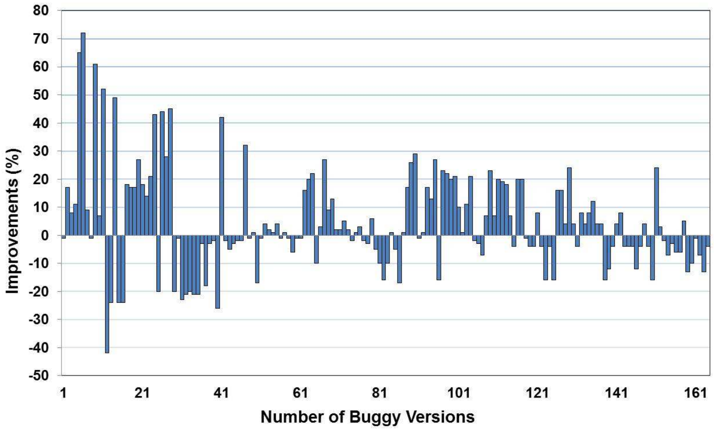


Figure 5.2: Improvement of $F^{Zero-One,Bias}_{CombANZ}$ over Naish2 in Terms of Percentage of Code Inspected

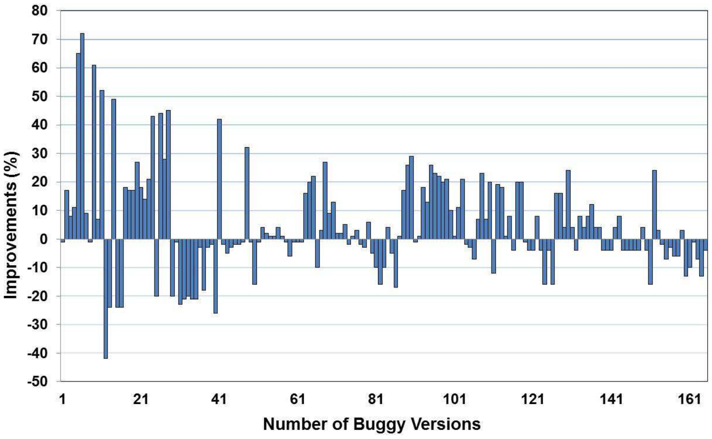


Figure 5.3: Improvement of $F_{CombANZ}^{Zero-One,Bias}$ over GP13 in Terms of Percentage of Code Inspected

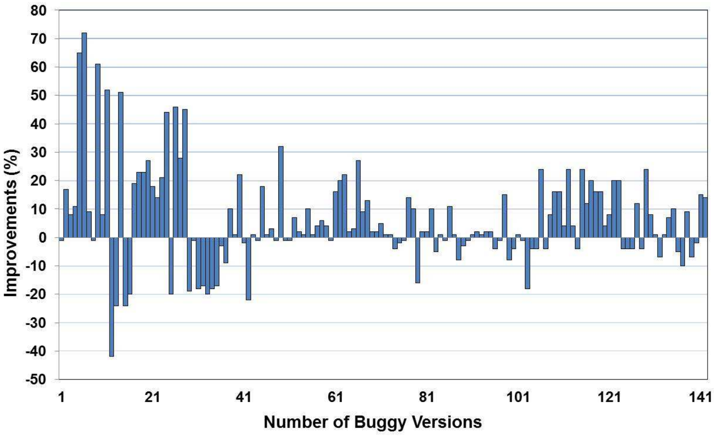


Figure 5.4: Improvement of $F_{CombANZ}^{Zero-One,Bias}$ over Ochiai in Terms of Percentage of Code Inspected

Technique	%Bug	Technique	%Bug
$F^{Zero-One,Overlap}_{CombANZ}$	46.96%	$F^{Zero-One,Bias}_{CombMNZ}$	38.70%
$F^{Zero-One,Bias}_{CombANZ}$	46.52%	GP03	38.70%
$F^{Zero-One,Bias}_{CombSUM}$	43.48%	GP02	37.83%
$F^{Zero-One,Overlap}_{CombSUM}$	43.48%	Tarantula	37.39%
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	43.48%	Naish2	36.96%
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	43.48%	GP13	36.96%
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	43.48%	Naish1	36.09%
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	43.04%	GP19	29.13%
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	43.04%	Russel&Rao	5.65%
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	42.61%	Binary	4.78%
Ochiai	42.17%	Wong1	3.04%
$F^{Zero-One,Overlap}_{CombMNZ}$	41.30%		

Table 5.7: Proportion of Bug Localized When Only 10% of Blocks are Inspected.

Proportion of Bugs Localized

We also evaluate the number of buggy versions in which the faulty program elements could be found when developers inspect a certain percentage of program elements (i.e., program blocks). Table 5.7 shows the proportion of bug localized when developers only inspect the top 10% of the most suspicious program elements produced by the state-of-the-art fault localization techniques and our twelve variants that have better average percentage of code inspected than the best performing state-of-the-art fault localization techniques.

When 10% of the most suspicious program elements are inspected, $F^{Zero-One,Overlap}_{CombANZ}$ and $F^{Zero-One,Bias}_{correlation-based}$ can localize more bugs as compared to the other variants of *Fusion Localizer* and the state-of-the-art fault localization techniques. They can localize 46.96% and 46.52% of the bugs, while the best performing state-of-the-art fault localization technique (i.e., Ochiai) can localize 42.17% of the bugs. Naish2 and GP13 can localize only 36.96% of the bugs. Thus, these two best variants could improve the best performing state-of-the-art fault localization techniques by 11.26% to 26.95%.

In addition, there are eight other variants of *Fusion Localizer* which can localize

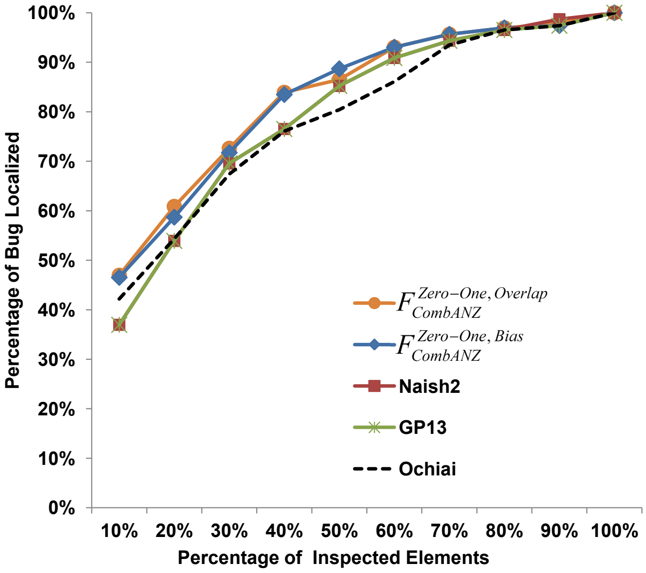


Figure 5.5: Comparing $F^{Zero-One,Bias}_{CombANZ}$, $F^{Zero-One,Overlap}_{CombANZ}$ with Naish2, GP13, and Ochiai

more bugs than the best performing state-of-the-art fault localization technique (i.e., Ochiai) when 10% of program elements are inspected. They can localize 42.17% to 43.48% of the bugs. Thus, our best variants can improve number of bugs that can be localized when developers only inspect a small percentage of code.

Furthermore, we plot the proportion of bug localized using our two best variants (i.e., $F^{Zero-One,Overlap}_{CombANZ}$ and $F^{Zero-One,Bias}_{CombANZ}$), Naish2, GP13, Ochiai, and Tarantula when different percentage of program elements are inspected, as shown in Figures 5.5. Based on the figure, our two best variants can localize more bugs when different percentages of program elements are inspected as compared to the best performing state-of-the-art fault localization techniques.

Absolute Amount of Code Inspected

We also investigate the number of bugs that can be localized when only a small number of program elements (i.e., 10 basic blocks) are inspected. Table 5.8 shows that our twelve variants can localize more bugs than the state-of-the-art fault localization techniques. They can localize 78 to 91 bugs, while the best performing state-of-the-art fault localization technique (i.e., Ochiai) can only localize 74 bugs. Amongst our variants, $F^{Zero-One,Bias}_{CombANZ}$ can localize the largest number of bugs as compared to

Technique	Hit@10	Technique	Hit@10
$F^{Zero-One,Bias}_{CombANZ}$	91	Ochiai	74
$F^{Zero-One,Overlap}_{CombANZ}$	87	Naish1	73
$F^{Zero-One,Overlap}_{CombSUM}$	87	Naish2	73
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	86	GP13	72
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	86	GP03	71
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	85	GP02	64
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	85	Tarantula	56
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	85	GP19	51
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	84	Russel&Rao	3
$F^{Zero-One,Bias}_{CombSUM}$	84	Binary	3
$F^{Zero-One,Overlap}_{CombMNZ}$	84	Wong1	0
$F^{Zero-One,Bias}_{CombMNZ}$	78		

Table 5.8: Number of Bugs Localized When Only Up To 10 Most Suspicious Program Elements Are Inspected (i.e., Hit@10)

the other techniques. Its relative improvement over Ochiai, Naish2, and GP13 are 23%, 25%, and 26% respectively. These results also show that our best variants can substantially improve the number of bug localized when developers only inspect a small number of program elements.

5.4.4 Discussion

In this section, we first discuss the effect of score normalization on the performance of *Fusion Localizer*. We then describe the effect of technique selection on the performance of *Fusion Localizer*. Next, we describe the effect of varying the number of techniques to be fused together. At the end of this section, we describe some threats to validity.

Effect of Score Normalization. To investigate the effect of the score normalization step, we disable this step and evaluate the effectiveness of the resultant solution. We find that without score normalization, the performance of the best performing variant of *Fusion Localizer* is not as good as the best performing variant that normalizes the scores.

Without normalization, the average percentage of program block inspected to localize all bugs achieved by the best variant is 30.28% which is larger than the result of the best variant that normalizes the scores (i.e., 21.36%). When developers only inspect up to 10% of program blocks, the best variant that does not normalize the scores can only localize 30% of the bugs which is smaller than the percentage of bugs that are successfully localized by the best variant that normalizes the scores (i.e., 46.96%). Also, when inspecting up to 10 program blocks, the number of bugs localized by the best variant that does not normalize the scores is not even half of those achieved by the best variant that normalizes the scores (38 bugs vs. 91 bugs). Therefore, normalization is an important step to improve the performance of *Fusion Localizer*.

Effect of Technique Selection. To investigate the effect of the technique selection step, we disable this step and evaluate the effectiveness of the resultant solution. When we fuse all 42 fault localization techniques, the performance of the best variant is worse than the best variant that employs technique selection. When the technique selection step is disabled, on average, the best variant can localize all bugs when 22.27% of code are inspected, which is not as good as the result achieved by the best variant that employs technique selection (i.e., 21.36%). When developers only inspect up to 10% of program blocks, the best variant that does not use technique selection localizes a smaller percentage of bugs than the percentage localized by the best variant that uses technique selection (i.e., 46.52% vs. 46.96%). Similarly, when developers only inspect up to 10 program blocks, the best variant that does not use technique selection localizes a smaller number of bugs than the number localized by the best variant that employs technique selection (i.e., 87 vs. 91). Therefore, applying technique selection can improve the performance of *Fusion Localizer*.

Effect of Number of Techniques to be Fused. To evaluate the effect of the number of techniques to be fused by our *Fusion Localizer*, we use one of our best variants (i.e., $F^{Zero-One,Bias}_{CombANZ}$) and set the number of techniques to be selected to top 25%, top 50%, top 75%, and all of the techniques. Let us refer to the 4 sub-variants of $F^{Zero-One,Bias}_{CombANZ}$ as V25, V50, V75, and V100. By default *Fusion Localizer* selects top 50% of the techniques. We find that the average percentage of code inspected to locate all bugs are 27.13%, 21.36%, 21.88%, and 22.27% for V25, V50, V75, and V100 respectively. When developers only inspect the top 10% of the code, the percentage of bugs localized by V25, V50, V75, and V100 are 30%, 46.96%, 46.52%, and 46.52% respectively. When developers only inspect the top 10 program blocks, the number of bugs localized by V25, V50, V75, and V100 are 57, 91, 87, and 87 respectively. The results show that selecting the top 50% of the techniques (the default option) is the best setting.

Threats to Validity. Threats to internal validity relates to errors in our experiments. We have checked our implementation but there could be bugs that we do not notice. Threats to external validity relates to the generalizability of our findings. We have analyzed 230 bugs from 13 programs written in C and Java. The programs vary from small to large programs. The bugs vary from synthetic to real bugs. In the future, we plan to reduce this threat to external validity further by investigating more bugs from more systems written in various programming languages. Threats to construct validity relates to the suitability of our evaluation metrics. We have used common metrics used to analyze past fault localization studies [4, 6, 31, 65, 66, 137]. We have also used another metric (i.e., Hit@10) to address the concern raised by Parnin and Orso [106]. Hit@10 only considers the performance of a fault localization tool when a small number of program elements (in our case: program blocks) are inspected. The measure was previously used by information retrieval (IR) based bug localization studies that find buggy program files given a textual bug report [115, 120, 136, 154].

5.5 Conclusion

In this chapter, we propose an approach named *Fusion Localizer* to fuse a number of spectrum-based fault localization techniques. Our propose approach consists of three steps: score normalization, technique selection, and data fusion. We investigate two score normalization methods, two technique selection methods, and five data fusion methods resulting in twenty variants of *Fusion Localizer*. *Fusion Localizer* does not require any training data, i.e., execution traces of past relevant program failures, to select the set of techniques to be fused. This allows our approach to be used for new programs or programs where program spectra of past relevant bugs are unavailable. Furthermore, our approach is bug specific in which the set of techniques to be fused is adaptively selected for each buggy program based on its spectra.

We evaluate our approach using a common benchmark dataset and a dataset that contains real bugs from three medium to large programs. Our evaluation shows that the best performing variants of *Fusion Localizer* (i.e., $F^{Zero-One,Bias}_{CombANZ}$ and $F^{Zero-One,Overlap}_{CombANZ}$) statistically significantly outperform the state-of-the-art spectrum-based fault localization techniques (i.e., Ochiai, theoretically best spectrum-based fault localization techniques, and theoretically best genetic programming (GP) based fault localization techniques). Our best variants require smaller average percentage of code inspected to locate faulty elements as compared to the best performing state-of-the-art fault localization techniques (i.e., 21% vs. 24%). When developers only inspect 10% of the most suspicious program elements, these best variants could improve the best performing state-of-the-art fault localization techniques (i.e., Ochiai, Naish2, and GP13) by 11% to 26%. Furthermore when developers only inspect the top 10 most suspicious program blocks, these best variants could improve the best performing state-of-the-art fault localization techniques by 23% to 26%. In addition, there are many other variants of *Fusion Localizer* that can outperform the state-of-the-art fault localization techniques (albeit not statistically significantly). These variants use CorrA_Top10%, CorrB_Top10%, CorrB_Top50%, CombSUM, and CombMNZ to fuse the selected techniques, by first normalizing the scores using Zero-One normalization and by selecting the techniques using the bias-based or overlap-based selection methods.

Chapter 6

Active Refinement of Clone Anomaly Reports

In this chapter, we propose an approach that complements spectrum-based fault localization for cases in which test cases or program spectra are not available to be analyzed to locate faults in programs. In the absence of program spectra, clone-based anomaly detection techniques can find real bugs on large systems, nevertheless these techniques often have a high false positive rate. In order to reduce the false positive rate of such techniques, we propose an active learning method to re-rank anomaly reports produced by a clone-based anomaly detection technique so that the anomaly reports that contain faults would appear in the top positions for developers' inspection.

We organize this chapter as follows. Section 6.1 presents the motivation and main contribution of this chapter. In Section 6.2, we review the concept of clone-based anomaly detection. In Section 6.3, we describe our overall active refinement framework which iteratively invokes a refinement engine. We elaborate this engine in Section 6.4. Our evaluation metric is described in Section 6.5. In Section 6.6, we describe our evaluation results on three large software systems. We conclude and mention potential future work in Section 6.7.

6.1 Code Clones for Bug Detection

Code clones, or pieces of similar code, commonly occur in large software systems [19, 86] for various reasons, which range from improper code reuse via the prevalent copy-and-paste practice, to the introduction of redundant code to improve runtime efficiency and/or reliability of systems. Code clones have attracted much research interest resulting in various studies on detecting code clones [17,19,62,69], tracking and managing code clones [36, 57, 101], and examining the harmfulness or usefulness of code clones [68, 70, 72].

One important use of code clones is their applicability in detecting bugs [43, 53, 64, 68, 78, 81]. These *clone-based anomaly detection* tools look for *inconsistencies* among code clones in every clone group (i.e., a group of code fragments similar to each other) and report them as anomalies (i.e., potential bugs). For example, Li *et al.* [81] look for different identifier names among clones and check whether all names are changed consistently; Jiang *et al.* [64] look at syntactic structures of the code surrounding every clone, in addition to the identifier names in clones, and report differences as anomalies. These tools have found a number of true positives of diverse characteristics from large systems, such as the Linux kernel and Eclipse. Figure 6.2 shows a true positive from the Linux kernel: there is a missing null-check on the `tmp` variable in the code fragment 2. Such a detection is possible because most parts of the two code fragments (after the `if` condition) are detected as clones, and the code surrounding the clones (which are the variable declaration and the `if` condition) shows some structural differences (no `if` in code fragment 2).

However, the set of reported anomalies can be huge, containing hundreds or even thousands of reports. Among these anomalies, only a small proportion are true positives; others are *benign variations* among clones in a clone group, which are intended changes rather than mistakes. Verifying whether these anomalies are true or not can be painstaking and time-consuming. Developers tend to give up if many of the first set of anomaly reports that they check are false positives. For example,

Case 1: Without refinement
Jack is presented with 500 bug reports.He investigates the first 100, and can find five true positives. If the bugs are mission critical, it's worth the effort.
Case 2: With refinement
Jack is presented with 500 bug reports. As he navigates through the bug reports and labels each of them as true bugs or false positives, the system automatically reorders the remaining unlabeled bug reports. He can now find ten true positives after investigating 100 reports. Jack's productivity in finding bugs is doubled.

Table 6.1: Informal illustration: Refinement process

Jiang *et al.* [64] report that among more than 800 reports generated by their tool for the Linux kernel, only 57 are true bugsor bad programming styles. Gabel *et al.* [43] apply more advanced filtering techniques based on textual similarity and sequence alignment on inconsistent clones detected from a large commercial code base. They report that among 500 manually checked anomaly reports (out of 8103 in total), 149 may be true bugs and 109 may be code smells, while for the rest they are unsure. Hence, reducing the manual effort in locating true positives in clone-based anomaly reports is an important task for the wide adoption of such tools.

In this chapter, we propose an active-learning and user-feedback directed approach to help alleviate the problem of false positives. The task is challenging as there are only a few true positives embedded in a mass of false positives. Our idea is to actively, iteratively incorporate user feedback to refine anomaly reports. Users are presented anomaly reports one by one; when a user labels a report as a false positive or true positive, our system *actively updates* the remaining set of anomaly reports. In so doing, we aim to make true positives appear earlier in the list of all anomaly reports according to this information. Thus, we provide a feedback loop between bug detection tools and developers, and help to improve the quality of anomaly reports and reduce the effort of manual investigation. Table 6.1 presents two scenarios for an informal illustration.

Now we describe how this active refinement of anomaly reports could be performed. Conceptually, we divide the space of possible clone groups into four quad-

	Inconsistent	**Consistent**
Rigid	True Positive	✓
Flexible	✓	✓

Figure 6.1: Clone group's four quadrants

rants as shown in Figure 6.1. The columns separate clone groups that *have* inconsistencies from those that do not; the rows separate clone groups that *allow* variations (i.e., *flexible*) from those that do not (i.e., *rigid*). A rigid clone group is a set of clones where variations among clones are harmful; a flexible clone group is a set of clones where variations are benign.[1] Current clone-based anomaly detection tools would separate clone groups in the two quadrants on the left from those in the two quadrants on the right. However, clone groups in the bottom left quadrant would be all false positives since the inconsistent changes in those clones are allowed or intentional and should not be reported as anomalies. The goal of our approach is to learn a *discriminative model* to provide the likelihood of a clone group belonging to the top left quadrant (i.e., rigid but inconsistent) versus belonging to the bottom left quadrant (i.e., flexible and inconsistent). Then, this model can be used to re-sort the list of anomaly reports and make true positives appear earlier.

We observe that false positives could be similar to one another in certain ways. For example, consider the code snippets in Figure 6.3 containing two clone groups with two clones each. All of code snippets involve the same number of `if` statements, `||` operators, function calls, and assignments, but they are quite different from the true positive shown in Figure 6.2. Thus, the intuition of our approach is that false positives may have similar characteristics among themselves, but they have different characteristics from true positives, and the differences between false and true positives could be leveraged to build a discriminative model.

In order to characterize the similarities and differences among clones, we con-

[1] Both notions allow gapped clones, and are orthogonal to the concept of gapped clones.

vert each clone group into a set of features. These features are built from the various syntactical patterns in the clones of each group. A discriminative model is a composition of features that collectively captures the differences between false and true positives. Often such models are built from a fixed *static* training dataset, e.g., [22, 79]. However, in our bug refinement process, the training dataset is incrementally updated as a new anomaly report is inspected and marked by a developer as either a false or true positive, and we would like to build our discriminative model based on such a *dynamic* training dataset.

We propose a framework consisting of a refinement engine that leverages user feedbacks and is iteratively invoked. Developers need to take action on anomaly reports, to either get bugs fixed or discard them. Our feedback is from such actions and no extra effort is needed. The refinement engine is composed of a feature extractor, a pre-processor, and a classifier, arranged in a pipeline. It takes in the given feedback, to build and refine the discriminative models. The resultant discriminative model in each iteration is used to refine the remaining uninvestigated anomaly reports.

We evaluate our framework on three sets of clone anomaly reports for three large programs: the Linux kernel (C), Eclipse, and ArgoUML (Java) [64] extracted by a clone-based anomaly detection tool. Our evaluation shows that compared to the original ordering of bug reports, we can improve the *average percentage of true positives found*, an evaluation metric adopted from the test case prioritization community [37], by 11%, 87%, and 86% for the Linux kernel, Eclipse, and ArgoUML respectively. We further discuss these improvements in Section 6.6.

The main contributions of this work are as follows:

Code Fragment 1

```
File: linux-2.6.19/fs/sysfs/inode.c

219: struct dentry * dentry = sd->s_dentry;
220:
221: if (dentry) {
        /* the following parts are detected as clones */
222:         spin_lock(&dcache_lock);
223:         spin_lock(&dentry->d_lock);
224:         if (!(d_unhashed(dentry) && dentry->d_inode)) {
225:                 dget_locked(dentry);
226:                 __d_drop(dentry);
227:                 spin_unlock(&dentry->d_lock);
228:                 spin_unlock(&dcache_lock);
229:                 ......
```

Code Fragment 2

```
File: linux-2.6.19/drivers/infiniband/hw/ipath/ipath_fs.c

456: struct dentry *tmp;
457:
458: tmp = lookup_one_len(name, parent, strlen(name));
459:
460: spin_lock(&dcache_lock);
461: spin_lock(&tmp->d_lock);
462: if (!(d_unhashed(tmp) && tmp->d_inode)) {
463:         dget_locked(tmp);
464:         __d_drop(tmp);
465:         spin_unlock(&tmp->d_lock);
467:         spin_unlock(&dcache_lock);
468:         ......
```

Figure 6.2: A sample bug (missing null-check) revealed by contextual inconsistency among clones in a clone group from the Linux kernel – compare lines 221 & 224 in code fragment 1 with lines 459 & 462 in code fragment 2.

G# 1

Code Clone 1

```
File: linux-2.6.19/fs/nfsd/nfs3xdr.c

423: if (!(p = decode_fh(p, &args->fh))
424:  || !(p = decode_filename(p, &args->name, &args->len))
425:  || !(p = decode_sattr3(p, &args->attrs)))
426:         return 0;
```

Code Clone 2

```
File: linux-2.6.19/fs/nfsd/nfsxdr.c

344: if (!(p = decode_fh(p, &args->ffh))
345:  || !(p = decode_fh(p, &args->tfh))
346:  || !(p = decode_filename(p, &args->tname, &args->tlen)))
347:         return 0;
```

G# 2

Code Clone 1

```
File: linux-2.6.19/drivers/hwmon/lm87.c

688: if ((err = device_create_file(&new_client->dev,
689:                         &dev_attr_in6_input))
690:  || (err = device_create_file(&new_client->dev,
691:                         &dev_attr_in6_min))
692:  || (err = device_create_file(&new_client->dev,
693:                         &dev_attr_in6_max)))
694:         goto exit_remove;
```

Code Clone 2

```
File: linux-2.6.19/drivers/hwmon/gl520sm.c

615: if ((err = device_create_file(&new_client->dev,
616:                                 &dev_attr_in4_input))
617:  || (err = device_create_file(&new_client->dev,
618:                                 &dev_attr_in4_min))
619:  || (err = device_create_file(&new_client->dev,
620:                                 &dev_attr_in4_max)))
621:         goto exit_remove_files;
```

Figure 6.3: False positive clone groups in Linux Kernel. Each row is a pair of inconsistent clones that do not correspond to bugs. Each pair of clones involve the same numbers of `if` statements, `||` operators, function calls, and assignments.

1. We present the topic of refining clone anomaly reports.
2. We propose an active learning approach to incrementally refine anomaly reports with user feedback.
3. We present an engine that learns discriminative models that can assign the likelihood of each anomaly report being a false positive.
4. We evaluate our proposed approach on three large systems—the Linux Kernel, Eclipse, and ArgoUML—with promising results.

6.2 Clone-Based Anomaly Detection

Clone-based bug detection techniques [64, 81] are based on code clone detection and the concept of *contextual consistency*. The intuition behind the technique is that code clones should be inherently similar to each other, and inconsistent changes to the clones themselves or their surrounding code (which are called *contexts*) may indicate unintentional changes, bad programming styles, and bugs [64].

We summarize the technique as follows:

1. It uses a code clone detection tool, DECKARD [62], to detect code clones in programs. The output of this step is a set of clone groups, where each clone group is a set of code pieces that are syntactically similar to each other (*a.k.a.* clones);
2. Then, it locates the locations of every clone in the source code and generates parse (sub)trees for them;
3. Next, it detects inconsistencies among the parse trees of the clones and their contexts, e.g., whether the clones contain different numbers of unique identifiers, and how the language constructs of the contexts are different. The inconsistencies are heuristically ranked based on their potential relationship with bugs. Inconsistent clones unlike to be buggy are also filtered out.
4. Finally, it outputs a list of anomaly reports, each of which indicates the location of a potential bug in the source code, for developers to inspect.

It has been reported that this technique has a high false positive rate, even though it can find true bugs of diverse characteristics that are difficult to detect by other techniques. For example, among more than 800 reported bugs for the Linux Kernel, only 40 are true bugs and another 17 are bad programming style; among more than 400 reported bugs for the Eclipse, only 21 are true bugs and 17 are issues with bad programming style [64].

6.3 Overall Refinement Framework

To alleviate a high false positive rate of anomaly reports produced by clone-based anomaly detection tools, we propose an active learning approach that can *dynamically* and continually refine anomaly reports based on user feedback; each item of feedback is immediately incorporated by our approach into the ordering of anomaly reports to move newly possible true positive reports up in the list while moving likely false positives towards the end of the list.

Our proposed approach is shown in Figure 6.4. It is composed of five parts corresponding to the boxes in the figure.[2] We refer to them as Blocks 1 to 5 (counter-clockwise from left to right).

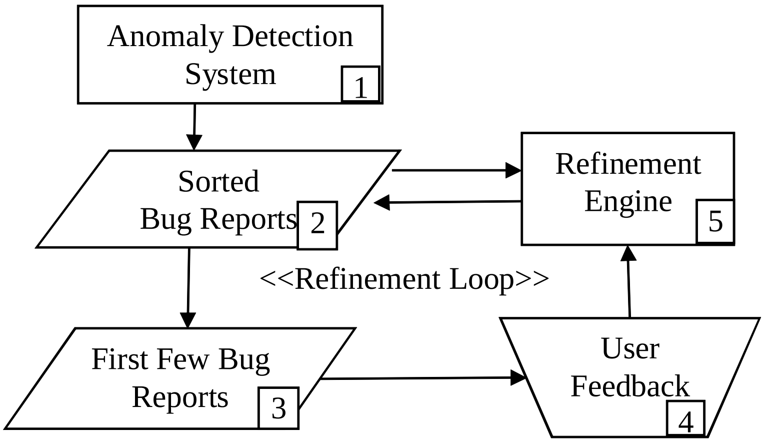


Figure 6.4: Active refinement process

[2] A square, a trapeze, and a parallelogram represent a process, a manual operation, and data respectively.

Block 1 represents a typical batch-mode clone-based anomaly detection system. Given a program, the system identifies parts of the program that are different from the norm, where the norm corresponds to the common characteristics in a clone group. Then, the set of anomalies or bugs (*i.e.*, Block 2) is presented for manual user inspection.

We extend such a clone-based anomaly detection system by incorporating incremental user feedback through the feedback and refinement loop starting at Block 2 followed by Blocks 3, 4, and 5, and back to Block 2. At Blocks 3 and 4, a user is presented with a few bug reports and is asked to provide feedbacks on whether the reports are false or true positives. This feedback is then fed into our refinement engine (*i.e.*, Block 5) to update the original or intermediate lists of bug reports.

With user feedback, the refinement engine analyzes the characteristics of both false positives and true positives labeled by users and hypothesizes about other false positives and true positives in the list based on various classification and machine learning techniques. The resulting hypotheses are then used to rearrange the remaining bug reports. It is possible that a true positive, that is originally ranked low, is moved up in the list; a false positive, that is originally ranked high, may be "downgraded" or pushed down in the list.

The active refinement process repeats and users are asked for more feedbacks. With more iterations, more feedback is received, and a better hypothesis can be made for the remaining unlabeled reports.

The ultimate goal of our refinement process is to produce a *better ordering* of bug reports so that true positive reports are listed ahead of false positives, which we refer to as the *bug report ordering problem*. With a better ordering, true positives can be identified earlier without the need to investigate the entire report list. With fewer false positives appearing early in the list, a developer can be encouraged to continue investigating the rest of the reports and can find more bugs in a fixed period of time. If all (or most) of the true positives appear early, a developer may stop analyzing the anomaly reports once he or she finds many false positives.

6.4 Refinement Engine

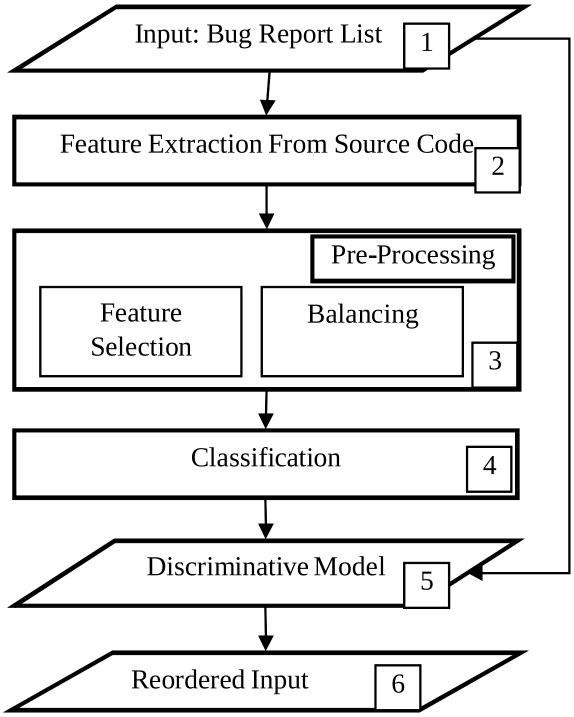


Figure 6.5: Refinement engine

This section further elaborates our refinement engine. Our refinement engine takes in a list of anomaly reports and refines it by reordering. Each anomaly report is a set of code clones (*i.e.*, a clone group) which contain inconsistencies among the clones. Given a list of anomaly reports, ordered either arbitrarily or with some ad-hoc criteria, and user-provided labels (*i.e.*, true positives or false positives) for *some* of the reports, our refinement process reorders *unlabeled* anomaly reports based on the predicted likelihood of each of them being true positives. Figure 6.5 shows our refinement engine that is composed of three main blocks: feature extraction, pre-processing (including feature selection and data balancing), and classification.

The feature extraction aims to transform each clone group into a set of *features*; each feature is a quantitative value that represents a certain property of the code. We refer this set of features to as a *data point*. In our case, we apply feature extraction to each inconsistent clone group reported by the clone anomaly detection tool and collect a set of data points (*a.k.a. a dataset*) for all clone groups in the reported list.

This feature set is then fed to the preprocessor, which analyzes the data points, and may remove some features or data points from the dataset. Its goal is to smooth over data "noise" as much as possible before classification.

The classifier then takes a preprocessed dataset to train a classification model that can discriminate features belonging to one class from the other. In our setting, the two classes are false positives and true positives. We use *class labels* (`False` and `True`) to indicate whether a clone group is a false or true positive. This trained model in turn is used to predict the class labels of the reported clone groups that have received no user feedback. We also design our classification engine to provide the degree of *likelihood* for a clone group to be in each of the two classes, which is used as a key to sort unlabeled clone groups.

6.4.1 Feature Extraction

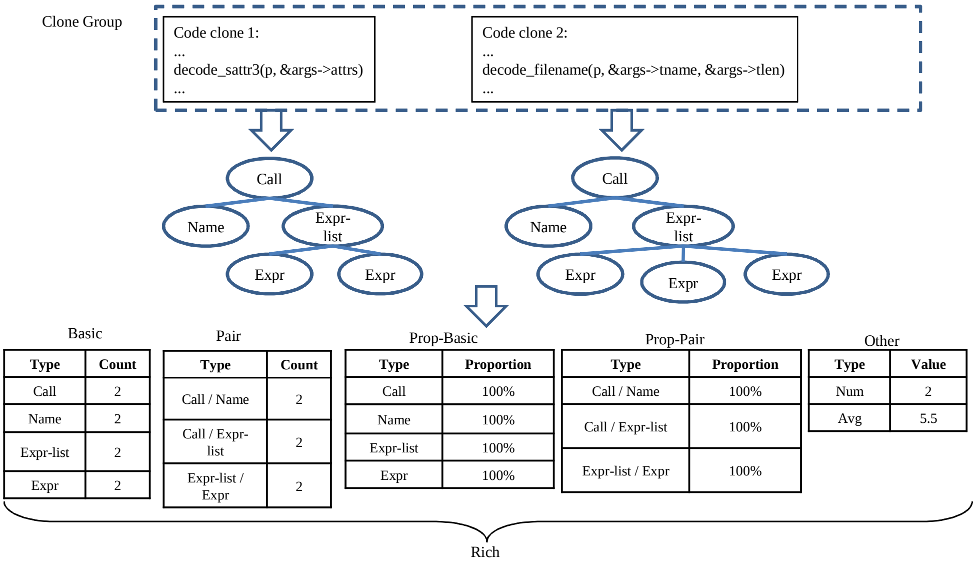
Our feature extraction component analyzes parse trees which are commonly used to represent programs written in various languages. As a benefit, it is easier to adopt our refinement engine to code written in different programming languages.

A parse tree node is labeled with information to represent various program constructs e.g., for, switch, etc. Each clone is reported as a sub-tree rooted in a particular node in a parse tree. The feature extraction constructs parse trees for every reported inconsistent clone group and traverses the trees to collect features. More specifically, it performs the following two steps:

1) Tree Construction: For each clone in the anomaly reports, we invoke a parser on the source file containing the clone to construct a parse tree for the file; each node in the tree contains a label indicating its type (*e.g.*, `for`, `if`, `assignment`, etc.). Then, we locate the root node of the subtree that corresponds to the clone. We refer to this subtree as a *clone tree*. We also locate the scope of the clone which is the first ancestor node of this subtree in the source file and refer to the subtree rooted at this ancestor node as a *clone ancestor tree*.

Clone ancestor trees correspond to more code than clone trees. They may contain more useful information that helps in deciding whether an anomaly report is false or true positive. Thus, we extract features from clone ancestor trees.

2) Representing Clone Ancestor Trees as Features: We define five sets of features that could be extracted from a clone ancestor tree, namely: basic, pair, proportional-basic, proportional-pair, and rich. Consider a clone group CG containing a set of clones corresponding to a set of clone ancestor trees, the five sets of features are defined in Definitions 6.4.1, 6.4.2, 6.4.3, 6.4.4, and 6.4.5. Rich features belong to the most comprehensive feature set that is a superset of the other four feature sets. Our engine would convert the clone ancestor trees into rich features.



Type	Count
Call	2
Name	2
Expr-list	2
Expr	2

Type	Count
Call / Name	2
Call / Expr-list	2
Expr-list / Expr	2

Type	Proportion
Call	100%
Name	100%
Expr-list	100%
Expr	100%

Type	Proportion
Call / Name	100%
Call / Expr-list	100%
Expr-list / Expr	100%

Type	Value
Num	2
Avg	5.5

Figure 6.6: Feature extraction

Definition 6.4.1 (Basic Features) *The basic feature set (*Basic*) of a clone group CG is the set of* type-count pairs *in which each pair contains a node type and the number of parse trees in CG having that particular type. For example, considering the trees in Figure 6.6, the basic feature set contains the following pairs:* $\langle Call, 2\rangle$, $\langle Name, 2\rangle$, $\langle Expr\text{-}list, 2\rangle$, *and* $\langle Expr, 2\rangle$. *Mathematically,* $Basic(CG) =$

$$\left| \begin{array}{l} (t, |CS|), where \\ CS = \{c \in CG \mid c\, has\, a\, node\, of\, type\, t\} \wedge |CS| > 0 \end{array} \right|$$

Definition 6.4.2 (Pair Features) *The pair feature set (*Pair*) of a clone group CG is the set of* type-count pairs *in which each pair contains a* pair of node types *and the number of parse trees in CG having that particular pair. For example, considering the trees in Figure 6.6, the pair feature set contains the following pairs:* $\langle Call/Name, 2\rangle$*,* $\langle Call/Expr\text{-}list, 2\rangle$*, and* $\langle Expr\text{-}list/Expr, 2\rangle$*. Mathematically,* $Pair(CG)$ =

$$\left| \begin{array}{l} ((t_1, t_2), |CS|), where \\ CS = \{c \in CG \mid \exists_{n_1, n_2 \in c}. n_1\ \&\ n_2\ are\ connected \wedge \\ n_1\ is\ of\ type\ t_1\ \wedge\ n_2\ is\ of\ type\ t_2\} \wedge |CS| > 0 \end{array} \right|$$

Definition 6.4.3 (Proportional Features—Basic)

*The proportional-basic feature set (*Prop-Basic*) of a clone group CG is the set of* type-count pairs *in which each pair contains a node type and the* proportion *of parse trees in CG having that particular type. For example, considering the trees in Figure 6.6, the* Prop-Basic *feature set contains the following pairs:* $\langle Call, 100\%\rangle$*,* $\langle Name, 100\%\rangle$*,* $\langle Expr\text{-}list, 100\%\rangle$*, and* $\langle Expr, 100\%\rangle$*. Mathematically,* $PrBasic(CG)$ =

$$\left| \begin{array}{l} (t, \frac{|CS|}{|CG|}), where \\ CS = \{c \in CG \mid c\, has\, a\, node\, of\, type\, t\} \wedge |CS| > 0 \end{array} \right|$$

Definition 6.4.4 (Proportional Features—Pair)

*The proportional-pair feature set (*Prop-Pair*) of a clone group CG is the set of* type-count pairs *in which each pair contains a* pair of node types *and the* proportion *of parse trees of CG having that particular pair. For example, considering the trees in Figure 6.6, the* Prop-Pair *feature set contains the following pairs:* $\langle Call/Name, 100\%\rangle$*,* $\langle Call/Expr\text{-}list, 100\%\rangle$*, and* $\langle Expr\text{-}List/Expr, 100\%\rangle$*. Mathematically,* $PrPair(CG)$ =

$$\left|\begin{array}{l} ((t_1, t_2), \frac{|CS|}{|CG|}), where \\ CS = \{c \in CG \mid \exists_{n_1, n_2 \in c}.n_1 \ \&\ n_2 \ are \ connected \wedge \\ n_1 \ is \ of \ type \ t_1 \wedge n_2 \ is \ of \ type \ t_2\} \ \wedge |CS| > 0 \end{array}\right|$$

Definition 6.4.5 (Rich Features) *The rich feature set (*Rich*) of a clone group CG is the union of the Basic, Pair, Prop-Basic, Prop-Pair feature sets, plus two additional features: the number of clones in CG (Num), and the average size of the clones in CG (Avg). For example, considering the trees in Figure 6.6, the* Rich *feature set is the union of other features sets plus two additional features:* $\langle Num, 2\rangle$*, and* $\langle Avg, 5.5\rangle$*. Mathematically,* $Rich(CG)$ =

$$\begin{array}{c} Basic(CG) \cup Pair(CG) \cup PrBasic(CG) \cup \\ PrPair(CG) \cup \{(Num, |CG|), (Avg, \frac{\Sigma_{c \in CG}.|c|}{|CG|})\} \end{array}$$

6.4.2 Preprocessing

We consider two pre-processing options: feature selection and dataset re-balancing. Feature selection is to reduce the number of features by removing unimportant ones. Unimportant features are noise and are good to be removed. In addition, as our clone anomaly report typically contains many more false positives than true positives, the false positive reports are more likely to get selected as training data. Our classifier uses this training data to learn to discriminate the true positives from the false positives. In order to make the classifier work effectively, we need to reduce the number of false positives in the training data so that it is the same as the number of true positives. By doing so, we produce a discriminative model that is not biased towards always labeling unknown reports as false positives. We refer this process to as dataset re-balancing.

1) Feature Selection: Various approaches have been proposed to select important features. Information gain has been widely used to evaluate the usefulness of a feature, e.g., [112]. If we use c to denote the class labels (true positive [$+ve$ class] vs. false positive [$-ve$ class]), and use f to represent a feature, then the information gain of f is defined as in Eq.(6.1).

$$IG(c|f) = H(c) - H(c|f) \tag{6.1}$$

where $H(c) = -\sum_{c_i \in \{\pm ve\}} P(c_i) \log P(c_i)$ is the entropy and $H(c|f) = -\sum P(f) \sum_{c_i \in \{\pm ve\}} P(c_i|f) \log P(c_i|f)$ is the conditional entropy given the feature f.

We select important features based on information gain as implemented in the Weka toolkit [140] using its default configuration.

2) Dataset Re-balancing: In our dataset, we can group our data into two classes i.e., data that corresponds to true positive reports and data that corresponds to false positive reports. To re-balance the dataset, we reduce the number of data points in the larger class (e.g., false positives). We retain all data points in the smaller class (e.g., true positives). For each data point in the smaller class, we find the *most similar* data points in the larger class and retain it. We use cosine similarity [50] to measure the similarity between two feature sets corresponding to the two data points. Other data points in the larger class are dropped. This is motivated by the nearest neighbor approach by Renieris and Reiss that localizes bugs by comparing the nearest faulty and correct executions [116]. In their setting, they also have the issue of unbalanced dataset: correct executions are much more common than faulty ones.

6.4.3 Classification

The classification block takes preprocessed datasets and learns a discriminative model that discriminates true positives from false ones. We refer to the true and false positives as class labels. The purpose of a discriminative model is to take an unlabeled datapoint (*i.e.*, a datapoint or an anomaly report that is not known to be a true positive or a false positive) and assign a class label to it. To produce a discriminative model, the classifier learns from a given labeled training data points. In our case, the training data points are the clone reports that have been investigated by developers to be true positives or false positives.

In this work, we use a variant of nearest neighbor classification scheme, namely nearest neighbor with non-nested generalization (NNGe) [90]. Nearest neighbor classification has been proved successful for various tasks, e.g., [133]. Also, this technique matches our intuition: an instance similar to known false positives is likely to be a false positive. Our initial study showed that NNGe performs no worse than other common classification approaches.

As the name suggests, nearest neighbor classification labels unknown data with the same label as its nearest neighbor. The time needed to build a model would be small as it only involves index building and distance calculation [96].

The nearest neighbor approach can not generalize or group several data points together, which potentially reduces its accuracy in classification. Salzberg [121] extends the approach with generalization. Rather than loading all data points into memory, the training phase constructs multi-dimensional rectangles (*i.e.*, hyper-rectangles) that generalize a few data points in a multi-dimensional space. This approach has poor performance on some settings due to nested generalization (*i.e.*, hyper-rectangles are contained inside other hyper-rectangles or overlap with one another) [138]. Martin [90] addresses this issue by proposing nearest neighbor with non-nested generalization, called NNGe which we use in this work. In particular, we use the implementation available in Weka [140] with its default distance function.

In this work, we extend NNGe to output the likelihood for a data point dp to belong to each of the two classes (*i.e.*, true positives (T) and false positives (F)). Let's refer to the set of exemplars as D, the set of exemplars that refers to true positive reports is denoted as D_T, and the set of exemplars that refers to false positive reports is denoted as D_F. Also, considering an exemplar d, let $sim(dp, d) = 1 - dist(dp, d)$, where $dist(dp, d)$ is the distance between dp and the exemplar d which ranges from 0 to 1. The relative similarity of a data point to true positive class as compared to false positive class (i.e., $RS(dp)$) is represented by the normalized similarity of the data point dp to all data points in D_T as compared to all data points in D_F. We compute $RS(dp)$ as follows.

$$RS(dp) = \frac{|\sum_{d_T \in D_T} sim(dp, d_T)|}{|D_T|} - \frac{|\sum_{d_F \in D_F} sim(dp, d_F)|}{|D_F|}$$

Based on this relative similarity, we measure the likelihood that an anomaly report is a true positive (i.e., $LH(dp)$) and re-sort the bug reports in descending order of this likelihood. Bug reports with higher likelihood are more likely to be true positives and would be listed first. This likelihood is computed as follows:

$$LH(dp) = 0.5 + \frac{RS(dp)}{2}$$

6.4.4 Concrete Refinement Process

Algorithm 1 is the pseudo-code of our refinement process. The inputs are the list of bug reports (BR) from a bug detection tool, the initial number of bug reports to be labeled (k), and the feedback pool size (p). The process would be bootstrapped by manually labeling the first k bug reports which are used to train an initial model (Lines 1–5). The classification model is then used to re-sort the unlabeled bug reports (Line 6). The next top p reports are presented for user feedback (Lines 7–8). We only repeat the refinement process after p new feedback are obtained. Then, the feedback is incorporated by learning a new discriminative model and applying it to the remaining unlabeled bug reports in the refinement loop (Lines 3–14). When the false positive rate goes too high, it is likely that there are no more true positives among the remaining uninvestigated bug reports. The user can choose to stop the refinement process (Lines 10–11).

With accumulated user feedback (Lines 8 and 13), the refinement process can incrementally improve the classification and ranking accuracy of the discriminative models so that true positives can be ranked higher.

Algorithm 1 Clone report refinement process

```
Input:  BR: Bug Reports
        k: Initial set of bug reports to be labeled
        p: Feedback pool size
Output: Re-ordered Bug Reports

1: Let BK = Select the first k bug reports
2: Label all bug reports in BK (manual)
3: Let FK = Features extracted from BK
4: Perform pre-processing on FK
5: Let M = Classification model created from FK
6: Refine BR using M
7: Let BP = Select the new top p unlabeled bug reports
8: Ask for user feedback on bug reports in BP
9: Let FPRate = Compute false positive rate
10: If FPRate is too high (based on user feedback)
11:     Stop
12: Else
13:     Set BK = BK ∪ BP
14:     Goto 3
```

6.5 Evaluation Criteria

In this section we define a suitable metric to measure the quality of the re-sorted bug reports to evaluate the effectiveness of our active refinement process.

Our refinement process is effective if it could re-sort the reports such that all reports corresponding to true positives are listed first. As an illustration, consider a scenario where our refinement process starts with a set of k labeled bug reports and there are m true positive reports among the remaining unlabeled reports. The ideal situation happens when all m other true positives are listed in the $(k+1)^{th}$ to $(k+m)^{th}$ positions after the refinement process. The worst case happens when the true positives are listed as the last m reports after refinement.

To measure the quality of the refinement process, we adapt a measure proposed in test case prioritization area—average percentage faults detected (APFD) [37]. There are a number of similarities between test case prioritization and our problem. In test case prioritization, test cases need to be sorted (*i.e.*, prioritized) in the order of their likelihood to reveal program failures. Also, there is a need to measure and compare the quality of different test case orderings.

In [37], a graph capturing the cumulative proportion of faults captured as more test cases are run is plotted. APFD, defined as the area under this curve, measures the *rate of fault detection*. In our work, we use the same concept and plot a graph capturing the cumulative proportion of true positives found as more anomaly reports are inspected by users. We refer to this graph as the *cumulative true positives curve*.

In the *cumulative true positives curve*, a larger area under the curve indicates that more true positives are found by developers early, which means that the refinement process effectively re-sorts the anomaly reports. Consider the sample graphs in Figure 6.7 and assume that there are five true positives among 10 reports. Each of the five increments in each of the two cumulative curves corresponds to when each of the five true positives is found. The left curve shows the cumulative of true positives that can be found using the original list. Using this list, the true positives

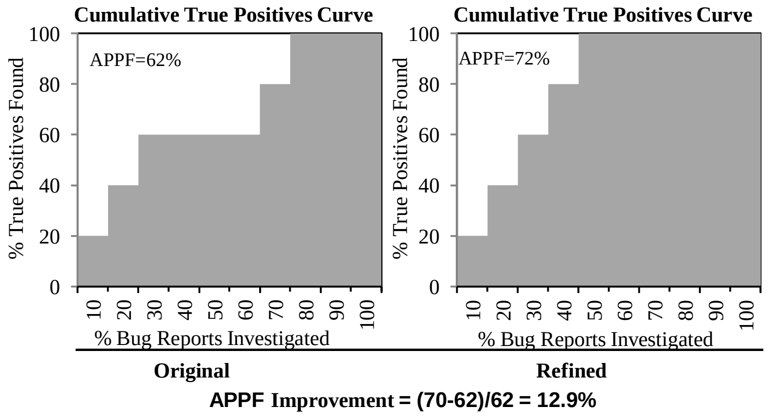


Figure 6.7: Computing and Comparing APPF

are found at positions 1, 2, 3, 7, and 8, hence the developers can find the five true positives by inspecting the first eight reports. The right curve shows the cumulative of true positives that can be found using the refined list produced by the refinement process. Using this list, the developers can find all five true positives by inspecting only the first five reports i.e., the true positives are found at positions 1, 2, 3, 4, and 5. Therefore, by performing the same number of inspections (which may correspond to the time budget of a developer in real-world situations), the developer could find more true positives using the refined report list than using the original list.

The idea of using APFD as the evaluation criteria for bug finding is also used by Kremenek and Engler [77]. Following the same idea, we define *average percentage true positives found* (APPF) as the area under the *cumulative true positives curve*. Our goal is to improve the APPF score which measures the rate of true positives found. We illustrate APPF improvement computation in Figure 6.7.

6.6 Empirical Evaluation

In this section, we describe our experimental settings, our evaluation results, and the threats to validity.

6.6.1 Settings and Results

1) Settings: We evaluate our approach on clone-based anomaly reports for three real programs written in different programming languages: the Linux kernel (C program), Eclipse and ArgoUML (Java programs). We analyze the reports generated by Jiang *et al.* [64]. We choose these reports as they contain a large number of false positives. There are more than 800 anomaly reports (*i.e.*, clone groups) for the Linux kernel, and only 57 of them are true positives or programming style issues. There are more than 400 anomaly reports for Eclipse, and only 38 of them are true positive. There are more than 50 anomaly reports for ArgoUML, and only 15 of them are true positive.

Finding a few true positives on the large number of false positives is a challenging task that would stress test the usability of our approach. Jiang et al. [64] have manually labeled all the reported inconsistent clone groups from the Linux kernel and Eclipse as either true positives or false positives. We manually label the reported inconsistent clone groups from ArgoUML. We use these clone groups and their labels to simulate initial and incremental user feedbacks as inputs to our refinement engine.

The tool of Jiang et al. [64] returns the list of anomalies in a particular order. We take the ordering returned by the tool and refine it. Following the steps in Section 6.4, we initially take the first k labeled clone groups. We set k to 50 for Eclipse since there is only one true positive among the first 50 bug reports. We also use $k = 50$ for the Linux kernel. Since there are only 50 inconsistent clone groups reported for ArgoUML, we set its k to 10. We set the feedback pool size (i.e., p) to 1. We thus iteratively refine the bug reports when each feedback is received. Inthis work, we repeat the refinement process until all anomaly reports are inspected.

2) Evaluation Results: We improve APPF by 11%, 87%, and 86% for the Linux kernel, Eclipse, and ArgoUML bug reports respectively. These measures mean that within a limited period of time, a developer may find more true positives in the

System	**Top-5 Re-orderings**
Linux	694 ↦ 18, 672 ↦ 64, 760 ↦ 131, 770 ↦ 179, 792 ↦ 206
Eclipse	373 ↦ 4, 348 ↦ 11, 394 ↦ 29, 388 ↦ 43, 370 ↦ 49
ArgoUML	40 ↦ 12, 35 ↦ 15, 34 ↦ 11, 29 ↦ 9, 23 ↦ 8

Table 6.2: Top-5 re-orderings. $x \mapsto y$ means that a report of a true positive at position x is reordered to position y.

Top	**Feature**	**Info. Gain**
1	extdef^P	0.015941
2	extdef_1^P	0.015941
3	program##extdefs^P	0.015941

Table 6.3: Top 3 features based on their Information Gain: Linux kernel. The ## symbol is the separator between two features for a pair feature set. The ^{P} superscript denotes a proportional feature.

Linux kernel, Eclipse, and ArgoUML. The improvement for the Linux kernel is not as great as Eclipse and ArgoUML. The bugs in the Linux kernel often involves identifier changes (e.g., variations in variable names, function names, type names, etc.) which are not well captured by our feature sets. Our feature sets are mostly based on syntactical node types which perform well in capturing the bugs in Eclipse and ArgoUML that often involve conditionals. In future, we plan to add more features to construct better discriminative models for Linux and more programs.

The top-5 successful re-orderings for the Linux kernel, Eclipse and ArgoUML are shown in Table 6.6.1. We highlight sample bugs that are successfully reordered. Figure 6.8 shows a buggy clone group in Linux that is reordered from position 694 to 18. The bug is related to an early unlock of a variable. Figure 6.9 shows a bug from Eclipse that is successfully reordered from position 373 to 4. The bug is similar to the bug in Figure 6.2; it misses a null-check in code fragment 2, and was reported to developers and fixed. For ArgoUML, a bug shown in Figure 6.10 is reordered from position 40 to 12. This bug is related to a missing validation before a variable is used in the next statement.

To further investigate which program elements (*i.e.*, features as described in Section 6.4.1) may be better bug indicators, we compute the *information gain* [95]

Code Fragment 1

```
File: linux-2.6.19/drivers/net/wireless/bcm43xx/bcm43xx_sysfs.c

347: struct bcm43xx_private *bcm = dev_to_bcm(dev);
 …
351: mutex_lock(&(bcm)->mutex);
352: switch (bcm43xx_current_phy(bcm)->type) {
353:   case BCM43xx_PHYTYPE_A:
 …
362:   default:
363:     assert(0);
364: }
365: mutex_unlock(&(bcm)->mutex);
```

Code Fragment 2

```
File: linux-2.6.19/drivers/net/wireless/bcm43xx/bcm43xx_wx.c

615: struct bcm43xx_private *bcm = bcm43xx_priv(net_dev);
 …
618: mutex_lock(&bcm->mutex);
619: mode = bcm43xx_current_radio(bcm)->interfmode;
620: mutex_unlock(&bcm->mutex);
621: switch (mode) {
622:   case BCM43xx_RADIO_INTERFMODE_NONE:
 …
632:   default:
633:     assert(0);
634: ..
```

Figure 6.8: Report of a true positive in Linux that is reordered from position 694 to 18

Code Fragment 1

```
File: eclipse-cvs20070108/org.eclipse.debug.core/core/org/
  eclipse/debug/internal/core/LaunchConfiguration.java

253: if (file != null) {
254:         // validate edit
255:         if (file.isReadOnly()) {
256:                 IStatus status =  ResourcesPlugin.
         getWorkspace().validateEdit(new IFile[] {file}, null);
257:                 if (!status.isOK()) {
258:                      throw new CoreException(status);
259: ......
```

Code Fragment 2

```
File: eclipse-cvs20070108/org.eclipse.debug.core/core/org/
  eclipse/debug/internal/core/LaunchConfigurationWorkingCopy.java

310: ......
311: // validate edit
312: if (file.isReadOnly()) {
313:         IStatus status = ResourcesPlugin.
     getWorkspace().validateEdit(new IFile[] {file}, null);
314: if (!status.isOK()) {
315:         throw new CoreException(status);
316: ......
```

Figure 6.9: Report of a true positive in Eclipse that is reordered from position 373 to 4

Code Fragment 1	Code Fragment2

File: /argouml/src/argouml-app/src/org/argouml/uml/diagram/UMLMutableGraphSupport.java
 	
331: if (edge instanceof CommentEdge) {
332:     …
333: } else if (Model.getFacade().isARelationship(edge)
334:          || Model.getFacade().isATransition(edge)
335:          || Model.getFacade().isAAssociationEnd(edge)){
336:     return Model.getUmlHelper().getDestination(edge);
337 } else if (Model.getFacade().isALink(edge)) {
338:     ..
339: }
	File: /argouml/src/argouml-app/src/org/argouml/uml/diagram/UMLMutableGraphSupport.java
	
	360: if (edge instanceof CommentEdge) {
	361: …
	362: } else if (Model.getFacade().isAAssociation(edge)) {
	363: List conns = new
	         ArrayList(Model.getFacade().getConnections(edge));
	364: return conns.get(1);
	365: } else if (Model.getFacade().isARelationship(edge)
	366: || Model.getFacade().isATransition(edge)
	367: || Model.getFacade().isAAssociationEnd(edge)){
	368: return Model.getUmlHelper().getDestination(edge);
	369 } else if (Model.getFacade().isALink(edge)) {
	370: ..
	371: }

Figure 6.10: Report of a true positive in ArgoUML that is reordered from position 40 to 12

Top	Feature	Info. Gain
1	BOOL_OR_TK^P	0.01898
2	conditional_or_expression ## conditional_or_expression^P	0.01898
3	BOOL_OR_TK^B	0.01898

Table 6.4: Top 3 features based on their Information Gain: Eclipse. The ^{B} superscript denotes a basic feature while ^{P} superscript denotes a proportional feature

Top	Feature	Info. Gain
1	local_variable_declaration_statement^B	0.145772
2	variable_initializer^B	0.145772
3	block_statement ## local_variable_declaration_statement^B	0.145772

Table 6.5: Top 3 features based on their Information Gain: ArgoUML

of each feature in Linux and Eclipse bug reports. Information gain is frequently used to find important features that differentiate two contrasting datasets (*i.e.*, in our case, true positives and false positives), e.g., [112].

The top 3 features for Linux kernel, Eclipse, and ArgoUML are shown in Table 6.6.1, 6.4, and 6.5. We notice that the individual features have low information gain. Thus, a single feature may not be able to distinguish true positives from false positives, composing them into a discriminative model is however more effective in improving true positives rate.

Based on the list of the top features, one could intuitively infer that if a clone from Eclipse involves inconsistent changes related to conditionals, it is more likely to be buggy. For ArgoUML, the inconsistent changes that involve variable declaration and initialization are more likely to be buggy, e.g., a declared variable being used without further validation (e.g., null checking), inappropriate type of a variable, etc. Discriminative features using *information gain* could mean that the features are either highly related to buggy clones or highly related to non-buggy clones. In the Linux kernel, if a clone involves inconsistent changes related to global definitions (*e.g.*, extdef), it is more likely not a bug. Overall, the features used in our approach can be useful to discriminate the anomaly reports that contain bugs from those that are false positives.

6.6.2 Threats to Validity

Threat to construct validity corresponds to the suitableness of our evaluation metric. In this study we have adapted a measure commonly used in test case prioritization, which also needs to re-sorts (*i.e.*, prioritize) test cases. Their goal is to find an optimal ordering of test cases that would identify the failures early. They measure the quality of an ordering using *average percentage faults detected* (APFD). We propose a similar measure referred to as *average percentage of true positives found* (APPF). Like APFD that measures the rate of fault detection, APPF measures the rate of true positives found. We believe that this measure is relevant in measuring the performance of a refinement framework. A higher APPF score indicates that within the same period of time a developer can find more true positives.

Threat to internal validity corresponds to the ability of our experiments to correlate the independent and dependent variables. The threat could be manifested due to experimental or human errors. The labels of the bug reports are manually decided by Jiang et al. [64] which might be prone to errors. Still, the authors have taken some precautions to prevent these from happenning – at least two people are assigned to label each inconsistent clone group; for any discrepancy, a third person would break the tie.

Threat to external validity corresponds to the generalizability of our result. We have performed a study on three large real systems that are written in two most popular programming languages: C and Java. Although these help, there is still a threat to external validity. In the future, we plan to investigate more systems written in various programming languages.

6.7 Conclusion

Code clones have been widely studied in the literature. Various techniques have been proposed to recover clones from a code base. One important usage of clones is to find bugs by detecting inconsistencies among members of the same clone group. These correspond to bugs that might arise due to inconsistent updates among parallel code fragments or violations of a common programming practice.

Past techniques, e.g., [64], have demonstrated the ability of clone-based bug detection tools to detect true positives in large systems. However, the tools also return a substantial number of false positives. This could affect the usability of such a system as a developer could spend a lot of time in performing a debugging activity. The system might be futile in the end, if the reported anomaly turns out to be a false alarm.

Our work addresses this issue by proposing an approach to automatically refine bug reports by incorporating user feedback. Rather than having a *static* sorted list of bug reports, our bug report list is *dynamic*. As a user investigates the top few bug reports and feedback to the system, the system automatically re-sorts the remaining uninvestigated bug reports, and thus *refines* it accordingly. This refinement process is performed multiple times when more feedback is available. For each refinement, we perform feature extraction, pre-processing (feature selection and dataset re-balancing), and discriminative model learning.

We evaluate the quality of a list of ordered bug reports using *average percentage of true positives found* (APPF) that measures the rate of true positives. We evaluate our refinement process on three sets of clone-based anomaly reports from three large real programs: the Linux kernel (C), Eclipse, and ArgoUML (Java). A clone-based anomaly detection tool is used to produce the clone anomaly reports of the programs. The results show that, compared to the original ordering of bug reports, our approach can improve APPF by 11%, 87%, and 86% for Linux kernel, Eclipse, and ArgoUML respectively.

Chapter 7

Conclusion

7.1 Summary of Contributions

We summarize our contributions as follows:

1. Many fault localization techniques have been proposed to facilitate debugging activities. Most of them attempt to pinpoint the location of bugs (i.e., localize bugs) based on a set of failing and correct executions and expect developers to investigate a certain number of located program elements to find faults. These techniques thus assume that faults are localizable, i.e., only one or a few lines of code that are close to one another are responsible for each fault. However, in reality, are faults localizable?

   This dissertation has investigated hundreds of real faults in several software systems, and has found that a number of faults that often involve a few lines of code or a few methods. However, a substantial number of faults (i.e., 30% of the faults) still manifest in a single line of code and more than 60% of the faults are localizable within one method.

2. To localize faults in programs, spectrum-based fault localization is a promising approach to automatically locate root causes of failures quickly. Two well-known spectrum-based fault localization techniques, Tarantula and Ochiai,

measure how likely a program element is a root cause of failures based on profiles of correct and failed program executions. These techniques are conceptually similar to association measures that have been proposed in statistics, data mining, and have been utilized to quantify the relationship strength between two variables of interest (e.g., the use of a medicine and the rate of recovery).

This dissertation has viewed fault localization as a measurement of the relationship strength between the execution of program elements and program failures. This dissertation has investigated the effectiveness of 40 association measures from the literature in localizing faults for single-bug and multiple-bug programs. This dissertation has found that there is no best single measure for all cases. Klosgen and Ochiai outperform other measures in localizing faults in single-bug programs. On average, Added Value could localize the faults in multiple-bug programs with the smallest percentage of code inspected, whereas a number of other measures have similar performance. As developers need to inspect more elements in multiple-bug programs, the accuracy of the measures in localizing multi-bug programs is expected to be lower than localizing faults in single-bug programs, which provokes future research.

3. For various bugs, the best techniques to localize the bugs may differ due to the characteristics of the buggy programs and their program spectra. This dissertation has leveraged the diversity of existing spectrum-based fault localization techniques to better localize bugs using data fusion methods. Our proposed approach consists of three steps: score normalization, technique selection, and data fusion. This dissertation has investigated two score normalization methods, two technique selection methods, and five data fusion methods resulting in twenty variants of Fusion Localizer. Our approach is bug specific in which the set of techniques to be fused are adaptively selected

for each buggy program based on its spectra. Also, it requires no training data, i.e., execution traces of the past buggy programs. Our evaluation have demonstrated that our approach can significantly improve the effectiveness of existing state-of-the-art fault localization techniques. Compared to these state-of-the-art techniques, the best variants of Fusion Localizer can statistically significantly reduce the amount of code to be inspected to find all bugs, increase the proportion of bugs localized when developers only inspect the top 10% most suspicious program elements by more than 10%, and increase the number of bugs that can be successfully localized when developers only inspect up to 10 program blocks by more than 20%.

4. To complement spectrum-based fault localization for cases in which test cases or program spectra are not available to be analyzed to locate faults in programs, this dissertation have proposed an approach to re-rank anomaly reports produced by a clone-based anomaly detection technique. A clone-based anomaly detection tool extracts clone groups, finds a piece of code in the group that are different from the rest, and flags it as an anomaly. Although clone-based anomaly detection can find real bugs on large systems, the number of false positives is often very high. The excessive number of false positives could easily impede broad adoption of clone-based bug detection approaches.

   This dissertation has improved the true positive rate found when a developer analyzes clone anomaly reports. The idea is to control the number of anomaly reports a user can see at a time and actively incorporate user feedback to continually refine the anomaly reports. Our system first presents the top few anomaly reports from a list of reports generated by a tool in its default ordering. Users then either accept or reject each of the reports. Based on the feedback, our system automatically and iteratively refines a classification model for anomalies and reorders the rest of the reports. The goal is to present the true positives to the users earlier than the default ordering. The rationale of the idea is based on our observation that false positives among the inconsistent clone groups could share common features (in terms of code structure, programming patterns, etc.), and these features can be learned from the incremental user feedback.

This dissertation has evaluated the proposed refinement process on three sets of clone-based anomaly reports from three large real programs: the Linux Kernel (C), Eclipse, and ArgoUML (Java), extracted by a clone-based anomaly detection tool. As compared to the original ordering of bug reports, our approach can improve the rate of true positives found (i.e., true positives are found faster) by 11%, 87%, and 86% for Linux kernel, Eclipse, and ArgoUML, respectively.

7.2 Future Work

Based on our findings in this dissertation, several issues remains worthy for investigation. This dissertation has shown that a substantial proportion of faults can be localizable within a few line of code, while others spread across a larger number of lines of code. Localizing the bugs that are not localizable within a few lines of code is challenging. Studying the relationships among the faulty lines could be useful to better localize bugs that manifest in multiple lines of code. Designing techniques that can help developers locate faults that are not localizable within a few lines of code will benefit software debugging task.

Assuming that faults are localizable within a few line of code, we have proposed association measures to be used in locating faults. Our empirical study have demonstrated that the accuracy of these association measures as well as two well-known spectrum-based fault localization techniques in localizing multiple bugs in a program is lower than their accuracy in localizing a single bug in a program. The program spectra collected from a program that contains multiple bugs could not distinguish the effect of one bug from the other. Future research is required to improve the ability of fault localization approaches in localizing multiple bugs in programs. One possibility is to rerun a fault localization approach every time a bug is found and analyze the new ranked list to find the next bug.

In addition, fault localization techniques that we have proposed assume that developers could locate the bugs by inspecting the list of most suspicious code recommended by the fault localization techniques. As pointed out by Parnin and Orso [106], showing the right contexts for developers to understand the root causes of a bug would be useful to guide developers to find the bugs. Therefore, we plan to extend our approach by providing additional information to developers to guide them to locate bugs in programs. Information about how faulty codes relate to one another and the way developers locate the bugs (e.g., information or logic used by developers to locate bugs) could be helpful in providing useful guidance for developers to locate the faulty lines. A user study will be needed to investigate to what extent this additional information helps developers in finding faulty code.

Many software systems developed today are written in multiple programming languages, e.g., Java, JavaScript, etc. In this work, we only evaluate the effectiveness of our approach in localizing faults for software written in one language, i.e., C or Java. In the future, we want to consider software that are written in multiple programming languages. We also plan to integrate our fault localization techniques into popular IDEs and debugging tools such as Eclipse and Visual Studio.Net for wider adoption.

The effectiveness of our approach to refine anomaly reports is affected by the features being used to discriminate the true positive from false positive reports. In order to be more effective in refining the anomaly reports, we need to incorporate more features that can capture wider range of bugs. Also, it would be advantageous to extend our refinement engine by providing developers with information about how likely additional true positives remain in the unchecked anomaly reports. By providing this information, developers could have better idea when to stop investigating the reports and they could save more time and effort.

Our active refinement approach can help in refining anomaly reports produced by clone detection tools. It would be interesting to extend our approach to refine other anomaly reports such as those produced by various bug finding tools, e.g., FindBugs, PMD, JLint, etc. Hence, we could potentially improve the adoption of these tools by reducing the rate of false positives.

Bibliography

[1] Ant. http://svn.apache.org/.

[2] Lucene-JIRA. https://issues.apache.org/jira/browse/LUCENE.

[3] R. Abreu. Spectrum-based fault localization in embedded software., 2009.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evalution on spectrum-based fault localization. *The Journal of System and Software*, 82:1780–1792, 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. van Gemund. Spectrum-Based Multiple Fault Localization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Auckland, New Zealand, 2009.

[6] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Mutation Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION'07)*, 2007.

[7] C. Aggarwal and P. S. Yu. A new framework for itemset generation. In *Symposium on Principles of Database Systems (PODS'98)*, 1998.

[8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, 1994.

[9] A. Agresti. *An Introduction to Categorical Data Analysis*. John Wiley & Sons, 1996.

[10] R. B. Altman and T. E. Klein. Challenges for biomedical informatics and pharmacogenomics. *Annu. Rev. Pharmacol. Toxicol.*, 42:113–133, 2002.

[11] M. Anderberg. *Clustering Analysis for Applications*. London, Academic Press, 1973.

[12] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis(ISSTA'10)*, 2010.

[13] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, 2010.

[14] J. A. Aslam and M. Montague. Models for metasearch. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'01)*, pages 276–284. ACM, 2001.

[15] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[16] G. Baah, A. Podgurski, and M.J.Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE'11)*, pages 146–156, 2011.

[17] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Transactions on Software Engineering (TSE'07)*, 2007.

[18] A. Bandyopadhyay and S. Ghosh. On the effectiveness of the tarantula fault localization technique for different fault classes. In *IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE'11)*, pages 317–324, 2011.

[19] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS©: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.

[20] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, Boston, 2nd edition, 1990.

[21] B.Jiang, W.K.Chan, and T.H.Tse. On practical adequate test suites for integrated test case prioritization and fault localization. In *Proceedings the 11th International Conference on Quality Software (QSIC'11)*, pages 21–30, 2011.

[22] A. Bosch, A. Zisserman, and X. Munoz. Scene classification using a hybrid generative/discriminative approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[23] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, volume 37, pages 211–230. ACM, 2002.

[24] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket analysis. In *Proceedings of 1997 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 255–264, Tucson, AZ, May 1997.

[25] J. Carey, N. Gross, M. Stepanek, and O. Port. Software hell. *Business Week*, pages 391–411, 1999.

[26] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the 18th International Symposium on Software Testing and Analysis(ISSTA'09)*, 2009.

[27] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering (ICSE'09)*, pages 34–44. IEEE Computer Society, May 2009.

[28] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Notices*, volume 37, pages 258–269. ACM, 2002.

[29] P. Clark and R. Boswell. Rule induction with cn2: Some recent improvements. In *Machine Learning - EWSL-91*, page 151163, 1991.

[30] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 342–351, St. Louis, Missouri, May. 2005.

[31] C.Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. In *Proc. of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC / SIGSOFT FSE '05)*, Lisbon, Portugal, Sep. 2005.

[32] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[33] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07)*, pages 433–436, 2007.

[34] L. R. Dice. Measures of the amount of ecologic association between species. In *Ecology*, 1945.

[35] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[36] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[37] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering (TSE'02)*, 28:159–182, 2002.

[38] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[39] A. Feldman and A. van Gemund. A two-step hierarchical algorithm for model-based diagnosis. In *Proceedings of the 21st National Conference on On Artificial Intelligence*, pages 827–833, Boston, Massachusetts, 2006. AAAI Press.

[40] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[41] E. A. Fox, M. P. Koushik, J. Shaw, R. Modlin, D. Rao, et al. Combining evidence from multiple searches. In *The first text retrieval conference (TREC-1)*, pages 319–328. US Department of Commerce, National Institute of Standards and Technology, 1993.

[42] E. A. Fox and J. A. Shaw. Combination of multiple searches. *NIST SPECIAL PUBLICATION SP*, pages 243–243, 1994.

[43] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, 2010.

[44] L. Geng and H. Hamilton. Interestingness measures for data mining: A survey. In *ACM Computing Surveys*, 2006.

[45] C. Gini. Variability and mutability, contribution to the study of statistical distributions and relations. *Studi Economico-Giuricici della R. Universita de Cagliari*, 3(part 2)(i-iii):3–159, 1912.

[46] L. Goodman and W. Kruskal. Measures of association for cross classifications. *Journal of the American Statistical Association*, 49:732–764, 1954.

[47] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pages 40–49, 2012.

[48] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, pages 263–272, 2005.

[49] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41:4–12, 2002.

[50] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2006.

[51] D. J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[52] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[53] Y. Hayase, Y. Y. Lee, and K. Inoue. A criterion for filtering code clone related bugs. In *Proceedings of the 2008 workshop on Defects in large software systems (DEFECTS'08)*, 2008.

[54] J. F. Healey. *Statistics: A Tool for Social Research*. Wadsworth Publishing, 8th edition, 2008.

[55] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, pages 41–50, 2008.

[56] K. Herzig and A. Zeller. Untangling changes. September 2011.

[57] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance*, 20(6):435–461, 2008.

[58] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 191–200, 1994.

[59] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications workshop on eclipse technology eXchange*, 2007.

[60] S. Jarzabek and S. Li. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference (ESEC/FSE'03)*, 2003.

[61] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *International Symposium on Software Testing and Analysis (ISSTA'08)*, 2008.

[62] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[63] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 184–193. ACM, 2007.

[64] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/SIGSOFT FSE'07)*, 2007.

[65] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, 2005.

[66] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 467–477, Orlando, Florida, May. 2002.

[67] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 16–26, 2007.

[68] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009.

[69] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE'02)*, 2002.

[70] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings the 13th Working Conference on Reverse Engineering (WCRE'06))*, 2006.

[71] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 351–360, 2011.

[72] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, 2005.

[73] S. Kim and M. D. Ernst. Which warnings should i fix first? In *ESEC-FSE*, 2007.

[74] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[75] W. Klosgen. Explora: A multipattern and multistrategy discovery assistant. In *Advances in Knowledge Discovery and Data Mining*, 1996.

[76] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Static Analysis*, pages 40–56. Springer, 2001.

[77] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximation. In *SAS*, pages 295–315, 2003.

[78] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering(WCRE'07)*, 2007.

[79] J.-G. Lee, J. Han, X.Li, and H. Cheng. Mining discriminative patterns for classifying trajectories on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):713–726, 2011.

[80] M. Leszak, D. E. Perry, and D. Stoll. A case study in root cause defect analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'00)*, pages 428–437, 2000.

[81] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[82] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, UC Berkeley, 2002.

[83] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 141–154, San Diego, CA, June 2003.

[84] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings ACM SIGPLAN 2005 International Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[85] D. Lillis, F. Toolan, R. Collier, and J. Dunnion. Probfuse: a probabilistic approach to data fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 139–146. ACM, 2006.

[86] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[87] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 37–48. ACM, 2006.

[88] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[89] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis.

[90] B. Martin. Instance-based learning : Nearest neighbor with generalization. In *Dissertation, University of Waikato, Hamilton, New Zealand*, 1995.

[91] W. Mayer, R. Abreu, M. Stumptner, and A. van Gemund. Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis (DX)*, 2009.

[92] W. Mayer and M. Stumptner. Abstract interpretation of programs for model-based debugging. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 471–476, 2007.

[93] W. Mayer and M. Stumptner. Model-Based Debugging – State of the Art And Future Challenges. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174(4), 2007.

[94] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 128–137, 2008.

[95] T. M. Mitchell. *Machine Learning*. McGraw-Hill, March 1997.

[96] A. Moore. An introductory tutorial on KD-Trees. Technical Report 209, Computer Laboratory, University of Cambridge, 1991. Extract from Andrew Moore PhD Dissertation: Efficient Memory-based Learning for Robot Control.

[97] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[98] N.DiGiuseppe and J.A.Jones. Semantic fault diagnosis: Automatic natural-language fault descriptions. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, page 23, 2012.

[99] R. H. Netzer and B. P. Miller. *Improving the accuracy of data race detection*, volume 26. ACM, 1991.

[100] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, pages 263–272. IEEE, 2011.

[101] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *Proceedings the 24th IEEE/ACM International Conference on Automated Software Engineering(ASE'09)*, 2009.

[102] A. Ochiai. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. In *Bull Jnp Soc Sci Fish*, 1957.

[103] M. Ohsaki, S. Kitaguchi, K. Okamoto, H. Yokoi, and T. Yamaguchi. Evaluation of rule interestingness measures with a clinical dataset on hepatitis. In *Proc. of the 8th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2004)*, pages 362–373, 2004.

[104] T. J. Ostrand and E. J. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4), 1984.

[105] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[106] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 199–209, 2011.

[107] D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Software Engineering-ESEC'93*, pages 48–67, 1993.

[108] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In *Knowledge Discovery in Databases*, 1991.

[109] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 465–477, 2003.

[110] A. Podgurski and L. Pierce. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM'93)*, 1993.

[111] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings the 30th Annual International Symposium on Computer Architecture*, pages 110–121. IEEE, 2003.

[112] J. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[113] R. R. Nuray and F. Can. Automatic ranking of information retrieval systems using data fusion. *Information Processing & Management*, 42(3):595–614, 2006.

[114] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[115] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*, pages 43–52. ACM, 2011.

[116] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings. 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 141–154, 2003.

[117] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. Springer, 1997.

[118] J. Rogers and T. T. Tanimoto. A computer program for classying plants. In *Science*, 1960.

[119] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008.

[120] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[121] S. Salzberg. A nearest hyperrectangle learning method. *Machine Learning*, 1991.

[122] R. Santelices, J. Jones, Y. Yu, and M. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009.

[123] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[124] E. Shortliffe and B. Buchanan. A model of inexact reasoning in medicine. *Mathematical Biosciences*, 23:351379, 1975.

[125] Siemens, M. Harrold, and G. Rothermel. *Aristotle Analysis System – Siemens Programs, HR Variants*. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[126] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, pages 50–59. IEEE, 2012.

[127] P. Smyth and R. Goodman. An information theoretic approach to rule induction from databases. *IEEE Trans. Knowledge and Data Eng.*, 4(4):301–316, 1992.

[128] R. R. Sokal and C. Michener. A statistical method for evaluating systematic relationships. In *Univ Kans Sci Bull*, 1958.

[129] T. Sorensen. 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. In *Vidensk Selsk Biol Skr*, 1948.

[130] C. D. Sterling and R. A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience (SP&E)*, 37(10):1061–1086, August 2007. John Wiley & Sons, Inc.

[131] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. In *Proceeding the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, 2008.

[132] P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, pages 32–41, Edmonton, Canada, July 2002.

[133] S. Tan. An effective refinement strategy for knn text classifier. *Expert Syst. Appl.*, 2006.

[134] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[135] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, 2004.

[136] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings the 22nd International Conference on Program Comprehension (ICPC'14)*, 2014.

[137] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau. Search-based fault localization. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Engineering (ASE'11)*, pages 556–559, 2011.

[138] D. Wettschereck and G. Dietterich. An experimental comparison of the nearest-neighbour and nearest-hyperrectangle algorithms. *Machine Learning*, 1994.

[139] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin available: http://www.jstor.org/stable/3001968*, 1:80–83, 1945.

[140] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2nd edition, 2005.

[141] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*.

[142] S. Wu. *Data Fusion in Information Retrieval*, volume 13. Springer, Heidelberg, 2012.

[143] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[144] X. Xie, F. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *Search Based Software Engineering*, pages 224–238. Springer, 2013.

[145] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Notices*, volume 40, pages 1–14. ACM, 2005.

[146] R. B. Yates and B. R. Neto. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[147] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Search Based Software Engineering*, pages 244–258. Springer, 2012.

[148] G. U. Yule. On the association of attributes in statistics. *Philosophical Transcriptions of the Royal Society*, A194:257–319, 1900.

[149] G. U. Yule. On the methods of measuring association between two attributes. *Journal of the Royal Statistical Society*, 75:579–642, 1912.

[150] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of Software Engineering (ASE'02)*, pages 1–10, 2002.

[151] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2 edition, 2009.

[152] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, 2006.

[153] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *ACM SIGPLAN Notices*, volume 41, pages 169–180. ACM, 2006.

[154] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings the 34th International Conference on Software Engineering (ICSE'12)*, pages 14–24, 2012.

[155] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE'09)*, 2009.