DISSERTATION

MITIGATING THE EFFECT OF COINCIDENTAL CORRECTNESS IN SPECTRUM BASED FAULT LOCALIZATION

Submitted by

Aritra Bandyopadhyay

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2013

Doctoral Committee:

Advisor: Sudipto Ghosh

James M. Bieman
Robert B. France
Michelle Mills Strout
Daniel Turk

ABSTRACT

MITIGATING THE EFFECT OF COINCIDENTAL CORRECTNESS IN SPECTRUM BASED FAULT LOCALIZATION

Coincidentally correct test cases are those that execute faulty program statements but do not result in failures. The presence of such test cases in a test suite reduces the effectiveness of spectrum-based fault localization approaches, such as Ochiai and Tarantula, which localize faulty statements by calculating a suspiciousness score for every program statement from test coverage information.

The goal of this dissertation is to improve the understanding of how the presence of coincidentally correct test cases impacts the effectiveness of spectrum-based fault localization approaches and to develop a family of approaches that improve fault localization effectiveness by mitigating the effect of coincidentally correct test cases.

Each approach (1) classifies coincidentally correct test cases using test coverage information, and (2) recalculates a suspiciousness score for every program statement using the classification information. We developed classification approaches using test coverage metrics at different levels of granularity, such as statement, branch, and function. We developed a new approach for ranking program statements using suspiciousness scores calculated based on the heuristic that the statements covered by more failing and coincidentally correct test cases are more suspicious. We extended the family of fault localization approaches to support multiple faults.

We developed an approach to incorporate tester feedback to mitigate the effect of coincidental correctness. The approach analyzes tester feedback to determine a lower bound for the number of coincidentally correct test cases present in a test suite. The lower bound is also used to determine when classification of coincidentally correct test cases can improve fault localization effectiveness.

We evaluated the fault localization effectiveness of our approaches and studied how the effectiveness changes for varying characteristics of test suites, such as size, test suite type (e.g., random, coverage adequate), and the percentage of passing test cases that are coincidentally correct. Our key findings are summarized as follows. Mitigating the effect of coincidentally correct test cases improved fault localization effectiveness. The extent of the improvement increased with an increase in the percentage of passing test cases that were coincidentally correct, although no improvement was observed when most passing test cases in a test suite were coincidentally correct. When random test suites were used to localize faults, a coarse-grained coverage spectrum, such as function coverage, resulted in better classification than fine-grained coverage spectra, such as statement and branch coverage. Utilizing tester feedback improved the precision of classification. Mitigating the effect of coincidental correctness in the presence of two faults improved the effectiveness for both faults simultaneously for most faulty programs. Faulty statements that were harder to reach and that affected fewer program statements resulted in fewer coincidentally correct test cases and were more effectively localized.

ACKNOWLEDGMENTS

I would not be writing this dissertation, if it was not for the untiring support of my Ph.D. advisor, Dr. Sudipto Ghosh. From the conception of this work to its execution, refinement, and presentation, he spent countless hours to help me make my pursuit of the Ph.D. a successful and fulfilling journey. I particularly thank him for his efforts in meticulously checking my drafts and providing feedback, not just to improve the drafts but to improve my ability to think and articulate ideas in general.

As a graduate student, I went through periods of productivity and motivation, as well as periods of withdrawal and frustration. My advisor supported me not only in my brightest hours by inspiring me to achieve more and keeping me from falling into complacency, but also in my darkest hours by being most patient and encouraging me to never stop trying. At times he was the kindest mentor, while at others, he was the toughest reviewer of my work. I did not always fully understand his intent, but in retrospect, I realized that I had always received from him the most appropriate guidance, given the circumstances. Because of his tutelage, I have grown into a better researcher, a better thinker, and a more self-reliant person.

I thank my Ph.D. committee members Dr. James Bieman, Dr. Robert France, Dr. Michelle Strout, and Dr. Dan Turk for carefully reading my dissertation and for providing valuable feedback. Their evaluation of my work during a number of other prior talks, such as the Ph.D. preliminary examination, has improved my understanding and helped me refine my work as well. In particular, I received valuable guidance from Dr. Bieman for correctly designing my empirical studies.

I thank the staff of the computer science department, especially Kim and Sharon, for their help in completing paperwork and for explaining official procedures. They are some of the nicest people I have ever met; their welcoming gestures have brightened my days.

Words will never be strong enough to express my gratitude for my parents, Mrs. Rita Bandyopadhyay and Dr. Bimal Bandyopadhyay, who dedicated their lives for my happiness.

Everything I achieved was possible because of their unconditional love and limitless sacrifice. My dear sister, Romani, always stood by my side.

My friends were my sources of strength. Sajib and I perhaps share the strongest form of friendship and camaraderie. We trust each other with our deepest fears, doubts, and confusion, and always received from reach other the courage and strength to face them. Pramita always treated my concerns as sincerely as her own. Meghan and I regularly discussed our goals and progress. She untiringly encouraged me to persevere and never gave up believing in me. I also thank Apurba, Anant, Claudia, David, Helen, John, Kunal, Nand, Neelam, and Prathamesh for their incredible friendship.

I am grateful to the organization, Soka Gakkai International (SGI), which promotes the practice of Nichiren Buddhism worldwide as a means for the empowerment of individuals. The practice has brought about many positive changes in my life. It has helped me overcome my negativities and taught me a life-changing lesson – to always seek strength and happiness within, instead of blaming the environment. I also thank the members of SGI for supporting me to embrace this practice and to understand its true essence.

DEDICATION

To my mother, Mrs. Rita Bandyopadhyay, my father, Dr. Bimal Bandyopadhyay, and my sister, Mrs. Romani Bandyopadhyay.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Context

Manual program debugging involves searching for faulty statements by inspecting the execution states of the program at different points. Pan and Spafford [38] described the tasks developers repeatedly perform while debugging. Developers first determine the parts of the program involved in the failure, such as execution paths. They use their knowledge of the program to make a hypothesis about potentially faulty statements. They set breakpoints at the potentially faulty statements and suspend the program execution for verifying the program state. These tasks are time consuming and can be expedited by using automated support for narrowing down the search domain of faulty statements. Fault localization refers to the task of automatically identifying potentially faulty program statements.

A class of automated fault localization approaches, referred to as spectrum-based fault localization approaches [2, 9, 21, 29, 41] analyze the spectra of passing and failing test cases to produce a report of possible fault locations in a faulty program. The spectrum of a test case contains information associated with the execution of the test case, such as the set of statements or branches covered by the test case. Some approaches generate a suspiciousness score for every program statement based on the heuristic that statements executed more often by failing test cases than passing test cases are highly suspicious. Two such approaches are Ochiai [2] and Tarantula [29]. The Ochiai suspiciousness score of a statement, s, is calculated using Equation 1.1. The terms, $passed(s)$, $failed(s)$, and $totalFailed$, denote the number of passing test cases that execute s, the number of failing test cases that execute s, and the total number of failing test cases, respectively.

$$Ochiai(s) = \frac{failed(s)}{\sqrt{totalFailed \times (passed(s) + failed(s))}}. \tag{1.1}$$

The Tarantula suspiciousness score of a statement, s, is calculated using Equation 1.2. The term, $totalPassed$, denotes the total number of passing test cases.

$$Tarantula(s) = \frac{\frac{failed(s)}{totalFailed}}{\frac{failed(s)}{totalFailed} + \frac{passed(s)}{totalPassed}} \tag{1.2}$$

The Ochiai and Tarantula suspiciousness scores can also be calculated for other types of program elements, such as functions, branches, and def-use pairs [43], in terms of the numbers of passing and failing test cases that cover the program elements.

To localize the faulty statement, all the program statements are ranked in decreasing order of their suspiciousness scores. A tester inspects each statement in the ranked order and determines whether the statement is faulty or not. The tester may need to perform additional tasks, such as setting breakpoints and analyzing the execution states. The tester repeats the process until the faulty statement is identified. When the rank of the actual faulty statement is high, the effort needed to locate it is less. One fault localization approach is considered to be more effective than another if the former ranks the faulty statement higher than the latter.

1.2 Problem

The presence of coincidentally correct test cases degrades the effectiveness of spectrum-based fault localization approaches, such as Ochiai and Tarantula. A coincidentally correct test case is defined to be one that passes in spite of executing the faulty statement. In order to cause a failure, a test case must execute the faulty statement, result in a local failure state, and propagate the local failure state to the output [42]. A coincidentally correct test case executes the faulty statement but either does not result in a local failure state or results in a local failure state that does not propagate to the output. The Ochiai and Tarantula approaches assign a higher suspiciousness score to a statement if the statement is executed by fewer passing test cases and more failing test cases. Coincidentally correct test cases add to the number of passing test cases that execute the faulty statement and lower the suspiciousness score of the faulty statement. Wang et al. [48] reported that when the

percentage of coincidentally correct test cases increased from 0% to 60%, the percentage of faults effectively localized in the *space* [1] program by using the Tarantula approach decreased from 90% to 25%. In this study, a fault was considered to be effectively localized if the faulty statement was in the top 5% of the list of statements ranked in decreasing order of their suspiciousness scores.

There are two challenges in mitigating the effect of coincidentally correct test cases in spectrum-based fault localization. The first challenge is to classify every passing test case as coincidentally correct or not coincidentally correct. Since the fault location is not known, the classification must be based on other information, such as the spectrum of the test cases. For spectrum-based classification, the research challenge is to identify what characteristics of a spectrum indicate coincidental correctness.

The second challenge is to effectively use the result of the classification in the calculation of the suspiciousness scores of statements. Not all approaches for using the classification results may be effective. For example, one can remove the test cases classified as coincidentally correct and calculate the suspiciousness scores using the remaining test cases [37]. However, removal of test cases causes loss of information by reducing the coverage of the test suite used for fault localization. For example, suppose that every coincidentally correct test case is correctly classified and removed. As a result, for the faulty statement and any other non-faulty statement that is executed by all failing test cases and only coincidentally correct test cases, the Ochiai and Tarantula suspiciousness scores become 1. For these statements, in Equations 1.1 and 1.2, $failed(s)$ equals $totalFailed$ and $passed(s)$ equals zero after the classified test cases are removed. Thus, many non-faulty statements share the same suspiciousness score as the faulty statement, thereby reducing the rank of the faulty statement and the effectiveness of the fault localization approach.

Masri et al. [37] proposed an approach to mitigate the effect of coincidental correctness. The approach uses test coverage at different levels of granularity, such as as basic block coverage, branch coverage, and def-use coverage, to classify coincidentally correct test cases using

the k-means clustering approach. The classified test cases are removed and suspiciousness scores are recalculated using the remaining test cases.

Masri et al.'s approach leaves new opportunities for improvement and poses new research challenges. Although the approach used three types of coverage spectra for classification, further investigation is needed to understand the differences in the recall and precision of classification using coverage spectra at different levels of granularity. The approach is used to perform classification even when classification cannot improve fault localization effectiveness, such as when there are few coincidentally correct test cases or none. The approach removes test cases, and as described previously, the removal of test cases can lead to reduced fault localization effectiveness. It is worthwhile to investigate whether the classified test cases, instead of being discarded, can be used to localize faulty statements more effectively. Masri et al.'s approach assumes a single fault, but for most real world programs, coincidental correctness needs to be addressed for multiple faults.

Additionally, any approach that mitigates the effect of coincidental correctness needs to be evaluated by varying the factors that have been shown to affect the effectiveness of spectrum-based fault localization approaches. For example, prior studies [7] showed that the type of test suites used affects how effectively a fault is localized. Thus, approaches for addressing coincidentally correct test cases need to be evaluated for commonly used types of test suites, such as random and coverage adequate test suites. Other characteristics of test suites, such as size and percentage of passing test cases that are coincidentally correct, can affect the effectiveness of the approaches. Therefore, the effect of these characteristics needs to be studied as well.

1.3 Approach and Contributions

We present a family of fault localization approaches to mitigate the effect of coincidentally correct test cases in spectrum-based fault localization. Each approach has two components: (1) an approach for classifying every passing test case into two classes, coincidentally correct

and not coincidentally correct, and (2) an approach for recalculating suspiciousness scores for statements using the test cases and the information obtained from the classification.

Our classification approaches are based on the heuristic that the passing test cases that cover similar program elements as the failing test cases are likely to be coincidentally correct. We expect the heuristic to hold because both the coincidentally correct test cases and the failing test cases need to execute the program elements on the path to the faulty statement. As a result, both the types of test cases are likely to cover many similar program elements. Additionally, in our exploratory studies described in Chapter 3, we also empirically observed that most coincidentally correct test cases covered similar program elements as the failing test cases. We use coverage spectra using program elements at three different levels of granularity: statement, branch, and function. Each level offers unique advantages. A fine-grained coverage spectrum can improve classification by magnifying the important differences between a passing test case and a coincidentally correct test case. A coarse-grained coverage spectrum can result in an accurate classification by abstracting away the unimportant differences between a passing and a coincidentally correct test case.

The classification approaches use k-means clustering to group the passing test cases into two clusters. The cluster of test cases that are more similar to the failing test cases is identified as the cluster of coincidentally correct test cases. We use two similarity measures: Euclidean distance and average suspiciousness score of covered program elements. Our goal is to measure the extent to which each passing test case covers similar program elements as the failing test cases. Euclidean distance was used by Renieris et al. [41] to measure the similarity of covered program elements between a passing and a failing test case. The average suspiciousness score of the program elements covered by a passing test case is high if the test case covers program elements that have high suspiciousness scores. Program elements with high suspiciousness scores are also covered by the failing test cases. Thus, average suspiciousness score can be used as a measure of similarity of covered elements between a passing and a failing test case. Masri et al. [37] also used average suspiciousness score for classifying coincidentally correct test cases.

We developed an approach to recalculate suspiciousness scores based on the Jaccard similarity metric [24]. The approach assumes the presence of a single fault and uses the heuristic that among the statements that are executed by all failing test cases, a statement is more suspicious if it is executed by more coincidentally correct test cases and fewer passing test cases that are not coincidentally correct. This heuristic is based on the observation that the actual faulty statement is executed by all the failing test cases as well as by every coincidentally correct test case, under the single fault assumption.

In some situations, it is not required to address coincidentally correct test cases because the faulty statement obtains the highest suspiciousness score even when coincidental correctness is not addressed. This happens when the test suite either does not contain any coincidentally correct test case or contains too few coincidentally correct test cases to reduce the suspiciousness score of the faulty statement below that of other non-faulty statements. Forcing classification of test cases in such situations may reduce the fault localization effectiveness. We present a method to determine whether or not there is an opportunity to improve effectiveness by mitigating the effect of coincidentally correct test cases and apply our approach only if such an opportunity exists.

We extend our family of approaches to support the localization of multiple faults by modifying (1) our Jaccard similarity based approach for calculating suspiciousness scores, and (2) our method for determining whether or not there is an opportunity for improving fault localization effectiveness by mitigating the effect of coincidentally correct test cases. These modifications are required because in the presence of multiple faults, a faulty statement may not be executed by all failing test cases.

We implemented a framework of fault localization approaches for mitigating the effect of coincidental correctness. The framework provides a concrete implementation of our family of fault localization approaches for localizing faults in C programs, and is also easily extensible for researchers that need to support: (1) new approaches for classifying test cases, (2) new approaches for calculating suspiciousness scores, (3) new types of coverage spectra, and (4) other programming languages.

We performed a set of studies to evaluate our family of fault localization approaches. We studied how the recall and precision of the classification approach varied with the choice of the level of granularity of the coverage spectrum. We evaluated the fault localization effectiveness of our approach for calculating suspiciousness scores by assuming perfect classification and also in combination with the classification approaches. We studied how the effectiveness of our family of approaches varied with different characteristics of test suites, such as size, test suite type (e.g., random, statement adequate, and branch adequate), and percentage of passing test cases that are coincidentally correct.

Chapter 2

Related Work

In this chapter, we describe existing spectrum-based fault localization approaches. We summarize the approaches that mitigate the effect of coincidental correctness in spectrum-based fault localization. Because mitigating the effect requires classifying and selecting test cases for fault localization, we also present existing approaches that investigate the effect of different test selection, generation, and reduction strategies on the effectiveness of fault localization. Because we use tester feedback, we describe existing algorithmic debugging approaches that also utilize tester feedback to guide the search for the faulty statement. We survey applications of clustering approaches to classify test cases. We also discuss existing research on localizing multiple faults.

2.1 Spectrum Based Fault Localization

Spectrum-based fault localization approaches collect information about the program's execution with passing and failing test cases, and analyze the information to produce a report of possible fault locations. Two attributes characterize the approaches and determine their effectiveness: (1) abstractions of program executions, and (2) heuristics about fault location. An abstraction of a program execution contains information about the execution, such as statement coverage, basic block coverage, and dynamic slices. Based on the abstraction, a heuristic defines rules for identifying potentially faulty program elements. Each approach uses a heuristic to calculate a suspiciousness score for every program element (e.g., statement, branch, or predicate), or identifies a set of suspicious program elements.

2.1.1 Abstractions Used in Fault Localization Approaches

Statement coverage spectrum, branch coverage spectrum, and def-use pair coverage spectrum: The statement coverage spectrum [29] of a test case is the set of statements executed at least once by the test case. The branch coverage spectrum and the def-use pair coverage spectrum [43] record the branches and def-use pairs covered by a test case at least once.

Binary coverage spectra and permutation spectrum: The binary coverage spectrum [41] of a test case is the set of basic blocks executed at least once by the test case. The permutation spectrum [41] of a test case is the sequence of basic blocks executed by the test case, sorted in the order of their execution counts.

Predicate profile: The predicate profile [34] of a test case records what predicates hold true during the execution of the test case. The predicates are specified at different locations in a program, such as branches, function calls, and assignment statements. The predicates are checked by instrumenting the corresponding program locations. To achieve scalability, each predicate is randomly sampled during a test execution by executing the corresponding instrumentation site with some probability.

Path Profile: The path profile [15] of a test case records which intra-procedural, acyclic path segments are observed or executed. A path is said to be observed by a test case when the test case visits the start node of the path. An observed path may or may not be executed. A path is said to be executed by a test case when the test case executes the whole path.

Program slice: Some approaches [5, 21] use dynamic program slices as abstractions. A program slice with respect to a point of interest consists of those parts of the program that either affect or are affected by the values computed at that point [47]. The point of interest is called the slicing criterion and is described by a pair (program point, set of variables). For example, a program point may be a statement and the corresponding set of variables is any subset of the set of all variables defined or used at the statement. A backward slice with respect to a criterion contains the program parts that affect the variables in the criterion. A forward slice contains the parts affected by those variables.

A program slice can be static or dynamic. The former uses only static information about the program and does not make any assumption about the program input. The latter is defined in the context of a program execution with a fixed input. A static slicing criterion consists of a static statement and a set of variables defined or used at the statement. A static slice with respect to the criterion is computed by tracing the transitive dependencies from the point specified by the criterion. A dynamic slicing criterion specifies a program input, occurrence of a statement in the execution and a set of variables defined or used by the statement. Unlike static slicing, dynamic slicing only traces those transitive dependencies that occur in a specific execution with a test input.

Program slicing may use one or both types of dependencies: data and control. Data slices consider only data dependencies, control slices consider only control dependencies, and full slices consider both types of dependencies. Horwitz et al. [23] defined callstack sensitive slices. A callstack sensitive slice of a failing test case reduces the size of dynamic backward slice of the failing test case by leveraging the calls that are active in the stack at the point when the program fails.

Call sequence sets: A call sequence set [16] of an object in an execution of a test case is the set of sequences of method calls invoked on, or received by, the object. A call sequence set may contain only calls received or calls invoked. The calls invoked by an object are used to analyze how the object uses other objects. The calls received by an object are used to analyze how the object is used by other objects. The call sequence sets of all the objects of a class can be aggregated to produce a class-level call sequence set for the class.

The type of spectrum influences how effectively faults are localized and what types of faults are localized more effectively than others. Santelices et al. [43] showed that using def-use coverage spectrum results in more effective fault localization than branch coverage spectrum, which results in more effective fault localization than using statement coverage spectrum. As will be discussed in Section 2.1.2, fault localization heuristics determine what program elements appear in the spectra of the failing test cases more often than that of the passing test cases. The heuristics identify such elements to be suspicious. Thus, which

elements are identified as suspicious depends on how the spectra of the failing and the passing test cases differ. Different types of spectra have different abilities to exhibit differences between executions of test cases.

2.1.2 Fault Localization Heuristics

Tarantula: The heuristic used in the Tarantula approach states that the statements executed primarily by the passing test cases are the least suspicious, and the ones executed primarily by the failing test cases are the most suspicious. For every statement, s, two values are calculated: $color(s)$ and $brightness(s)$. The terms, $\%passed(s)$ and $\%failed(s)$, denote the percentage of passing test cases and the percentage of failing test cases that execute s, respectively. Equations 2.1 and 2.2 show how color and brightness values are calculated.

$$color(s) = \text{low color (red)} + \frac{\%passed(s)}{\%passed(s) + \%failed(s)} \times \text{color range} \tag{2.1}$$

$$brightness(s) = max(\%passed(s), \%failed(s)) \tag{2.2}$$

Higher the color value of the statement, the more suspicious is the statement. Two statements may share the same color value because the value of the ratio in the expression for $color(s)$ may be the same for them even though they have different values for $\%passed(s)$ and $\%failed(s)$. In such situations, the $brightness(s)$ component is used to resolve the tie by comparing the absolute values of $\%passed(s)$ or $\%failed(s)$, whichever is larger.

Ochiai: The heuristic used in the Ochiai approach [2] is the same as that used in the Tarantula approach. However, a different expression is used to calculate the suspiciousness scores, which is shown in Equation 2.3. The terms, $passed(s)$, $failed(s)$, and $totalFailed$, denote number of passing test cases that execute the statement, s, the number of failing test cases that execute s, and the total number of failing test cases, respectively.

$$Ochiai(s) = \frac{failed(s)}{\sqrt{totalFailed \times (passed(s) + failed(s))}} \tag{2.3}$$

The Ochiai approach has been used with several types of abstractions, such as statement coverage spectrum [2], branch coverage spectrum [43], and def-use coverage spectrum [43].

Jaccard: The heuristic used by the Jaccard approach [2] is the same as that of the Ochiai and Tarantula approaches, but is mathematically expressed using the Jaccard similarity coefficient for binary variables, as shown in Equation 2.4. This equation is different from our expression for calculating suspiciousness scores presented in Chapter 4, which is based on the Jaccard metric of the similarity of two sets.

$$Jaccard(s) = \frac{failed(s)}{totalFailed + passed(s)} \tag{2.4}$$

The Jaccard heuristic has been used with several abstractions, such as statement coverage spectrum [2] and unserialized interleaving patterns of shared memory access [39].

.

Heuristic for call sequences: Dallmeier et al. [16] proposed the following heuristics to assign suspiciousness scores to individual call sequences, assuming that fault is localized using a single failing and multiple passing test runs:

1. If a sequence appears in a failing run, it is more suspicious if it is present in fewer passing runs. The presence of the sequence in the failing run is the suspected cause of the failure.

2. If a sequence is missing in a failing run, it is more suspicious if it is present in more passing runs. The absence of the sequence in the failing run is the suspected cause of the failure.

Equation 2.5 expresses the heuristics mathematically.

$$w(p) = \begin{cases} \frac{k(p)}{n} & \text{if } p \in P(r_0) \\ 1 - \frac{k(p)}{n} & \text{if } p \notin P(r_0) \end{cases} \tag{2.5}$$

In the equation, $w(p)$ denotes the weight of sequence, p. $P(r_0)$ denotes the sequence set of a class obtained from the single failing run, r_0. The term, $k(p)$, denotes the number of passing test cases where the sequence p appears in the sequence set of the class. The degree

of suspiciousness of a class is the sum of the weights of all the sequences in the sequence set of the class.

Union and intersection model: The union and the intersection model [41] assume the presence of a single failing test case and multiple passing test cases. The union model uses the heuristic that the statements that are uniquely covered by the failing test cases are suspicious. Thus, if f denotes the set of statements covered by the failing test case and $\bigcup s$ denotes the set of statements covered by the passing test cases, the statements in the set $f - \bigcup s$ are considered suspicious. The intersection model [41] uses the heuristic that the statements "whose absence is discriminant" in the failing test case are suspicious. Thus the statements in the set $\bigcap s - f$ are considered suspicious, where $\bigcap s$ denotes the statements that are covered by every passing test case.

Nearest neighbor model: The nearest neighbor model [41] uses the heuristic that the basic blocks that are covered by the failing test case but not covered by a passing test case are suspicious. Additionally, using the passing test case that is most similar to the failing test case results in the most accurate set of suspicious basic blocks. Similarity is measured by different coverage spectra, such as the binary coverage spectrum and the permutation spectrum.

Test case grouping and weighting: The heuristic used in this approach [18] is an extension of the heuristic of the Tarantula, Ochiai, and Jaccard approaches. In the Tarantula, Jaccard, and Ochiai equations, every additional passing or failing test case that executes a statement increases the value of the term $passed(s)$ or the term $failed(s)$, respectively, by a constant amount. In other words, the contribution of every passing or failing test case to the calculation of the suspiciousness score of a statement is constant. However, this approach is based on the heuristic that in the calculation of the suspiciousness score of a statement, the contribution of every passing or failing test case is larger than that of the next passing or failing test case that executes the statement. Test cases are distributed in groups and the groups are assigned decreasing weights. The suspiciousness score of a statement is calculated

in terms of the sum of weights of the failing test cases and that of the passing test cases that execute the statement.

χ^2 metric: The heuristic used in this approach [52] is that the degree of suspiciousness of a statement is proportional to the strength of the relationship between the two variables: test outcome (pass/fail) and test coverage of a statement (covered/not covered). For each statement, the approach constructs a 2x2 contingency table with these two variables and measures the strength of the relationship of the variables by calculating the χ^2 metric, which is used as the suspiciousness score of the statement.

Dicing: Agrawal et al. [5] proposed the heuristic that the statements contained in a *dice* are suspicious. They defined a *dice* as the set difference of the dynamic backward slice of a passing test execution from that of a failing test execution. Pan et al. [38] proposed a family of heuristics, including the intersection and the union model, based on backward dynamic slices of failed and passed test executions.

Failure inducing chops: Gupta et al. [21] used the heuristic that the statements contained in a failure inducing chop are suspicious. Given a failing test case, a failure inducing chop is constructed as the intersection of the forward dynamic slice of minimum failure inducing input and the backward dynamic slice of the failed output.

Statistical bug isolation : Liblit et al. [33, 34] defined a heuristic to estimate the suspiciousness score of every predicate P. The suspiciounsess score estimates "How much does P being true increase the probability of failure over simply reaching the line where P is sampled". The probability of failure when P is true is denoted by $Failure(P)$. The probability of failure when P is just sampled is denoted by $Context(P)$. The suspiciousness score, denoted by $Increase(P)$, is defined by the expression $Failure(P) - Context(P)$.

$Failure(P)$ is measured by the expression $\frac{F(P)}{S(P)+F(P)}$, where $F(P)$ and $S(P)$ respectively denote the number of failing test cases where P was observed to be true, and the number of passing test cases where P was observed to be true. $Context(P)$ is measured by the expression $\frac{F(P\,observed)}{S(P\,observed)+F(P\,observed)}$, where $F(P\,observed)$ and $S(P\,observed)$ denote the number of

failing test cases where P was sampled, and the number of successful test cases where P was sampled, respectively.

Tables 2.1 and 2.2 summarize the fault localization approaches along with their abstractions and heuristics.

2.2 Effect of Test Case Generation, Selection, and Reduction Strategies

Artzi et al. [7] compared the fault localization effectiveness resulting from test suites generated using different strategies. They used different variants of concolic testing [44] to generate test cases that are similar. They used two types of similarities: path constraint based similarity and input similarity. The path constraint based similarity of two test runs is measured by the number of conditional statements in those runs that evaluate to the same value. The input similarity between two runs is measured by the number of inputs that are identical in those runs. The authors performed an empirical study to show that path-constraint-similar test cases result in higher fault localization effectiveness.

Baudry et al. [13] proposed an approach to optimize a test suite for more effective fault localization. They state that the more statements a test suite is able to distinguish, the higher is the effectiveness of fault localization. Two statements are distinguished by a test suite if the statements are executed by different subsets of the test suite. Their approach to improve a test suite uses the bacteriologic algorithm, which mutates the test cases in the test suite. The mutation operator applied to a test case randomly replaces one command in the test case by another command. After the mutations are performed, the algorithm calculates the fitness value of the modified test suite. The fitness value is proportional to the degree to which the test suite is able to distinguish between the statements.

Yu et al. [54] investigated the impact of various test suite reduction strategies on the effectiveness of fault localization. They used two types of test suite reduction strategies: (1) statement-based reduction, which selects from a test suite a reduced test suite that covers the

Table 2.1: Fault Localization Approaches – 1

Approach	**Abstraction**	**Heuristic**
Dicing [5]	Dynamic slice	Statements that are in the dynamic slice of the failing test cases but are not in the dynamic slice of a passing test case are suspicious.
Tarantula [29]	Statement coverage spectrum	Statements executed by more failing test cases and fewer passing test cases are more suspicious. The *color* and *brightness* values in Equation 2.1 and 2.2 are used as measures of how frequently a statement is executed by failing test cases compared to by passing test cases.
Nearest Neighbor Model [41]	Binary coverage spectrum and permutation spectrum	Basic blocks covered by the failing test case but not covered by the passing test case that is closest to the failing test case are suspicious.
Statistical bug isolation [33, 34]	Predicate profile	Predicates that are evaluated to be true primarily in failing test cases are suspicious. The expression $Increase(p)$ is used to measure how frequently failing test cases evaluate a predicate p to be true.
Failure inducing chops [21]	Dynamic slice	Statements that are in the forward dynamic slice of a failure inducing input in a failing test case but are not in the backward dynamic slice of the output of the failing test case are suspicious.
Dallmeier et al.[16]	Method sequence sets	A method sequence that is covered by a failing test case and few passing test case is suspicious. A method sequence that is not covered by a failing test case and is covered by many passing test case is suspicious. Equation 2.5 is used to evaluate the suspiciousness of a method sequence. A class that has more suspicious method sequences is suspicious.
Ochiai [2]	Statement coverage spectrum	Same as Tarantula but suspiciousness scores are calculated using Equation 2.3.
Jaccard [2]	Statement coverage spectrum	Same as Tarantula but suspiciousness scores are calculated using Equation 2.4.
Gupta et al. [56]	Data, control, full, and relevant slices	Statements in the data/control/full slice of a failing test case are suspicious.

Table 2.2: Fault Localization Approaches – 2

Approach	**Abstraction**	**Heuristic**
Crosstab-based approach [52]	Statement coverage spectrum	Statements that have a higher value of χ^2 metric of the relationship of coverage by test cases and test outcome are more suspicious.
Fault localization with multiple coverage types [43]	Statement coverage spectrum, branch coverage spectrum, and def-use pair coverage spectrum	Statements/branches/def-use pairs that are covered primarily by failing test cases are suspicious. The Ochiai equation is used to measure the suspiciousness scores of statement/branch/def-use pairs.
Callstack sensitive slicing [23]	Callstack sensitive slice	Statements contained in the callstack sensitive slice are suspicious.
Test case grouping and weighting [18]	Statement coverage spectrum	Statements executed by more failing test cases and fewer passing test cases are more suspicious. Additionally, each passing test case's contribution is more than that of the next passing test case.
Falcon [39]	Unserialized interleaving pattern of shared memory access	Interleaving patterns that are covered primarily by failing test cases are suspicious. The Jaccard equation is used to measure how frequently an interleaving pattern is covered by failing test cases compared to passing test cases.
Holmes [15]	Path profile	Path segments that are primarily covered by failing test cases are suspicious. The expression $Increase(p)$ is used to measure how frequently failing test cases cover a path segment.

same statements as the original suite, and (2) vector-based reduction, where the reduced test suite covers the same set of statement vectors as the original test suite. A statement vector is the set of statements executed by one test case. They applied the reduction strategy one by one on the passing test cases, failing test cases, and the entire test suite. Their results showed that test suite reduction degrades the effectiveness of fault localization in general. Statement-based reduction significantly affects the effectiveness, while vector-based reduction has negligible effect.

Jeffrey and Gupta [27] used multiple coverage criteria to reduce a test suite. From a test suite, they first select a test case t_1 that satisfies a requirement of a coverage criterion C_1.

Among the test cases that become redundant with respect to C_1 due to the selection of t_1, they retain a test case t_2 that satisfies the requirement of a different criterion C_2. The result of their evaluation shows that the selective retention strategy leads to a smaller reduction of fault localization effectiveness compared to the strategies that use a single coverage criterion.

2.3 Coincidental Correctness

Abreu et al. [2] studied the effect of coincidentally correct test cases on the effectiveness of fault localization. They defined the observation quality of a test suite as the percentage $\frac{\#\text{failing tests}}{\#\text{test cases that execute the fault}} \times 100$. They showed that the effectiveness of fault localization increases with an increase in observation quality.

Wang et al. [48] used context patterns to mitigate the effect of coincidental correctness. A context pattern corresponding to a type of fault describes the data and control flow patterns that result in the propagation of a local failure state to the output. The authors specify context patterns for common fault types. The approach matches the execution of each test case with all the context patterns. The coverage of a test case is expressed by using a list of pairs *(statement, matched context pattern)*. The approach uses Tarantula to calculate the suspiciousness score of each *(statement, matched context pattern)* pair. The approach requires that the types of faults be known in advance, which may not be true in practice. It also requires specification of the context patterns of the faults types, which is an additional burden on the tester. The approach is computationally expensive because it matches the context pattern with every test case for every statement.

Masri et al. [37] proposed three classes of approaches for mitigating the effect of coincidentally correct test cases in spectrum-based fault localization. We discuss two of these classes that were demonstrated by the authors to be more effective than the third. The two classes are: (1) approaches based on calculating the likelihood of coincidental correctness, and (2) approaches based on k-means clustering. Each class contains three approaches, each of which uses a different type of program element (e.g., statements, branches, and def-use pairs).

Approaches based on calculating the likelihood of coincidental correctness use Equation 2.6 to calculate a value for every passing test case, p, representing the likelihood that p is coincidentally correct.

$$CC_{Masri}(p) = \frac{\sum_{s \in S_p} Tarantula(s)}{|S_p|} + \frac{|S_p|}{|S|} \times 100 \tag{2.6}$$

$Tarantula(s)$ denotes the suspiciousness score of the program element, s. S_p is the set of program elements that p executes and S is the set of all program elements. Both S_p and S include only those program elements that have a Tarantula suspiciousness score between 0.5 and 1.

The CC_{Masri} measure has the following two properties:

- If a passing test case, p, primarily executes statements with high suspiciousness scores, the first term is high, and the measure has a high value.
- If p achieves a higher statement coverage, the measure has a higher value.

A percentage of the passing test cases having the highest value of the CC_{Masri} measure is classified as coincidentally correct. Masri et al. empirically showed that selecting 60% of the passing test cases with the highest CC_{Masri} values results in the most accurate classification. The classified test cases are removed and the suspiciousness scores of the program statements are recalculated using the remaining test cases.

The k-means clustering based approaches first identify program elements that are "likely to be correlated with coincidentally correct tests". Masri et al. stated that such an element "occurs in all failing runs and in a non-zero but not excessively large percentage of passing runs". For identifying these program elements, Masri et al. experimented with different threshold values for the percentage of passing test cases and concluded that the threshold value of 70% yields best results. The approaches use the heuristic that coincidentally correct test cases execute more such elements compared to other passing test cases.

Each test case is associated with a vector of 0s and 1s. Each element in the vector corresponds to a program element previously identified. If a test case covers the program element, then the corresponding element in the vector of the test case is 1, and 0 otherwise.

Once the test vectors are formed, k-means clustering of the vectors is performed using the Euclidean distance between the vectors to form two clusters of the passing test cases. The initial centroids of the clusters are chosen to be the vectors that are separated by the maximum distance among all pairs of vectors. The cluster containing the coincidentally correct test cases is determined as follows. First, the Tarantula suspiciousness score, $Tarantula(s)$ for each program element, s, is calculated. Then, for each cluster, the average Tarantula score of the elements covered by the test cases in the cluster is calculated. The cluster with the higher average is selected to be the one containing coincidentally correct test cases.

As a result of the above steps, the classification approaches are expected to classify passing test cases that are similar to all the failing test cases as coincidentally correct. Passing test cases that are dissimilar to the failing test cases are expected to be classified as not coincidentally correct.

The classified test cases are removed and the suspiciousness scores of the program statements are recalculated using the remaining test cases.

2.4 Algorithmic Debugging

Algorithmic debugging approaches [14, 45, 46] use tester feedback to reduce the search space of a faulty program element. These approaches are applied to functional programs. Given a program execution, these approaches construct an execution tree. Each node in an execution tree represents a computation in the corresponding execution. A node's children represent the sub-computations of the node's computation.

A tester using an algorithmic debugging approach traverses the execution tree and answers a question about the correctness of each node. A question is typically of the form whether or not the run-time values of variables represent a correct program state. Based on the answer, a set of nodes is pruned from the tree. For example, if a tester reports that a computation is correct, all nodes in the sub-tree rooted at the computation's node are pruned. Advanced techniques use improved searching and pruning strategies that exploit the structure of the execution tree, such as balancing.

The main disadvantages of algorithmic debugging are: (1) the execution tree requires a large memory space, (2) the questions presented to the tester by the approach can be complex and large in number, and (3) faults can only be localized at the level of functions.

2.5 Clustering Test Cases

Clustering techniques have been applied to classify test cases for different goals. Leon et al. [32] applied clustering to select test cases from a large test suite such that the fault detection ability of the selected suite is as close to the original test suite as possible. They used a representation of each test case based on the execution profile of the test case and clustered using the Euclidean distance of the representations. They empirically demonstrated that the clustering based technique results in at least as effective test suites as the selection techniques that aim to maximize the coverage.

Yoo et al. [53] proposed an approach that improves human judgment based test prioritization approaches by clustering test cases. Test prioritization approaches that use human judgment require a domain expert to perform pair-wise comparison of all the test cases in a test suite. However, for large test suites, pair-wise comparison of all the test cases by a domain expert can be expensive and even infeasible. Yoo et al.'s approach reduces the number of comparisons performed by a domain expert without reducing the effectiveness of the prioritized test suites. Test cases are grouped into multiple clusters using the hierarchical clustering approach. The domain expert performs a pair-wise comparison of the test cases inside each cluster. From each cluster, a test case is chosen to represent the cluster. The domain expert performs a pair-wise comparison of the representative test cases to obtain a prioritized test suite.

Jones et al. [28] and Liu et al. [35] proposed approaches to classify failing test cases such that the failing test cases resulting from the same fault belong to the same class. For each failing test case, their approaches calculate the ranking of statements or predicates according to suspiciousness scores computed by a fault localization approach using that failing test case. Failing test cases that result in similar rankings are considered to be resulting from the same

fault. Failing test cases are classified by the agglomerative clustering [26] approach using the following steps. Initially, for each failing test case, there is one cluster containing only that failing test case. At each step, a pair of clusters are selected that are most similar to each other and are merged together. The merging is repeated until there are two clusters that have a similarity above a certain threshold.

2.6 Localizing Multiple Faults

Researchers have studied the effectiveness of fault localization approaches for programs containing multiple faults. DiGiuseppe et al. [17] evaluated the effect of the number of faults on the effectiveness of the Ochiai approach. They reported that the effectiveness is not significantly affected by the number of faults. They also observed that for certain benchmarks, effectiveness improved with an increase in the number of faults.

Spectrum-based fault localization approaches, such as Ochiai and Tarantula, use the heuristic that the statements that are executed by more failing test cases are more suspicious. However, this heuristic cannot be directly applied to localize multiple faults because a fault may only be executed by a subset of the failing test cases. These failing test cases are typically the ones that fail because of the fault.

Researchers [29, 34] have addressed the above problem by assuming that in the presence of multiple faults, only a single fault is localized at a time. Each time a fault is localized, the fault is fixed and fault localization is performed again to localize the remaining faults. This process is repeated until no failure is observed.

The iterative process of localizing individual faults, removing faults, and performing fault localization may require many test executions, resulting in poor scalability. Jones et al. [28] proposed an approach that first identifies specialized test suites for localizing each fault and then localizes the faults in parallel using the specialized test suites. Ideally, a specialized test suite for a fault should contain the passing test cases and only those test cases that failed due to the fault. However, because the faults are not known, test cases that failed due to the

same fault are identified by grouping similar failing test cases together. An agglomerative clustering [26] technique is used to derive the grouping of the failing test cases.

Abreu et al. [3] proposed an approach that identifies and ranks groups of statements instead of individual statements. Each group is a potential multiple fault candidate. The approach first uses coverage information of test cases and applies model-based reasoning to identify multiple fault candidates that are consistent with the test outcomes. The multiple fault candidates are ranked according to the extent each candidate explains the test outcomes. The extent to which the multiple fault candidates explain the observations is measured using Bayes' theorem.

None of the above approaches mitigate the effect of coincidental correctness in the presence of multiple faults. We extend our family of fault localization approaches to mitigate the effect of coincidental correctness in the presence of multiple faults.

Chapter 3

Exploratory Approaches and Studies

In this chapter, we present exploratory approaches and studies performed to build the foundation of our family of fault localization approaches that are described in Chapter 4. We begin by presenting in Section 3.1 a study of the effectiveness of the Tarantula approach for different fault classes [10]. The fault classes are obtained from three properties that determine how a fault results in a failure. This study not only corroborates that coincidentally correct test cases degrade fault localization, but also provides a number of insights that we use later to explain the effectiveness of our family of fault localization approaches in Chapter 4.

In Section 3.2, we present a proximity-based weighting approach [11] that quantifies the importance of a passing test case for effective fault localization. For each passing test case, a measure of the proximity of the passing test case from the failing test cases is used to calculate a weight for the passing test case. The measure represents the extent to which a passing test case covers program elements similar to the failing test cases. Such a notion of proximity or similarity of two test cases is also used for test case classification, as described in Chapter 1. The weights calculated for the test cases, instead of the numbers, are used to calculate suspiciousness scores. Although this approach does not directly target coincidental correctness, it led us to recognize the need for classifying coincidentally correct test cases.

In Section 3.3, we present our first fault localization approach that mitigates the effect of coincidental correctness [12]. It measures for every passing test case the likelihood that the test case is coincidentally correct based on our work on the proximity-based weighting of test cases. The approach also uses tester feedback to estimate a lower bound for the number of coincidentally correct test cases. The estimation of the lower bound improves existing approaches [37] that mitigate the effect of coincidental correctness by ensuring that one never classifies more coincidentally correct test cases than actually present. This ap-

proach improved our understanding of the appropriate use of tester feedback for addressing coincidental correctness. Based on the insight gained, we improve the use of tester feedback in our family of approaches presented in Chapter 4.

3.1 Fault Localization Effectiveness for Different Fault Classes

We present a study of the fault localization effectiveness of the Tarantula approach for different fault classes. Although fault localization effectiveness is known to depend on the fault being localized, there is a lack of systematic evaluations that investigate the effect of fault classes on the effectiveness of fault localization. To our knowledge, this is the first study that treated fault class as an independent variable to evaluate the fault localization effectiveness.

To perform such a study, classifying the faults in a manner that is relevant to fault localization is required. We develop a classification scheme based on the following three fault properties that affect how faults cause failures [42]: (1) accessibility, (2) original state failure condition, and (3) impact. Accessibility of a fault addresses how hard it is for a test to execute the fault. Original state failure condition is the condition that must be satisfied in order to raise a local failure state upon the execution of the faulty statement. Impact is concerned about the fraction of the program that is affected by the execution of the faulty statement. For the original state failure condition, we use the classification of faults by Richardson et al. [42]. We present our own fault classifications based on accessibility and impact. We classify faults in the Siemens suite and then study the fault localization effectiveness of Tarantula for the different classes.

3.1.1 Analysis of Fault Properties

We illustrate with examples how variations in the three properties can cause corresponding variations in the effectiveness of the Tarantula approach.

3.1.1.1 Accessibility

Accessibility measures how hard it is for a test execution to access a faulty statement. Harder to access faulty statements will be executed by fewer test cases. All failings test cases must execute the faulty statement in order to cause failures. However, a harder to access faulty statement will be executed by fewer passing test cases. Thus, faults that are harder to access will result in a fewer coincidentally correct test cases. As a result, these faults will have a lower value of $\%passed(s)$ and a higher suspiciousness score. We use the following example to illustrate the effect of accessibility.

Table 3.1: Examples of Faults with Differing Accessibility

Original Program	Faulty $Version_1$	Faulty $Version_2$

`1 int a,b,c;`
`2 int r=0;`
`3 read(a,b,c);`
`4 if (a >= b){`
`5  r = a+b;`
`6  if (b > c){`
`7  r = r+10;`
`8  }`
`9 }`
`10 print(r);`
	`1 int a,b,c;`
	`2 int r=0;`
	`3 read(a,b,c);`
	`4 if (a >= b){`
	`5  r = a-b;`
	`6  if (b > c){`
	`7  r = r+10;`
	`8  }`
	`9 }`
	`10 print(r);`
		`1 int a,b,c;`
		`2 int r=0;`
		`3 read(a,b,c);`
		`4 if (a >= b){`
		`5  r = a+b;`
		`6  if (b > c){`
		`7  r=r-10;`
		`8  }`
		`9 }`
		`10 print(r);`

Table 3.1 shows a program and its two faulty versions. The fault in each version is shown in bold. The fault in $Version_1$ is easier to execute than the one in $Version_2$ because the former is guarded by only one conditional statement, while the latter is guarded by two.

To illustrate the application of Tarantula on the two faulty versions, we consider 6 tuples that provide input values for (a, b, c): t_1 = (1, 2, 3), t_2 = (2, 1, 3), t_3 = (2, 0, 3), t_4 = (4, 3, 0), t_5 = (4, 3, 1), and t_6 = (4, 0, 1). Table 3.2 shows the suspiciousness score for each statement in $Version_1$ along with information about which test case executed the statement (denoted by a $\surd$ mark), and the result of each test case (P = Pass, F = Fail).

The statements are arranged in decreasing order of the suspiciousness score. Table 3.3 shows the same information for $Version_2$.

Table 3.2: Suspiciousness Scores of Statements in $Version_1$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	*susp*
r=r+10				√	√		1
r=a-b		√	√	√	√	√	0.6
if(b>c)		√	√	√	√	√	0.6
int a,b,c	√	√	√	√	√	√	0.5
int r=0	√	√	√	√	√	√	0.5
read(a,b,c)	√	√	√	√	√	√	0.5
if(a>=b)	√	√	√	√	√	√	0.5
print(r)	√	√	√	√	√	√	0.5
Result	P	F	P	F	F	P	

Table 3.3: Suspiciousness Scores of Statements in $Version_2$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	*susp*
r=r-10				√	√		1
r=a+b		√	√	√	√	√	0.57
if(b>c)		√	√	√	√	√	0.57
int a,b,c	√	√	√	√	√	√	0.5
int r=0	√	√	√	√	√	√	0.5
read(a,b,c)	√	√	√	√	√	√	0.5
if(a>=b)	√	√	√	√	√	√	0.5
print(r)	√	√	√	√	√	√	0.5
Result	P	P	P	F	F	P	

The suspiciousness score assigned to the faulty statement in $Version_1$ is 0.6, while the score of the faulty statement in $Version_2$ is 1. Locating the fault in $Version_1$ requires us to examine 2 statements, while locating the fault in $Version_2$ requires us to examine only one statement. The orignial state failure condition for the fault in $Version_1$ is is `a+b` $\neq$ `a-b`, which is `true`, assuming non-zero values for `b`. The original state failure condition for the fault in $Version_2$ is `r+10` $\neq$ `r-10`, which is also `true` regardless of the value of `r`. However, because the fault in $Version_1$ is more easily accessible than that in $Version_2$, the former results in more coincidentally correct test cases, which causes the $Version_1$ fault to have a lower suspiciousness score than the $Version_2$ fault. This results in a lower rank for

the $Version_1$ fault because the relative suspiciousness of the remaining statements of the program is same for both the versions.

In general, for any two faulty versions v_1 and v_2, if the fault in v_1 is more easily accessible than the one in v_2, the probability that an arbitrary test case executes the fault in v_1 is greater than that for v_2. Also, suppose that the faults in v_1 and v_2 have the same original failure condition. Thus, the likelihood that a test case will pass after executing the fault in v_1 is same as that for v_2. With these assumptions, for an arbitrary test suite, the fault in v_1 is more likely to result in more coincidentally correct test cases than the fault in v_2. As a result, the suspiciousness score of the fault in v_1 is likely to be lower than that of the fault in v_2. If the relative suspiciousness scores of the non-faulty statements remain the same in both versions, the fault in v_1 will rank lower than the fault in v_2.

3.1.1.2 Original State Failure Condition

Richardson et al. [42] proposed the RELAY model of fault detection, which provides a framework for analyzing how a fault causes a failure. In this model, the original failure condition of a faulty statement is defined as the condition to be satisfied for an execution of the faulty statement to result in a local failure. To cause a failure at the output, a test input must satisfy the original state failure condition and propagate the local failure to the output.

In the following example, we illustrate how the effectiveness of the Tarantula approach can vary with the original state failure condition. Table 3.4 shows the original program from Table 3.1, its faulty version $Version_1$ and a new faulty version $Version_3$. To illustrate the application of the Tarantula approach on these two versions we again consider the same test cases described in section 3.1.1.1. Table 3.5 shows which test case executes which statement, the suspiciousness scores, and the results of the test cases in $Version_3$. The statements are arranged in decreasing order of the suspiciousness score.

The fault in $Version_3$ is assigned a suspiciousness score of 1, while the fault in $Version_1$ is assigned a suspiciousness score of 0.6. In $Version_3$, we need to examine one statement

Table 3.4: Examples of Faults with Different Original State Failure Conditions

Original Program	Faulty $Version_1$	Faulty $Version_3$

`1 int a,b,c;`
`2 int r=0;`
`3 read(a,b,c);`
`4 if (a >= b){`
`5  r = a+b;`
`6  if (b > c){`
`7  r = r+10;`
`8  }`
`9 }`
`10 print(r);`
	`1 int a,b,c;`
	`2 int r=0;`
	`3 read(a,b,c);`
	`4 if (a >= b){`
	`5  r = a-b;`
	`6  if (b > c){`
	`7  r = r+10;`
	`8  }`
	`9 }`
	`10 print(r);`
		`1 int a,b,c;`
		`2 int r=0;`
		`3 read(a,b,c);`
		`4 if (a >= b){`
		`5  r = 100;`
		`6  if (b > c){`
		`7  r = r+10;`
		`8  }`
		`9 }`
		`10 print(r);`

Table 3.5: Suspiciousness Scores of Statements in $Version_3$

Statement	t_1	t_2	t_3	t_4	t_5	t_6	$susp$
r=100		√	√	√	√	√	1
if(b>c)		√	√	√	√	√	1
r=r+10				√	√		1
int a,b,c	√	√	√	√	√	√	0.5
int r=0	√	√	√	√	√	√	0.5
read(a,b,c)	√	√	√	√	√	√	0.5
if(a>=b)	√	√	√	√	√	√	0.5
print(r)	√	√	√	√	√	√	0.5
Result	P	F	F	F	F	F	

before finding the fault. The statement `r = r+10` has the same suspiciousness as the fault but has a lower *brightness* value. In $Version_2$, one needs to examine two statements before finding the fault.

The accessibility values for the faults in the two versions are the same but the original state failure conditions for the faults are different. The original state failure condition of the fault in $Version_1$ is $a + b \neq a - b$, while the original state failure condition of the fault in $Version_3$ is $a + b \neq 100$. The input values for a, b, and c in the test cases are each below 10. With such input values, the original state failure condition of the fault in $Version_3$ is easier to satisfy than that of the fault in $Version_1$. Thus, in $Version_3$, whenever the fault

is executed, the program fails. Thus, the faulty statement never results in a coincidentally correct test case and the faulty statement obtains a suspiciousness score of 1.

On the contrary, in $Version_1$, the faulty statement is executed by two passing test cases, that is, the faulty statement results in two coincidentally correct test cases. This reduces the suspiciousness score of the faulty statement to 0.6. Because the relative suspiciousness of the remaining statements are the same, the fault in $Version_3$ ranks higher than that in $Version_1$.

3.1.1.3 Impact

Impact measures what fraction of the program statements is affected by the execution of the faulty statement. Below, we illustrate how impact affects the effectiveness of the Tarantula approach. We consider an original program and its two faulty versions as shown in Table 3.6. The faulty statements are shown in bold.

Table 3.6: Examples of Faults with Different Impacts

Original Program	Faulty $Version_1$	Faulty $Version_2$
1 int a,b,c; 2 int r; 3 read(a,b,c); 4 if (a >= b) 5 r = a + b; 6 if (b >= c) 7 r+=foo(b,c); 8 print(r);	1 int a,b,c; 2 int r; 3 read(a,b,c); **4 if (a <= b)** 5 r = a + b; 6 if (b >= c) 7 r+=foo(b,c); 8 print(r);	1 int a,b,c; 2 int r; 3 read(a,b,c); 4 if (a >= b) 5 r=a + b; **6 if (b <= c)** 7 r+=foo(b,c); 8 print(r);

The faulty statements in $Version_1$ and $Version_2$ have the same accessibility because neither is nested within any conditional statement. The original state failure condition for the fault in $Version_1$ is `a` $\neq$ `b`, and that for the fault in $Version_2$ is `b` $\neq$ `c`. Both the conditions test the equality of two variables using values obtained from the user input before the values are changed by the program. Since any two randomly selected values from the domain of integer values are much more likely to be unequal than equal, both the original state failure conditions are more likely to be satisfied than not for any input. However,

the faulty statements have different impacts. The value of the faulty condition in $Version_1$ determines whether line 5 is executed or not. The value of the faulty condition in $Version_2$ determines whether the function `foo` is called or not. The function `foo` may have many statements in it. Thus, the evaluation of the faulty condition in $Version_2$ controls the execution of more statements than that for $Version_1$.

The faulty statements in both $Version_1$ and $Version_2$ will obtain a suspiciousness score of 0.5 with any arbitrary test suite. This is because any test case executes both the faulty statements. In $Version_1$, if failing test cases always evaluate the faulty condition to *true*, line 5 always gets executed in the failing runs. Thus, line 5 obtains a higher suspiciousness score than the faulty statement. In $Version_2$, if the failing test cases always evaluate the faulty condition to *true*, all the statements in `foo` get executed in the failing test cases, and obtain a higher suspiciousness value than the faulty statement. This lowers the rank of the faulty statement in $Version_2$. However, if the failing test cases execute the faulty condition in $Version_2$ to false, the statements in `foo` do not execute in the failing runs and are ranked lower than the faulty statement. This improves the rank of the faulty statement.

We state that if many statements are directly or indirectly control dependent on the fault, the fault may have a lower or higher rank depending on how it affects the execution of the control dependent statements. If the control dependent statements are executed in the failing test cases, then they obtain higher suspiciousness scores than the faulty statement. If failure involves non-execution of the control dependent statements, then the fault is likely to obtain a higher rank.

3.1.2 Fault Classification

We describe the fault classes based on accessibility, original state failure condition, and impact. We extend the original state failure based classification of faults by Richardson et al. [42]. We develop measures of accessibility and impact and then present classification schemes based on the measures.

3.1.2.1 Accessibility-based Classification

To measure accessibility, we state that if more conditional statements guard a faulty statement, then the faulty statement is less accessible. The backward static slice (calculated using only control dependencies) of the faulty statement contains all such guarding conditional statements. Therefore, we measure accessibility by the size of the backward slice of the faulty statement as a percentage of the program size. However, this measure does not address the strength of the conditions in the conditional statements.

We apply the k-means clustering algorithm with $k = 2$, to divide the slice sizes into two clusters. k-means is the most commonly used clustering algorithm when the number of clusters, k, is known and the nature of the distribution of the underlying data is not known. k-means clustering divides a set of data points into k clusters such that the sum of squares of distances between the data points and the corresponding cluster centroids is minimal.

The faults in the cluster with the higher mean size of backward slice are called *hard to access* faults. The faults in the cluster with the lower mean are called *easy to access* faults. This classification is ordinal because there is an ordering relationship between the faults of these two classes.

3.1.2.2 Classification Based on Original State Failure Condition

We use the fault classification proposed by Richardson et al. [42], which is based on the original state failure condition. The classification contains six fault classes: (1) constant reference faults, (2) variable reference faults, (3) variable definition faults, (4) conditional operator faults, (5) relational operator faults, and (6) arithmetic operator faults.

This classification is nominal because the original state failure condition of one fault class cannot be compared with that of another fault class. The original state failure conditions for different classes are derived in different ways.

On inspecting the faulty versions in the Siemens benchmark suite, we found the following additional fault classes: (7) missing/ added statement faults, (8) missing/added branch faults, (9) missing/added conditional clause faults. The above fault classes need to be consid-

ered because they have characteristic ways of deriving their original state failure conditions. This classification is nominal because the original state failure condition of one fault class cannot be compared with that of another fault class.

3.1.2.3 Impact-based Classification

We measure the impact of a faulty statement by the size of its forward static slice (as a percentage of the program size) using both data and control dependencies. The forward static slice with respect to a variable at a statement is the set of statements that are affected by the value of the variable at that statement through data and control-dependencies. The static forward slice of a statement is the union of the forward slices of all the variables appearing in the statement.

Based on the sizes of forward slices, we define an ordinal classification of the faults by dividing the slice sizes into two clusters using the k-means clustering. The faults in the cluster with the higher mean value of forward slice size are called *high impact* faults. The faults in the cluster with the lower mean value are called *low impact* faults. This classification is ordinal due to the ordering relation between the faults belonging to the two classes.

3.1.2.4 Factorial Design

The classifications based on accessibility and impact are both ordinal. We use a factorial design by crossing the classes from the two classifications to obtain four treatments as shown in Table 3.7.

Table 3.7: Factorial Design

Treatment	**Acronym**
Hard to access and high impact	HH
Hard to access and low impact	HL
Easy to access and low impact	LL
Easy to access and high impact	LH

3.1.3 Evaluation

The goal of our evaluation was to investigate the effectiveness of the Tarantula approach in localizing faults of different classes. Our independent variable was the fault class, described in Section 3.1.2. Our dependent variable was fault localization effectiveness, which is measured as follows. We measure the average number of statements a tester using the fault localization approach needs to examine in the ranked list of statements before finding the faulty statement. The statements to be examined include all the statements that have a higher suspiciousness score than the faulty statement, and on average, half the statements that share the same suspiciousness score as the faulty statement. The number of statements examined is divided by the total number of executable program statements to obtain the percentage of program statements examined. Higher is the percentage, lower is the effectiveness.

We considered the nature of the test suite to be a confounding variable in our study. Previous work (e.g., [7]) on selection of test suites demonstrated that the choice of test suites affects the effectiveness. To reduce the effect of the confounding variable, we performed our study with two types of test suites: (1) branch coverage based and (2) random.

We used the Siemens suite for this study. The benchmark suite contains 7 benchmark programs `print_tokens, print_tokens2, replace, schedule, schedule2`, and `tot_info`. The sizes of the programs range from 141 to 512 lines of code. There are 130 faulty versions of the programs. We used 20 branch coverage based test suites and 20 random test suites for each benchmark.

3.1.3.1 Original State Failure Condition Based Classes

To collect data on the effectiveness of the approaches, we executed each test suite on each faulty version. If at least one test case in a test suite fails for a faulty version, we obtained the Tarantula rank of the faulty statement for the (faulty version (F), test suite (T)) pair. If there is no failing run obtained for a (faulty version (F), test suite (T)) pair, then we do

not have a data point for that pair. Table 3.8 shows the number of faults and number of (F, T) pairs for each original state failure condition based fault class.

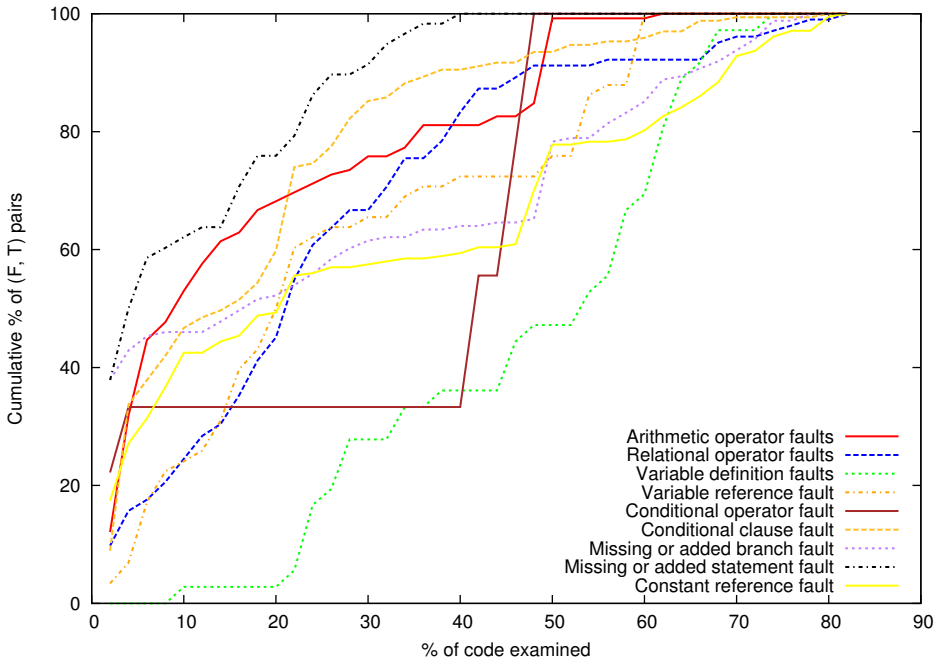
Table 3.8: Fault Data for Original State Failure Condition Based Fault Classes

		#(F, T)	
Fault class	**#faults**	**Branch**	**Random**
Arithmatic operator faults	12	132	114
Relational operator faults	18	102	114
Conditional operator faults	6	9	22
Variable definition faults	16	36	32
Variable reference faults	16	58	73
Constant reference faults	21	207	153
Missing/Added conditional clause	23	169	146
Missing/Added statement	6	58	30
Missing/Added branch	15	161	82

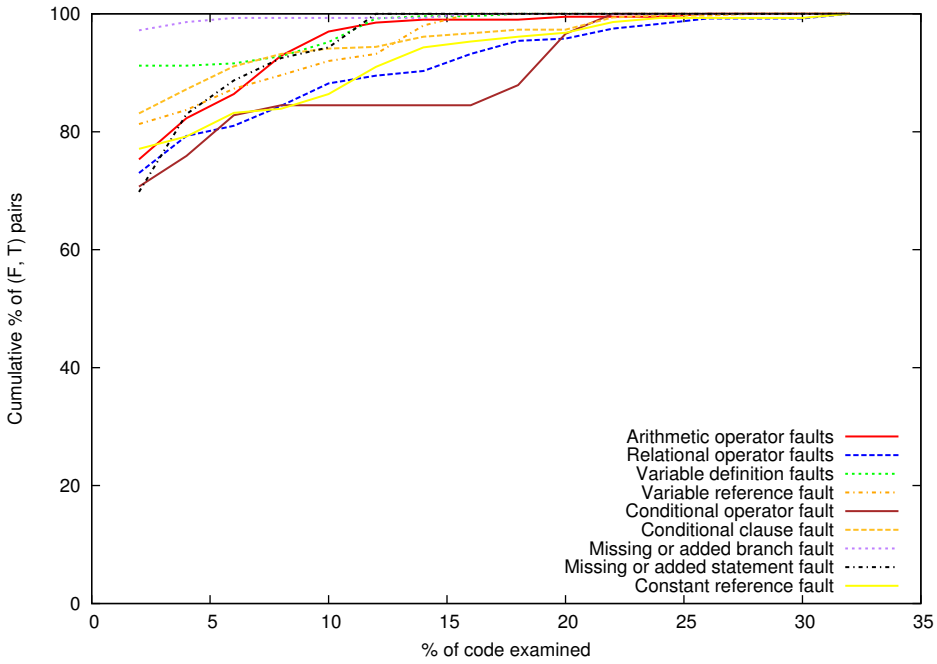
To obtain a graphical representation of the fault localization effectiveness of the Tarantula approach for faults of different classes, we divided the entire range of percentage of code examined (0-100%) into small ranges. We chose ranges of length 2%. For each range, we calculated the percentage of (faulty version, test suite) pairs for which the percentage of code examined belongs to that range. We calculated the cumulative frequency distribution from this data. We plotted the cumulative frequency distribution in a graph. The x-axis of the graphs shows the percentage of code examined. The y-axis shows the cumulative percentage of (faulty version, test suite) pairs. Thus, a point (10, 20) in the graph denotes that for 20% of the (faulty version, test suite) pairs one needs to examine at most 10% of the program statements.

Figures 3.1(a) and 3.1(b) show the plots for different classes of faults based on the original state failure condition, using branch coverage based test suite and random test suites, respectively.

As shown in Figure 3.1(a), when branch coverage test suites were used, there was no total ordering between the fault classes in terms of how effectively they were localized. The curves show that different fault classes were more frequently localized at different ranges of percentage of program statements examined. For example, missing/added branch faults were



(a) Branch Coverage Based Test Suite



(b) Random Test Suite

Figure 3.1: Effectiveness of the Tarantula approach for Original State Failure Condition Based Classes

more frequently localized within the range, 0–25%, than constant reference faults. However,

constant reference faults were more frequently localized within the range, 25–50%, than missing/added branch faults.

We can observe some partial orders of effectiveness among the fault classes. Arithmetic operator faults, conditional clause faults, and missing/added statement faults were clearly more effectively localized than relational operator faults, missing/added branch faults, and variable reference faults.

The relative effectiveness changed significantly when we used random test suites, as shown in Figure 3.1(b). We can arrange the following fault classes in the order of how effectively they were localized (from high effectiveness to low): (1) missing/added branch faults, (2) variable definition faults, (3) relational operator faults, (4) conditional operator faults. These orders did not hold for branch coverage based test suites.

In summary, the observations are as follows: (1) Only some partial order of effectiveness was found between the fault classes, (2) The relative effectiveness varied significantly with the type of test suites used. This led us to conclude that there was no clear relationship between the original state failure condition based fault classes and the effectiveness of fault localization.

3.1.3.2 Accessibility and Impact Based Fault Classes

Table 3.9 shows the accessibility-based faults clusters with the high and low means along with their corresponding mean values, ranges, and number of faults. The term $\%Bslice$ denotes the backward static slice size as a percentage of the size of the program. Table 3.10 shows the same information as Table 3.9 for the impact based clusters. The term $\%Fslice$ denotes the forward static slice size as a percentage of the size of the program.

Table 3.9: Clusters of Faults Based on Accessibility

Cluster	Cluster mean (%Bslice)	Cluster range (%Bslice)	#faults
High	4.67%	3.56% - 8.55%	57
Low	2.28%	0.6% - 3.42%	73

Table 3.10: Clusters of Faults Based on Impact

Cluster	Cluster mean (%Fslice)	Cluster range (%Fslice)	#faults
High	87.3%	63.5% - 99.0%	47
Low	33.7%	2.14% - 54.68%	83

Table 3.11 shows the number of faults for each of the four combinations of accessibility and impact based fault classes. It also shows the number of (F, T) pairs for each fault class and for each type of test suite.

Table 3.11: Fault Data for Accessibility and Impact Based Fault Classes

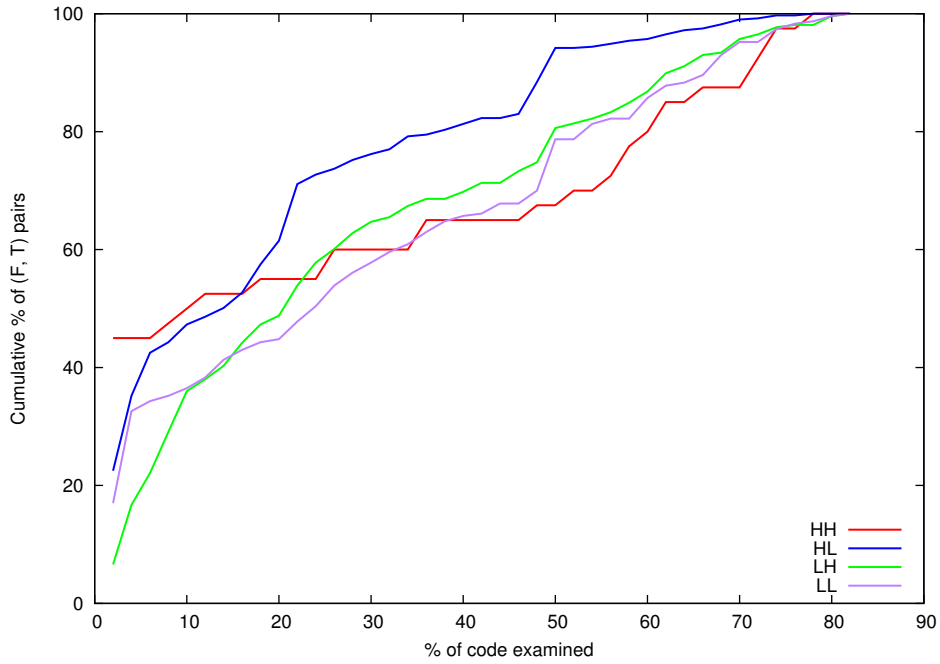
		#(F, T)	
Fault class	**#faults**	**Branch**	**Random**
HH	4	40	5
HL	53	395	342
LL	30	230	210
LH	43	258	201

Figures 3.2(a) and 3.2(b) show the plots for effectiveness obtained for different fault classes based on accessibility and impact, using branch coverage based test suites and random test suites, respectively.

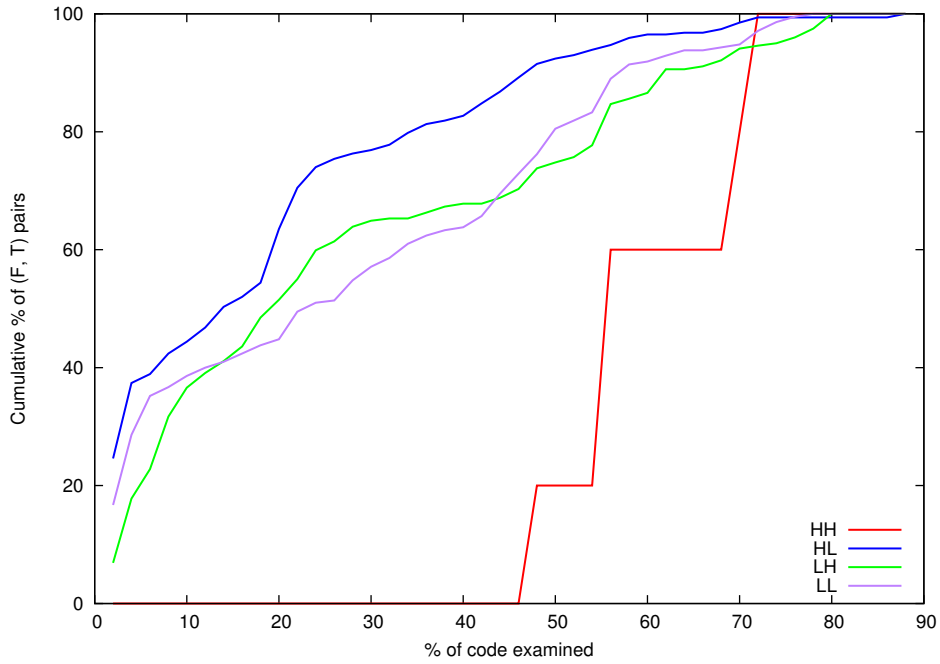
As Figure 3.2(a) shows, when branch coverage based test suites are used, HL faults are localized most effectively. This supports our earlier analysis that an HL fault is likely to be most effectively localized because (1) fewer passing test cases execute it, thereby increasing its suspiciousness score, and (2) fewer statements may have higher suspiciousness scores than the faulty statement because the execution of fewer statements depends on it.

Although, overall the HL faults are more effectively localized than HH faults, in the range 0-20% of code examined, HH faults are more frequently localized than HL faults. Each of these HH faults causes failure by not executing the statements that are dependent on it. This results in higher ranks for such a fault.

Both LH and LL faults are less effectively localized than HL faults. This again supports our analysis that faults that are harder to access are localized more effectively. The overall effectiveness for LL and LH faults are comparable. This also suggests that if a fault is easy



(a) Branch Coverage Based Test Suite



(b) Random Test Suite

Figure 3.2: Effectiveness of the Tarantula approach for Accessibility and Impact Based Fault Classes

to access, then the impact of the fault does not appreciably affect the effectiveness of its localization.

The observations are consistent when we use random test suites, as shown in Figure 3.2(b). HL faults are most effectively localized than all the other types. The effectiveness for LL and LH faults are comparable. We had five data points for HH faults with the random test suites. In all these cases, the HH faults caused failures by executing the statements that are control dependent on the faults. Therefore, many statements obtained higher suspiciousness scores than the faulty statements, thereby decreasing the rank of the faults.

3.1.4 Threats to Validity

External: Threats to external validity stem from the nature of benchmarks used. The programs are relatively small, and contain only one fault each.

Internal: There is a threat to internal validity in our study of effectiveness of the Tarantula approach for the accessibility and impact based fault classes. We did not consider the effect of the original state failure condition while inferring that faults that are hard to reach and have low impact are localized most effectively. The high effectiveness for these faults may have been caused by hard to satisfy original state failure conditions.

Construct: There are several threats to construct validity. The backward static slice of a faulty statement may not always accurately measure the accessibility of the faulty statement. A backward static slice is the union of all the statements that affect a given statement through control dependencies, along all program paths. When a test case executes, the faulty statement is accessed by a particular path. A faulty statement that can be accessed by many easily executable paths may have a large backward static slice but still be easily executable by a single test case.

We use the slice size as a measure of accessibility. However, the number of statements on which a given statement is control dependent does not always accurately measure the accessibility of the given statement. Consider a statement s_1 that is nested inside two conditional statements, and another statement s_2 that is nested within a single conditional statement. Thus the accessibility measure of s_1 is higher than that of s_2, and s_1 may be

categorized as a hard to access fault, while s_2 an easy to access fault. However, s_2 may be harder to access than s_1 because the condition guarding s_2 may be harder to satisfy than the conjunction of the two conditions that guard s_1.

Forward static slices may not always accurately measure the impact. A forward static slice contains all the statements dependent on a given statement along any program path. A test case executes only one path among them, which may result in a small impact. However, the union of all potentially impacted statements, which makes up the forward static slice, may be large.

The percentage of code examined may not correctly represent the effort to locate the faulty statement. This is because a tester examines each statement and determines if the statement is faulty or not. Determining the correctness of each statement requires an amount of effort, which may be different for different statements. The percentage of program statements examined does not address these differences in effort.

3.1.5 Conclusions and Lessons Learned

As discussed in Section 3.1.1.1, among any two faults having a similar impact and original state failure condition, the fault that is easier to be accessed by test cases is likely to cause test cases to be coincidentally correct more frequently and thus, is likely to be localized less effectively. This expectation was validated by our study, which showed that the easy to access faults, in general, are localized less effectively than hard to access faults. Thus, this study establishes that the presence of coincidentally correct test cases degrades fault localization effectiveness.

By classifying faults and analyzing the effect of fault classes on fault localization effectiveness, we also gained insight into how similar a coincidentally correct test case and a failing test case would be for faults with different levels of accessibility. For hard to access faults, a coincidentally correct test case is likely to cover similar statements as a failing test case. For easy to access faults, a coincidentally correct test case may or may not cover similar program statements as a failing test case. We explain the reason below.

For hard to access faults, there are many statements on which the faulty statement is control dependent. Both a failing and a coincidentally correct test case must reach the faulty statement and thus, must execute all the statements on which the faulty statement is control dependent. For faults that are easy to access, the faulty statement is control dependent on only a few statements, which are covered by both the failing and the coincidentally correct test cases. However, after executing the fault, the coincidentally correct test case may or may not execute a similar path as the failing test case. Depending on that, the coverage of a coincidentally correct test case may or may not be similar to a failing test case. These insights proved to be valuable when we studied the effectiveness of our family of approaches.

3.2 Proximity Based Weighting of Test Cases for Fault Localization

Spectrum based fault localization approaches, such as Ochiai and Tarantula, implicitly assume that all test cases are equally important. However, research on test case generation and selection techniques [7, 13] has shown that using certain test cases can lead to more effective fault localization than others. We believe that the knowledge of what test cases lead to better fault localization should be incorporated into the equations for calculating suspiciousness scores, such as in the equations of the Ochiai and Tarantula approaches. We present an approach that assigns weights to every passing test case representing the importance of the passing test case in fault localization. The weights, instead of the numbers, are then used to calculate suspiciousness scores. We present a study of fault localization effectiveness demonstrating that the weighting of test cases improves fault localization effectiveness.

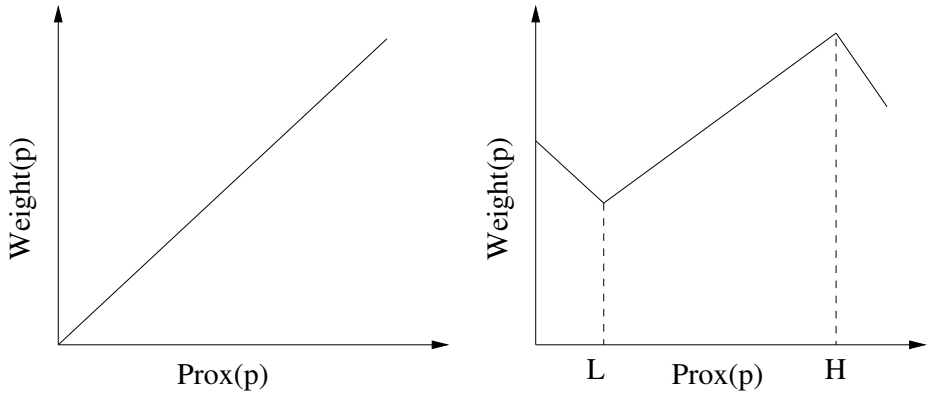
3.2.1 Approach

We extend the nearest neighbor model [41] to define the relative importance of test cases. The nearest neighbor model assumes the presence of a single failing test case and multiple passing test cases. It is based on the heuristic that the statements covered by the failing test case but not by a passing test case are suspicious. Additionally, selecting the passing test

case that is most similar to the failing test case results in the most accurate set of suspicious statements. The reason behind selecting the passing test case most similar to the failing test case is that the two test cases would execute the program with similar run-time contexts.

For evaluating the relative importance, we need to extend the heuristic used by the nearest neighbor model because (1) the nearest neighbor model uses one failing test case, while the Ochiai approach uses multiple failing test cases, and (2) the nearest neighbor model only selects the passing test case that is most similar to the failing test case, while the Ochiai approach includes all the passing test cases to calculate suspiciousness scores.

We address (1) by calculating the average proximity of each passing test case with all the failing test cases. To address (2), we state that the importance of a passing test case is proportional to its average proximity ($Prox$) to the failing test cases. To every test case, we assign a weight representing its importance. This relationship of weight with proximity is shown in Figure 3.3(a).



(a) Without Thresholds (b) With Two Thresholds

Figure 3.3: Variation of Weight with Proximity

However, in our experiments, we observed that this relationship does not hold for some of the test cases. Test cases that have $Prox$ values lower than a low threshold L, do not execute the faulty statement. Assigning a low weight to these test cases increases the suspiciousness scores of other statements but not of the faulty statement, resulting in a decrease in the rank of the faulty statement.

Some passing test cases that have a $Prox$ value higher than H may execute the faulty statement and execute the program from a similar run-time context as the failing test cases and still pass. This happens because these passing test cases have run-time contexts that are different from that of the failing test cases in terms of variable values, not in terms of sets of covered statements. Assigning a high weight to these test cases decreases the suspiciousness score of the faulty statement resulting in reduced effectiveness.

To address these two types of test cases, we vary the weight in negative proportions to proximity, if the proximity is below L or above H. This is shown in Figure 3.3(b).

We measure the code coverage based proximity (CC-Proximity) proposed by Liu et al. [35] of each passing test case with the failing test cases. CC-Proximity measures the proximity of two test cases using the Jaccard similarity of their sets of covered statements. If S_1 and S_2 respectively are the sets of covered statements of two test cases t_1 and t_2, the CC-Proximity $D(t_1, t_2)$ between the test cases given by the expression $\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$.

If there is more than one failing test case, we calculate the average CC-Proximity of each passing test case with all the failing test cases, so that each failing test case has equal contribution in the calculation of the weights. Thus, if F is the set of all failing test cases, the proximity of a passing test case p, $Prox(p)$ is given by the expression $\frac{\sum_{f \in F} D(p,f)}{|F|}$.

Based on the measure of proximity, we measure the weight of each passing test case p, $Wt(p)$, by Equation 3.1.

$$Wt(p) = \begin{cases} (1 - Prox(p)) & \text{if } Prox(p) < L \\ Prox(p) & \text{if } L \leq Prox(p) \leq H \\ (1 - Prox(p)) & \text{if } Prox(p) > H \end{cases} \tag{3.1}$$

In order to keep the range of suspiciousness scores unchanged, we multiply the $Wt(p)$ by a constant k, given by the expression $\frac{|P|}{\sum_{p \in P} Wt(p)}$.

We choose the lower threshold to be either the lower tail or the lower quartile. We choose the upper threshold to be either the upper tail or the upper quartile of a box plot. The tails result in more conservative choices than the quartiles. Table 3.12 lists different threshold selections.

Table 3.12: Alternatives For Weighting Passing Test Cases

Lower Threshold	**Upper Threshold**	**Acronym**
None	None	WT-NoThresh
Lower quartile	None	WT-LQ
Lower quartile	Upper quartile	WT-LQ-UQ
Lower tail	None	WT-LT
Lower tail	Upper tail	WT-LT-UT
Lower tail	Upper quartile	WT-LT-UQ
Lower quartile	Upper tail	WT-LQ-UT

We define the weighted Ochiai score, $wtSusp(s)$, of a statement s to be the Ochiai score of the statement calculated by replacing $passed(s)$ with the total weight of the passing test cases that execute s. This can be expressed mathematically by Equation 3.2. P_s denotes the set of passing test cases that execute s.

$$wtSusp(s) = \frac{failed(s)}{\sqrt{totalFailed \times (\sum_{p \in P_s} Wt(p) + failed(s))}} \tag{3.2}$$

A tester using our approach inspects the statements in the decreasing order of $wtSusp$ values until the faulty statement is found.

3.2.2 Evaluation

We performed an evaluation to investigate whether or not the weighting of passing test cases improves the effectiveness of fault localization. We evaluated the approaches listed in Table 3.12. We used the measure of effectiveness of fault localization described in Section 3.1.3.

We used the Siemens suite for this study. The suite contains 7 benchmark programs and 130 faulty versions of those programs. We used 50 branch coverage based test suites for each benchmark.

To collect data on the effectiveness of the approaches, we executed each test suite on each faulty version. If at least one test case in a test suite failed for a faulty version, we obtained the Ochiai rank of the faulty statement for the (faulty version, test suite) pair.

Table 3.13 shows the benchmark programs and the number of (faulty version, test suite) pairs producing at least one failing run for each benchmark program.

Table 3.13 also shows the average percentage of program statements examined for each approach and benchmark over all (faulty version, test suite) pairs. The column 'benchmark' shows the benchmark programs. The column #(F,T) denotes the number of (faulty version, test suite) pairs. Each of the remaining columns shows the fault localization effectiveness of the corresponding approach. For example, the column 'WT-LQ' shows the average percentage of program statements examined for each benchmark when the approach 'WT-LQ' was used to localize the fault.

Table 3.13: Average percentage of program statements examined for each benchmark and each approach with branch coverage based test suite

Benchmark	#(F,T)	Ochiai	WT-LQ	WT-LQ-UQ	WT-NoTh	WT-LT-UT	WT-LT	WT-LT-UQ	WT-LQ-UT
print_tokens	119	13.8	13.3	14.9	13.3	13.4	13.3	14.9	13.4
print_tokens2	380	15.5	15.5	15.7	15.5	15.3	15.6	15.8	15.5
replace	629	9.4	8.7	9.2	8.9	8.7	8.7	9.1	8.9
schedule	153	16.4	15.4	14.9	15.4	15.4	15.4	14.9	15.4
schedule2	38	56.2	56.3	56.0	56.0	56.3	56.3	56.1	56.0
tcas	366	20.0	20.0	20.1	20.1	20.1	20.0	20.1	20.0
tot_info	597	25.3	25.7	25.7	25.7	25.7	25.7	25.8	25.6

We make the following observations from the results: (1) For all benchmarks except for `tot_info`, at least one of the weighting approaches outperforms Ochiai. The weighting approaches, on average, require 0.1% to 1.5% less code to examine than Ochiai, and (2) For different benchmarks, different threshold selections lead to the highest effectiveness. No particular weighting approach can be considered to be the best.

We discuss the reasons why our approach only resulted in only 0.1% to 1.5% improvement in effectiveness and was not able the improve the effectiveness for some benchmarks. The goal of calculating weighted suspiciousness scores was to assign higher suspiciousness scores to the statements that were executed by test cases having lower weight values. However, Equation 3.2 often failed to assign suspiciousness scores in such a manner. For example, a

statement s_1 that is executed by n_1 test cases having low weight values may obtain a lower suspiciousness score than another statement s_2 that is executed by n_2 test cases having high weight values, if $n_2 < n_1$. In our study, the faulty statement was often executed by many test cases with low weight values, and thus, obtained a lower suspiciousness score than the non-faulty statements that were executed by a few test cases with high weight values.

Another reason for the limited improvement in effectiveness was that the CC-proximity measure did not always measure the proximity of run-time context, leading to incorrect estimation of the importance of the test cases. Also, in the presence of multiple failing test cases, averaging the CC-proximity values of a passing test case often resulted in moderate weights for the test case. In such cases, calculating weighted suspiciousness scores did not significantly change the ranking of statements.

3.2.3 Threats to Validity

External: Threats to external validity stem from the nature of the benchmarks used and the size of the test suites. Siemens programs are relatively small and they have simple control structures. We used test suites each containing 15–20 test cases. When larger test suites are used, the fault localization effectiveness of the Ochiai approach may already be high, and there may be no opportunity for improving the effectiveness by weighting test cases.

Construct: A threat to construct validity is associated with the measure of the fault localization effectiveness. This threat was discussed in Section 3.1.4.

3.2.4 Conclusions and Lessons Learned

Although the weighting of test cases did not directly address coincidental correctness, we observed that it is most effective to assign weights to the test cases in such a way that every coincidentally correct test case obtains a weight lower than the other passing test cases. With such an assignment of weights, the suspiciousness score of the faulty statement is increased by the highest amount. Thus, we realized that it is worthwhile to direct our research to

the classification of coincidentally correct test cases rather than to the evaluation of relative importance of test cases.

As we observed, using weighted suspiciousness scores often did not result in higher suspiciousness scores for the statements executed by test cases having lower weight values. Therefore, if coincidentally correct test cases are classified, assigning the classified test cases low weight values and calculating weighted suspiciousness scores will not be an effective approach to utilize the classification. Therefore, alternative approaches are needed to incorporate the information obtained from the classification in the calculation of suspiciousness scores.

3.3 Mitigating the Effect of Coincidental Correctness Using Tester Feedback

In Section 2.3, we described Masri et al.'s [37] approach that evaluates for each test case the likelihood that the test case is coincidentally correct and then classifies a fixed percentage of the test cases having the highest likelihood values as coincidentally correct. An approach that classifies a fixed percentage of test cases as coincidentally correct generates false positives when the fixed percentage is higher than the actual percentage of coincidentally correct test cases. If the fixed percentage is lower than the actual percentage, false negatives result. Masri et al. reports that their approach generated 5.1% and 54% of false positives and false negatives respectively, with $k = 60$. It follows from Equation 1.1 that removing false positives will increase the Ochiai suspiciousness scores of non-faulty statements that are executed by the false positives, thereby reducing the fault localization effectiveness.

We develop an approach that prevents false positives by ensuring that the number of coincidentally correct test cases classified is always less than the actual number of coincidentally correct test cases. Our approach iteratively classifies and removes coincidentally correct test cases. In each iteration, a tester is presented with only those statements that share the highest Ochiai suspiciousness score. Based on tester feedback on whether the faulty statement is one of the presented statements or not, a lower bound n for the number of coincidentally

correct test cases is estimated. Then, a measure of likelihood of coincidental correctness is calculated for every passing test case and n test cases with the highest measure are classified as coincidentally correct. The classified test cases are removed and the suspiciousness scores of the statements are recalculated. This process is repeated until the faulty statement is found. We describe our approach and present the result of an evaluation with the Siemens suite and Unix utility programs.

3.3.1 Approach

Figure 3.4 shows the procedure describing our approach. The inputs to the procedure are a list of program statements, $allStatements$; a list of passing test cases, $allPassingTests$; and a list of failing test cases, $allFailingTests$. Each $TestCase$ instance contains information regarding what statements are covered by the corresponding test case.

The variables, $passingTests$ and $statements$, are initialized to $allPassingTests$ and $allStatements$, respectively. The variable, $alreadyInspectedStatements$, initialized to be an empty set, contains the statements that are already presented to the tester. The variable, $remainingCC$, maintains the number of coincidentally correct test cases remaining to be classified.

Lines 7–9 calculate the Ochiai suspiciousness scores of the statements. We only use the $statements$, $passingTests$, and all the failing test cases to calculate the suspiciousness scores. The sets, $statements$ and $passingTests$, are updated as the tester inspects the statements and as more coincidentally correct test cases are classified and removed.

Line 11 presents the tester with the statements having the highest suspiciousness score. In lines 13 and 14, if the faulty statement is found in the list of statements presented, S, then the procedure returns.

Lines 14–24 show the steps performed when the tester reports that the faulty statement is not in the list of statements S. The statements in S are added to the set, $alreadyInspectedStatements$, and are removed from the set, $statements$. Next, the number of passing test cases that execute any statement with the second highest suspiciousness score is determined.

```
1: procedure FEEDBACKDRIVENLOCALIZATION((List < Statement > allStatements, List <
   TestCase > allPassingTests, List < TestCase > allFailingTests)
2:     passingTests = allPassingTests
3:     statements = allStatements
4:     alreadyInspectedStatements = {}
5:     alreadyPredictedCC = 0
6:     remainingCC = 0
7:     repeat
8:         for all Statement s ∈ statements do
9:             susp(s) = calculateOchiai(statements, passingTests, allFailingTests)
10:        end for
11:        Present the tester with the set of statements S having the highest susp(s)
12:        if tester reports that S contains the faulty statement then
13:            return
14:        else
15:            alreadyInspectedStatements =alreadyInspectedStatements ∪ S
16:            statements = statements − S
17:            minCC = number of passing test cases that execute the statements
18:            sharing the second highest suspiciousness score
19:            remainingCC = minCC − alreadyPredictedCC
20:            for all TestCase p ∈ passingTests do
21:                CC(p) = calculateCCProbability(p)
22:            end for
23:            selectedCC = remainingCC tests in passingTests having the highest CC values
24:            passingTests = passingTests − selectedCC
25:        end if
26:    until remainingCC ≤ 0
27:    ▷ Presence of false positives is detected. Therefore, use initial Ochiai suspiciousness scores
28:    S_unexamined=allStatements−alreadyInspectedStatements
29:    Present ranked list of statements in S_unexamined based on initial Ochiai suspiciousness scores
30:    return
31: end procedure
```

Figure 3.4: Procedure describing tester feedback based fault localization

This number is the same as the minimum number of coincidentally correct test cases, $minCC$, present in the test suite. This is because the maximum possible suspiciousness score of the faulty statement is the second highest score. Thus, the minimum number of passing test cases that execute the faulty statement is the same as the number of passing test cases that execute the statements with the second highest score. This is because the number of passing test cases monotonically increases with a decrease in suspiciousness scores.

Once we determine $minCC$, we derive $remainingCC$, the number of remaining coincidentally correct test cases, by subtracting the number of already classified coincidentally correct test cases from $minCC$.

In lines 19–22, we classify the remaining coincidentally correct test cases. For each passing test case p, we calculate a measure $CC(p)$, representing its likelihood of being coincidentally correct. In Section 3.3.2, we discuss the details of calculating the likelihood of coincidental correctness. We sort the passing test cases in decreasing order of $CC(p)$. We select the first $remainingCC$ number of passing tests in the sorted list and remove them from the list $passingTests$. We add the test cases classified as coincidentally correct to the list $alreadyIdentifiedCC$. In line 26 we return to the loop if there are more coincidentally correct test cases to classify.

If $remainingCC \leq 0$ and the faulty statement is still not found, we infer that our classifications contain false positives. A negative value of $remainingCC$ indicates that we have classified more coincidentally correct test cases than actually present. If $remainingCC$ is 0 and the faulty statement is still not found, it indicates that we have classified as many coincidentally correct test cases as actually present, but one or more of the test cases are classified incorrectly. This is because if we had classified all coincidentally correct test cases accurately, the faulty statement would have obtained the highest suspiciousness score. Thus, the faulty statement would have been found in the set of statements presented in the previous iteration.

When we infer that there are false positives, we terminate the loop. We present the tester the ranked list of statements, excluding the statements already inspected, based on the original Ochiai suspiciousness scores. We do not use the suspiciousness scores calculated in some iteration before the loop terminated, because we cannot determine at which iteration false positives were introduced.

In any iteration, our approach can result in false negatives. The number of coincidentally correct test cases that are classified never exceeds the lower bound for the number of actual coincidentally correct test cases. If the number of actual coincidentally correct test cases is

greater than the lower bound, some coincidentally correct test cases will not be classified, resulting in false negatives.

3.3.2 Estimating the Likelihood of Coincidental Correctness

We define a measure to estimate the likelihood that a passing test case is coincidentally correct. Our measure is based on the measure defined by Masri et al. [37], which was described in Section 2.3. We discuss the limitations of the measure and then develop a new measure that addresses the limitations.

Equation 2.6 shows Masri et al.'s measure for estimating the likelihood of coincidental correctness for every passing test case. The value of the first term in the measure varies between 0.5 and 1, while the second term is a percentage that varies between 0 and 100. By adding the two terms, the measure assigns more importance to statement coverage than the average suspiciousness score of the covered statements. Thus, even a small difference in coverage can nullify a large difference in the average suspiciousness score of covered statements.

The importance of the two terms can be made commensurate if we consider the fraction of statements covered instead of the percentage. This is given by the the expression $\frac{\sum_{s \in S_p} Tarantula(s)}{|S_p|} + \frac{|S_p|}{|S|}$. However, this measure considers the coverage of every statement in the same way. We state that the coverage of a statement having a higher suspiciousness score is more important.

We defined our own measure based on the measure proposed by Masri et al. We use the Ochiai suspiciousness score instead of the Tarantula suspiciousness score. We add the Ochiai suspiciousness score of all statements covered by a passing test case p, to measure the likelihood of p being coincidentally correct. Equation 3.3 expresses it mathematically.

$$CC(p) = \sum_{s \in S_p} Ochiai(s) \tag{3.3}$$

Our measure $CC(p)$ has the following similarities with Masri et al.'s measure $CC_{Masri}(p)$:

- Both the measures produce a high value for a passing test case that primarily executes statements with high suspiciousness scores.

- Both the measures produce a higher value for test cases having a higher coverage.
- $CC_{Masri}(p)$ only considers statements with their Tarantula suspiciousness score between 0.5 and 1. Our approach excludes all the statements that are not executed by every failing test case. For the statements that are executed by every failing test case, the Tarantula score varies between 0.5 and 1. This is because for these statements, $\%failed(s) = 100$ in Tarantula's expression $\frac{\%failed(s)}{\%passed(s)+failed(s)}$.

Our measure has the following differences with Masri et al.'s measure:

- $CC_{Masri}(p)$ considers the coverage of every statement equally important. $CC(p)$ considers the coverage of a statement with a higher suspiciousness score more important.
- $CC(p)$ uses the Ochiai score, while $CC_{Masri}(p)$ uses the Tarantula score because the Ochiai approach has been demonstrated to be more accurate than the Tarantula approach [2].

3.3.3 Evaluation

We performed an evaluation to investigate the following three research questions:

- **RQ1:** How accurate is the $CC(p)$ measure in classifying coincidentally correct test cases?
- **RQ2:** Is our approach more effective than the Ochiai approach?
- **RQ3:** Is our approach more effective than classification and removal of fixed percentage of coincidentally correct test cases?

We evaluated three approaches: (1) Ochiai, (2) our approach, and (3) *fixedCC*: an approach based on the classification of a fixed percentage of coincidentally correct test cases.

For the first approach we did not calculate the Ochiai scores of statements that are not executed by every failing test case. This is to remove bias in favor of our feedback driven approach, which excludes statements not executed by all failing test cases.

The approach *fixedCC* is adapted from Masri et al.'s approach. We did not compare our approach directly to Masri et al.'s approach. This is because our goal was to investigate the relative effectiveness resulting from iterative classification of coincidentally correct test cases versus the classification using a fixed percentage, rather than the effect of using different measures for estimating the likelihood of coincidental correctness. Masri et al.'s approach uses $CC_{Masri}(p)$, while our approach uses $CC(p)$ as a measure of likelihood of coincidental correctness.

For the *fixedCC* approach, we used $CC(p)$ to estimate the likelihood of coincidental correctness. We first calculated the $CC(p)$ for every passing test case. We sorted the passing test cases in decreasing order of the value of $CC(p)$ and then selected the top 60% of the test cases as coincidentally correct. We used the top 60% because Masri et al. empirically showed that selecting them results in the least number of false positives.

Our dependent variable is the effectiveness of fault localization. To measure effectiveness we calculated the percentage of the program statements a tester needs to examine before finding the faulty statement.

For the Ochiai approach, the fault localization effectiveness is calculated as described in Section 3.1.3. We present a mathematical expression for the fault localization effectiveness of the Ochiai approach to facilitate explaining the calculation of the fault localization effectiveness of our approach. Suppose that the faulty statement is denoted by f. The Ochiai suspiciousness score of any statement, s, is denoted by $Ochiai(s)$. Equation 3.4 shows the number of statements examined, $numExamined$, before finding the faulty statement.

$$\begin{aligned} numExamined = &|\{t : Ochiai(t) > Ochiai(f)\}| \\ &+|\{t : Ochiai(t) = Ochiai(f)\}|/2 \end{aligned} \tag{3.4}$$

We measured the effectiveness of Ochiai by the expression $\frac{numExamined}{n} \times 100$, where n denotes the number of executable statements in the program. A lower value of the measure indicates higher effectiveness.

In our approach, the calculation of $numExamined$ depends on whether or not the tester is able to locate the faulty statement in any iteration. If the faulty statement is located in

iteration i, then $numExamined$ is the sum of the number of statements presented in the iterations $1, 2, \ldots, i-1$, and the half the statements presented in iteration i. We assume that on average, a tester inspects half the statements that share the same suspiciousness score as the faulty statement.

If our approach fails after iteration i due to the identification of false positives, then we present the tester with the unexamined statements in the set $S_{unexamined}$, sorted in the order of the initial suspiciousness scores. The value of $numExamined$ is the sum of number of statements presented in iterations $1, 2, \ldots, i$, and the number statements inspected in the set $S_{unexamined}$, according to Equation 3.4.

We used the Siemens suite containing 7 benchmarks, 4 versions of the Unix grep program, and 5 versions of the Unix gzip program. We used the test suites included in the Siemens benchmark. We selected 50 random test suites for this study. For each of grep and gzip, only a single test suite containing all the test cases is available. We created 50 random test suites by randomly selecting and grouping test cases.

To collect data on the effectiveness of the approaches, we executed each test suite on each faulty version. If at least one test case in a test suite failed for a faulty version, we obtained a Ochiai rank of the faulty statement for the (faulty version, test suite) pair. If there was no failing run obtained for a (faulty version, test suite) pair, then we did not have a data point for that pair. To investigate RQ1, we also measured the percentage of test cases that were not coincidentally correct in the top 60% of test cases having the highest $CC(p)$ measure. In the following sections, we study the three research questions.

3.3.3.1 RQ1 Results

Table 3.14 shows the percentage of test cases not coincidentally correct (false positives) in the top 60% of test cases having the highest $CC(p)$ measure and the corresponding percentage of (faulty version, test suite) pairs. For example, false positives occurred in the range 80%–100% in 35.6% of the (faulty version, test suite) pairs.

For 68.6% of the (faulty version, test suite) pairs, the percentage of false positives was either in the range 0–20% or in the range 80-100%. This is because for these pairs, either there were no or few coincidentally correct test cases, or most of the passing test cases were coincidentally correct. In the former case, the percentage of false positives was high, while in the latter case the percentage of the false positives was low.

Table 3.14: Distribution of false positives

Percentage of false positives (%)	Percentage of (faulty version, test suite)
0–20%	33.0
20–40%	8.6
40–60%	10.8
60–80%	12.0
80–100%	35.6

3.3.3.2 RQ2 Results

Table 3.15 shows for each benchmark, the percentage of (faulty version, test suite) pairs for which our feedback based approach performed better than, worse than, or the same as the Ochiai approach.

Table 3.15: Distribution of the number of change in effectiveness

Program	Improvement (%)	Reduction (%)	Identical (%)
print_tokens	31.2	25.0	43.8
print_tokens2	18.6	19.7	61.7
replace	29.7	23.0	47.3
schedule	22.2	31.1	46.7
schedule2	55.0	25.0	20.0
tcas	25.2	24.6	50.1
tot_info	35.6	15.4	49.0
gzip	20.2	12.5	67.3
grep	17.3	6.4	76.3

Table 3.16 shows the distribution of the amount of positive improvements for each benchmark. Suppose for a (faulty version, test suite) pair, the Ochiai approach requires examining $x\%$ of the code and our feedback based approach $y\%$ of the code. If there is an improvement,

then $y < x$. In that case, we define percentage improvement by the expression $\frac{x-y}{x} \times 100$. In Table 3.16, we show the mean, median, maximum and the average improvement for each benchmark.

Table 3.16: Distribution of percentage of improvements

Program	Max(%)	Min (%)	Median (%)	Average (%)
print_tokens	50.0	2.0	16.7	21.4
print_tokens2	40.0	1.3	7.14	11.4
replace	66.6	1.03	16.7	19.2
schedule	50.0	5.7	33.3	27.8
schedule2	40.0	2.8	11.7	15.5
tcas	28.5	3.2	19.4	16.1
tot_info	63.3	1.14	9.2	13.8
gzip	33.0	0.3	1.8	7.4
grep	50.6	0.1	2.6	6.3

Table 3.15 shows that there are cases when the effectiveness of the feedback based approach is better, worse or identical compared to that of the Ochiai approach. In the Siemens suite, for the benchmarks `print_tokens2`, `replace`, `schedule2`, and `tot_info`, the effectiveness of our approach was better more often than it was worse. For the Unix utilities, the percentage of improvements was significantly more than the percentage of cases when our approach performed worse. In all the benchmarks except `schedule2`, the same effectiveness was observed in at least 40% of the cases. Our feedback based approach achieved up to 67% improvement over the Ochiai approach, for the `replace` benchmark. Below, we use examples to illustrate the differences between the performances of the two approaches.

An improvement in effectiveness or an identical effectiveness results when there are no false positives in the classifications of coincidentally correct test cases. In such cases, the tester locates the faulty statement during some iteration of the loop in our approach. Depending on the order in which coincidentally correct test cases are classified, the effectiveness may be higher or identical. An improvement occurs when the removal of coincidentally correct test cases increases the suspiciousness score of the faulty statement but does not increase the suspiciousness scores of statements with a higher original Ochiai rank. An identical effectiveness results when the removal of the coincidentally correct test cases increases the

suspiciousness scores of all the statements that ranked higher than the faulty statement in the original ranking.

An example of improvement in the benchmark `gzip-1.2.3` is shown in Table 3.17. Each row of the table shows the set of statements that share the same suspiciousness score. The suspiciousness score corresponding to a row is greater than or equal to the suspiciousness score in the row below. The row that contains the faulty statement is shaded. The table also shows the passing and failing test cases that cover the statements. Each passing test case is represented by a number that is the index of the test case in a list. A passing test case p that is classified to be coincidentally correct is shown as p'.

Test cases $152, 137, 197$, and 176 are coincidentally correct. Also all of them have been classified to be coincidentally correct. This increases the suspiciousness score of the statements in the row containing the fault and the statements in the first row. However, the suspiciousness scores of the statements in the second row are not increased. Thus, a tester does not need to inspect the statements in the second row, while a tester using Ochiai will inspect those statements.

Table 3.17: Example of improvement over Ochiai

Statements	Passing tests	Failing tests
$if(backlash), \ldots$	$176'$	f_1
$elseif(p[0] ==' -'\&\&p[1]! =']'), \ldots$	$70, 59, 46$	f_1
$\vdots$	$\vdots$	$\vdots$
$if(!fillsize), \ldots$	$152', 137', 197',$	f_1

False positives in the classification result in identical effectiveness or lowered effectiveness. Depending on which test cases are false positives, an identical or lower effectiveness may occur. In either case, false positives cause the $remainingCC$ to reduce to 0 and the loop in our approach to terminate.

Identical effectiveness occurs when $remainingCC$ turns to 0 before any non-faulty statement's suspiciousness score exceeds the faulty statement's suspiciousness score. Table 3.18 illustrates such a scenario with the example of the `print_tokens` benchmark. Passing test cases $16, 3$, and 5 have been already classified to be coincidentally correct. The classification

of test case 16 is a false positive. Suppose that the statements in the first row are being inspected during an iteration. Three test cases are already classified to be coincidentally correct. The statements sharing the second highest suspiciousness score are executed by three passing test cases. Therefore, *remainingCC* is 0. The loop terminates and the tester is presented with the original ranked list of statements. No non-faulty statement's suspiciousness score exceeded that of the faulty statement before the termination of the loop. Thus, thus effectiveness of our approach was identical to that of Ochiai.

Table 3.18: Example of identical effectiveness in presence of false positives

Statements	Passing tests	Failing tests
$case25 : fprintf(stdout,"comma.\n"),\ldots$	$16', 3', 5'$	f_1
$case22 : return(RSQUARE);,\ldots$	$5', 11, 12$	f_1
$\vdots$	$\vdots$	$\vdots$
$ch = getChar(\ldots),\ldots$	$0, 3', 5', 7, 8,\ldots$	f_1

Reduction in effectiveness results when some non-faulty statements' suspiciousness score exceeds that of the faulty statement before *remainingCC* turns to 0.

The above analysis shows that the effectiveness of our approach can be identical to that of the Ochiai approach both in the presence and absence of false positives. This explains the prevalence of cases where an identical effectiveness was observed.

Our approach performed better than the Ochiai approach more often for the Unix utilities compared to the Siemens suite. This is because the CC measure estimated the likelihood of coincidental correctness in the Unix utilities more accurately than in the Siemens suite. The Unix utilities are larger and have large number of paths, few of which contain the faulty statement. The statements along those paths obtain higher suspiciousness scores than other statements. Thus, the *CC* measures for the coincidentally correct test cases are considerably higher than that for the passing test cases.

3.3.3.3 RQ3 Results

To investigate RQ3, we compared the effectiveness of our approach with the *fixedCC* approach. Table 3.19 shows the distribution of the number of cases when there was an im-

provement in the effectiveness, a reduction in the effectiveness, and no change, i.e., identical effectiveness. Table 3.20 shows the distribution of amount of improvements.

Table 3.19: Distribution of the change in effectiveness

Program	Improvement (%)	Reduction (%)	Identical (%)
print_tokens	52.0	43.8	4.1
print_tokens2	57.9	30.3	11.7
replace	60.6	31.9	7.4
schedule	28.8	44.4	26.7
schedule2	40.0	25.0	35.0
tcas	36.5	44.9	18.4
tot_info	56.2	22.2	21.6
gzip	50.7	27.9	21.4
grep	74.8	20.6	4.6

Table 3.20: Distribution of percentage of improvements

Program	Max(%)	Min (%)	Median (%)	Average (%)
print_tokens	83.9	2.0	32.14	33.31
print_tokens2	96.0	1.08	77.8	61.8
replace	97.4	1.6	66.1	57.2
schedule	88.6	2.8	56.8	47.6
schedule2	88.3	2.5	18.18	19.8
tcas	91.7	3.22	33.3	47.8
tot_info	93.5	1.14	28.6	37.4
gzip	91.0	0.4	26.9	26.1
grep	99.7	0.4	70.4	64.6

For all the benchmarks except `schedule` and `tcas`, the percentage of improvement is considerably higher than the percentage of cases when our approach performed worse. The feedback driven approach also achieves up to 100% improvement over the *fixedCC* approach. We use examples from the benchmarks to illustrate some cases where our approach was better or worse.

Positive improvements primarily occurred when there were false positives. For the *fixedCC* approach, false positives occurred because of one or both of the following two reasons: (1) the number of coincidentally correct test cases is less than 60% of the total number

of passing tests, and (2) the measure inaccurately estimated the likelihood of coincidental correctness.

When false positives appeared for the *fixedCC* approach exclusively because of (1), our approach outperformed it, because our approach does not classify more coincidentally correct test cases than actually present. When false positives occurred due to (2), both our approach and the *fixedCC* approach removed passing test cases that are not coincidentally correct. This resulted in non-faulty statements to be assigned higher suspiciousness scores than the faulty statement. However, our approach detected the false positives at an early stage by checking the value of $remainingCC$, and thus, caused the suspiciousness scores of fewer non-faulty statements to exceed that of the faulty statement. We use an example in Table 3.21 to illustrate an improvement.

Table 3.21: Ranked list of statements

Statements	Passing tests	Failing tests
$junk = addstr(CCL, pat, j, MAXPAT);, \ldots$	$1', 4', 5', 8', 10, 15$	f_1
$for(k = src[*i - 1] + 1;, \ldots$	$4', 5', 8, 10, 15$	f_1
$\vdots$	$\vdots$	$\vdots$
$if((sub[i] == DITTO)), \ldots$	$16", 1', 18", 3", 12', 13", 14"$	f_1

Table 3.21 presents a ranked list of statements from the faulty version `v5` of the `replace` benchmark. A passing test case p that is classified as coincidentally correct by the feedback based approach is shown as p'. All test cases that are classified by the our approach are also classified by the *fixedCC* approach. Any passing test case p that is classified by the *fixedCC* approach but not our approach is shown as p''.

The tester inspects the statements in the first row and reports that they are correct. In the example there are 5 coincidentally correct test cases: $4, 5, 8, 10$, and 15. Because the CC measure inaccurately estimates the likelihood of coincidental correctness, our approach generates two false positives, 1 and 12. However, during this iteration, $remainingCC$ is 0 because there are already 5 test cases classified and the statements in the second row are executed by 5 test cases.

In this example, the *fixedCC* approach generates more false positives than our approach because the *fixedCC* approach classifies 11 test cases (60% of the total 19), while there are only 5 coincidentally correct test cases. Thus, false positives are generated, such as $16, 18, 3, 13$, and 14, which results in statements in the last row to be assigned a higher suspiciousness score than the faulty statement and to be inspected before the faulty statement.

Our approach performs worse than the *fixedCC* approach when there are few or no false positives in the top 60% of the passing test cases because the *fixedCC* approach classifies all the first 60% of the test cases at the same time. As a result, the suspiciousness score of the faulty statement exceeds that of more non-faulty statements when the *fixedCC* approach is used, than when our approach is used.

Based on the above examples, we infer that our approach is more robust than the *fixedCC* approach in the presence of false positives. When there are few false positives, the *fixedCC* approach performs better, while our approach performs better if there are many false positives. To investigate this hypothesis further, we calculated the percentage of false positives in the top 60% of the passing test cases in every (faulty version, test suite) pair. We plotted the variation of improvement of the feedback based approach over the *fixedCC* approach, against the percentage of false positives. Figure 3.5 shows the plot.

In Figure 3.5 each point corresponds to a (faulty version, test suite) pair. A point (x, y) represents the (faulty version, test suite) pair for which there were x% false positives in the top 60% of the test cases having the highest value of $CC(p)$, and also, our approach was y% more effective than the *fixedCC* approach. Thus, any point above the 0 value on the y-axis represents an improvement, while a point below the 0 value represents a reduction in effectiveness. It is clear from the plot that there were more improvements for the higher values of false positive percentages, while at lower values, there were more reductions in effectiveness.

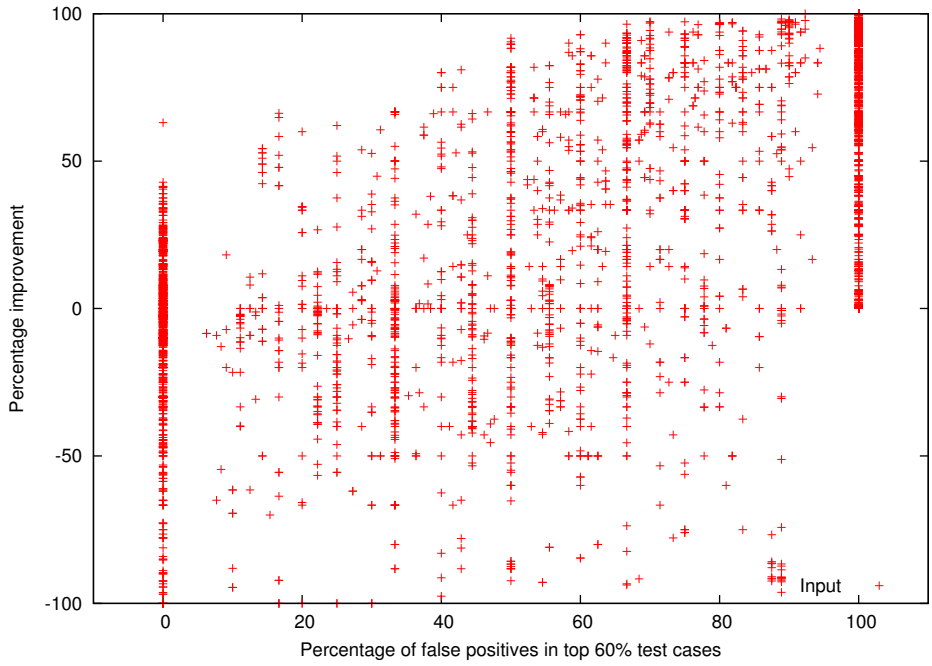


Figure 3.5: Variation of improvement with false positives

3.3.4 Threats to Validity

External: Threats to external validity stem from the nature of the benchmarks used and the size of the test suites. Siemens programs are relatively small and they have very simple control structures. Our test suites were small, containing 15-20 test cases each.

Construct: A threat to construct validity lies in the measure of effectiveness of our approach. Unlike traditional fault localization approaches that present the tester with a ranked list of all program statements, we present only a subset of the statements at a time. Thus, the tester needs to interact with our tool by reporting feedback and waiting for a new set of statements to inspect. In both the Ochiai approach and our approach, the tester determines whether or not the statements are correct. However, the additional interactions with our approach may impose more cognitive burden on the tester than the traditional approaches. Our measure of effectiveness does not address the effect of the interactions.

Another threat to construct validity results from the fact that the percentage of code examined may not correctly represent the effort to locate the faulty statement. This threat was discussed in Section 3.1.4.

3.3.5 Conclusions and Lessons Learned

Classifying and removing coincidentally correct test cases can improve the effectiveness of spectrum-based fault localization. A classification that is based on estimating the lower bound for the number of coincidentally correct test cases is more precise than classifying a fixed percentage of test cases. However, the limitation of iterative classification and removal is that it classifies test cases too conservatively, and as a result, the rank of the faulty statement is not improved by a large amount in a single iteration. This motivates us to investigate the effect of using tester feedback a limited number times instead of using it repeatedly. This also suggests that a different classification approach such as clustering, which does not require specifying the number of test cases to be classified, may be more advantageous.

Removing test cases is also not the most effective method of using the classification. As we observed, removal of test cases results in the non-faulty statements executed by any subset of the classified test cases to obtain higher suspiciousness scores than the faulty statement. As a result, fault localization effectiveness does not improve even when the actual coincidentally correct test cases are correctly classified. In the design of our family of approaches, we investigate more effective methods of using the classification.

Chapter 4

Family of Fault Localization Approaches

In this chapter, we describe our family of fault localization approaches. Section 4.1 presents an overview of the components of our family of approaches. Section 4.2 describes our classification approaches and Section 4.3 describes our approach for calculating suspiciousness scores. Section 4.4 analyzes the run-time complexity of our approaches. Section 4.5 describes an extension to the approaches to incorporate tester feedback. Section 4.6 presents the extensions for addressing multiple faults.

4.1 Family of Approaches

Figure 4.1 shows the steps in our family of approaches. The faulty program being debugged is instrumented for collecting test coverage data and the test suite is executed on the instrumented program. The program spectra of the test cases are collected. We assume that the expected output of each test case is known so that it can be determined whether each test case passed or failed.

The passing test cases are classified using a particular type of program spectra. The spectra of all the test cases and the information obtained from the classification are used to calculate a suspiciousness score for each statement. Statements are ranked in decreasing order of their suspiciousness scores.

A new fault localization approach is obtained by using a different combination of a classification approach and an approach to calculate suspiciousness scores. We use the following three classification approaches: (1) statement coverage based clustering, (2) branch coverage based clustering, and (3) function coverage based clustering. Each approach performs k-means clustering with the corresponding type of coverage spectrum. We use two approaches

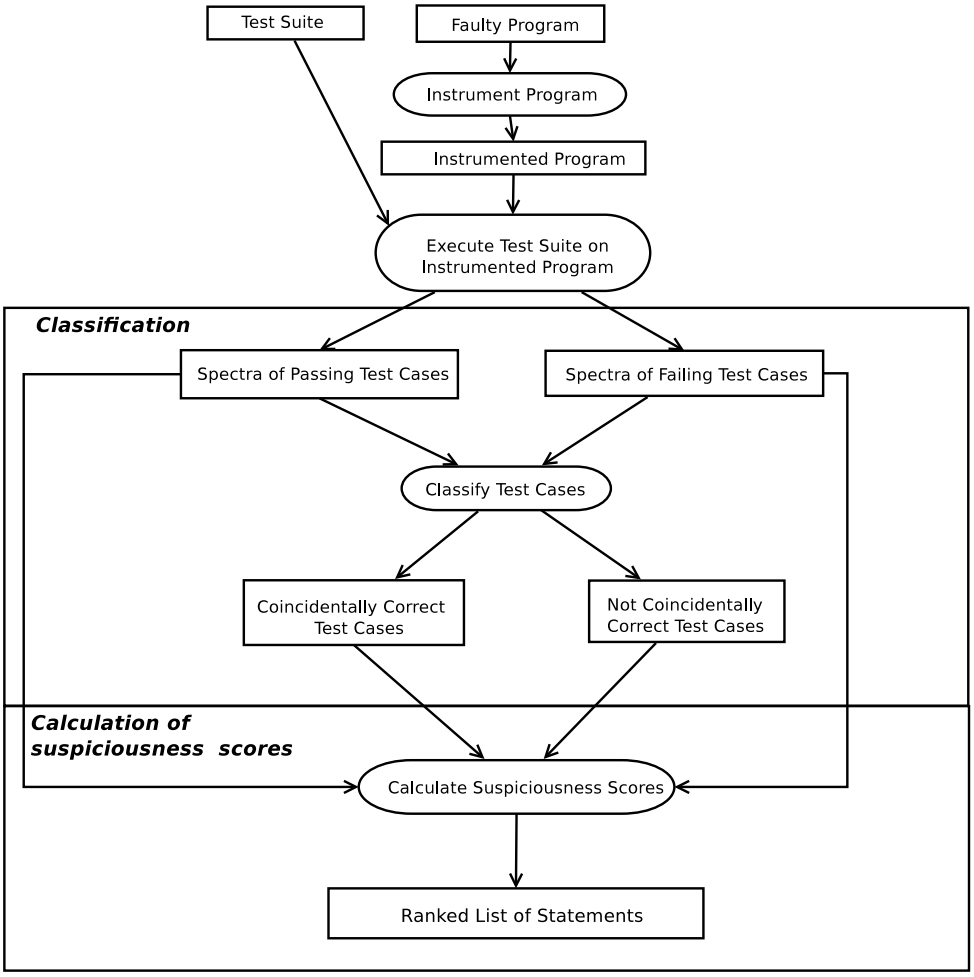


Figure 4.1: Steps in the Family of Approaches

for calculating suspiciousness scores of statements: (1) $susp_{rm}$, which calculates suspiciousness scores by removing the classified test cases and (2) $susp_J$, which calculates Jaccard similarity based suspiciousness scores and is described in Section 4.3. Table 4.1 lists the fault localization approaches of the family. For each fault localization approach, the table shows the approaches used for classification and for calculation of suspiciousness scores.

4.2 Classifying Test Cases

A fine-grained coverage spectrum can improve classification by magnifying the important differences between a passing test case and a coincidentally correct test case. A coarse-grained coverage spectrum can result in an accurate classification by abstracting away the

Table 4.1: Family of Fault Localization Approaches

Approach	**Classification**	**Calculation of Suspiciousness Scores**
StRm	Statement coverage based clustering	$susp_{rm}$
StJ	Statement coverage based clustering	$susp_J$
BrRm	Branch coverage based clustering	$susp_{rm}$
BrJ	Branch coverage based clustering	$susp_J$
FnRm	Function coverage based clustering	$susp_{rm}$
FnJ	Function coverage based clustering	$susp_J$

unimportant differences between a passing and a coincidentally correct test case. Our goal is to investigate which level of granularity results in the best classification. To illustrate the

T$_1$
```
Account ac = new Account(0);
ac.deposit(4);
ac.withdraw(1);
assertEquals(ac.balance, 3);
```

T$_2$
```
Account ac = new Account(0);
assertEquals(ac.balance, 0);
```

T$_3$
```
Account ac = new Account(0);
ac.deposit(4);
ac.withdraw(2);
assertEquals(ac.balance, 2);
```

T$_4$
```
Account ac = new Account(0);
ac.deposit(4);
assertEquals(ac.balance, 4);
```

Statement	T_1	T_2	T_3	T_4	$Ochiai(s)$
1: public Account (int balance) {	√	√	√	√	0.5
2: this.balance = balance;	√	√	√	√	0.5
3: }	√	√	√	√	0.5
4: public void deposit (int val) {	√		√	√	0.58
5: balance += val;	√		√	√	0.58
6: }	√		√	√	0.58
7: public void withdraw (int val) {	√		√		0.7
8: if (val <= balance)	√		√		0.7
9: **balance = val;**	√		√		0.7
10: else					0.0
11: balance = -1;					0.0
12: }	√		√		0.7

Figure 4.2: Example for Illustrating Classification

classification approaches, we use the example in Figure 4.2, which shows the execution of four test cases, T_1, T_2, T_3, and T_4, on a faulty program. The test cases are listed on the left. The table on the right shows for each statement, the test cases that execute it. The column $Ochiai(s)$ shows the Ochiai suspiciousness score for each statement. The program consists of a class, `Account`, with an attribute `balance`, a constructor, and two methods, `deposit` and `withdraw`. Statement 9 contains the fault because the correct statement is `balance -=`

`val`. T_1 fails for the program, while the other test cases pass. T_3 is a coincidentally correct test case.

4.2.1 Statement Coverage Based Clustering

In this approach, statements are the program elements used for constructing test vectors to perform k-means clustering. Test vectors are constructed using the statements that are executed by all the failing test cases and a percentage of passing test cases below the threshold of 70%.

Test Case \ Line No.	4	5	6	7	8	9	12	Susp.
T_1 (Failing)	1	1	1	1	1	1	1	0.7
T_2	0	0	0	0	0	0	0	0
T_3	1	1	1	1	1	1	1	0.7
T_4	1	1	1	0	0	0	0	0.25

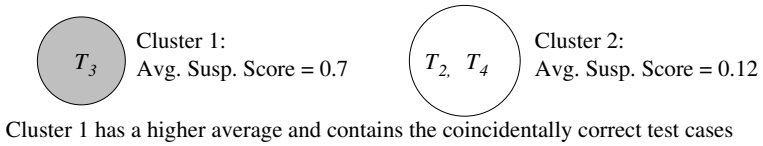


Figure 4.3: Statement Coverage Based Clustering

Figure 4.3 shows the vectors for the test cases listed in Figure 4.2. For example, the vector for T_3 is $(1, 1, 1, 1, 1, 1, 1)$. Statements 4–9 and 12 are executed by all the failing test cases and less than 70% of the passing test cases, and thus, only these statements are used for deriving the vectors. The 'Susp.' column shows the average suspiciousness scores of the statements covered by each test case.

Figure 4.3 also shows the clusters obtained by performing k-means clustering. For each cluster, it shows the average suspiciousness score of all statements covered by the test cases in the cluster. Because the average is higher for Cluster 1, it is selected to be the cluster of coincidentally correct test cases. Thus, T_3 is classified as coincidentally correct. T_3 also happens to be the only actual coincidentally correct test case in this example.

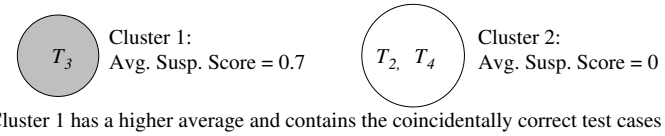
4.2.2 Branch Coverage Based Clustering

In this approach, branches are the program elements used for constructing test vectors to perform k-means clustering. Only the branches that are executed by all the failing test cases and a percentage of passing test cases below the threshold of 70% are used.

There are two branches in the program. The first branch is from statement 8 to statement 9, which is executed when the if-condition at line 8 evaluates to true. The second branch from statements 10 to 11 is executed when the if-condition at line 8 evaluates to false. Only the first branch is executed by the failing test case and less than 70% of the passing test cases. Therefore, only the first branch is selected for deriving the test vectors.

Branch	T_1	T_2	T_3	T_4	$Ochiai(br)$
8–9 (True)	√		√		0.7
10–11 (False)					0.0

Test Case \ Branch	8–9	Susp.
T_1 (Failing)	1	0.7
T_2	0	0.0
T_3	1	0.7
T_4	0	0.0



Cluster 1 has a higher average and contains the coincidentally correct test cases

Figure 4.4: Branch Coverage Based Clustering

Figure 4.4 shows the branches and their Ochiai suspiciousness scores. It shows the vectors for the test cases listed in Figure 4.2. For example, the vector of test case T_3 is (1). The clusters obtained by performing k-means clustering are shown. For each cluster, the average suspiciousness score of all the branches covered by the test cases in the cluster is presented. Because the average is higher for Cluster 1, it is selected to be the cluster of coincidentally correct test cases. Thus, T_3 is classified as coincidentally correct. T_3 also happens to be the only actual coincidentally correct test case in this example.

4.2.3 Function Coverage Based Clustering

In this approach, functions are the program elements used for constructing test vectors to perform k-means clustering. Compared to statement and branch coverage based clustering,

we use a different method for selecting the functions to be used for deriving the vectors. We select all the functions instead of selecting functions that are executed by a percentage of passing test cases below a threshold. In our studies, we observed that the function containing the faulty statement is often executed by a high percentage of passing test cases. Thus, a threshold based selection of functions can result in excluding the function containing the faulty statement. Including the function containing the faulty statement in the vectors is necessary to distinguish coincidentally correct test cases from other passing test cases because the coincidentally correct test cases will always execute that function, while other passing test cases may or may not.

Test vectors are constructed using the selected functions and then k-means clustering is performed. Once the clusters are obtained, one cluster is identified as the cluster of coincidentally correct test cases. Compared to statement and branch coverage based clustering, we use a different method for identifying the cluster, as described below.

Unlike statement and branch coverage based clustering, we cannot identify the cluster based on the average suspiciousness score of functions covered by the test cases in each cluster. The reason is that the suspiciousness score of the function containing the faulty statement was often observed to be low as the function was often executed by a high percentage of passing test cases. Therefore, instead of calculating average suspiciousness scores of functions, we calculate the average distance of every passing test case in each cluster from every failing test case and select the cluster having the lowest average.

In statement and branch coverage based clustering as well, one could have identified the cluster by calculating the average distance from all failing test cases. However, as explained in Section 4.4, calculating the average suspiciousness score of covered statements/branches is computationally less expensive than calculating the average distance from all failing test cases.

In the example in Figure 4.2, there are three functions: the constructor of the `Account` class, and the methods, `deposit` and `withdraw`. Figure 4.5 shows the vector representations of all the test cases. For example, the vector of the test case T_3 is $(1, 1, 1)$. The 'Distance'

Test Case \ Function	Account()	deposit()	withdraw()	Distance
T_1	1	1	1	1
T_2	1	0	0	1.41
T_3	1	1	1	0
T_4	1	1	0	1

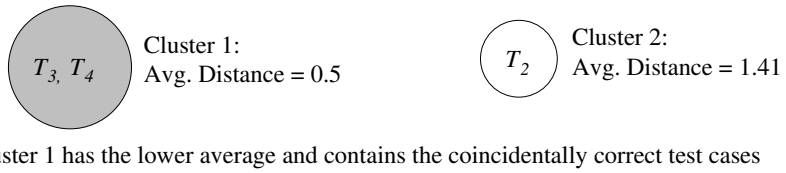


Figure 4.5: Function Coverage Based Clustering

column lists the average distance of each test case from all the failing test cases. The figure also shows the two clusters obtained, and for each cluster, the average distance of the test cases in the cluster from the failing test case. The average is lower for Cluster 1, which is identified as the cluster containing the coincidentally correct test cases. Thus, T_3 and T_4 are classified as coincidentally correct, while only T_3 is the actual coincidentally correct test case.

4.3 Calculating Suspiciousness Scores

We first present our approach for calculating suspiciousness scores. Then, we describe our adaptation of Masri et al.'s approach [37] for calculating suspiciousness scores discussed in Section 2.3. We adapted Masri et al.'s approach in order to perform a fair comparison with our approach.

Our approach for calculating suspiciousness scores retains test cases classified as coincidentally correct. The approach assumes that a single fault is being localized at a time. Suppose P, P_{Class}, and P_{CC} denote the set of all passing test cases, the set of test cases classified as coincidentally correct, and the set of actual coincidentally correct test cases, respectively. Consider a faulty statement, f, and two non-faulty statements, s_1 and s_2. Suppose that s_1 and s_2 are executed by all failing test cases. Suppose P_{s_1}, P_{s_2}, and P_f denote the set of passing test cases that execute s_1, s_2, and f, respectively.

Under the single fault assumption, f is executed by all failing test cases. $P_f = P_{CC}$ because every passing test case that executes f is coincidentally correct by definition. For any two statements that are executed by every failing test case, we state that the more suspicious statement is the one for which the set of passing test cases that execute the statement is more similar to P_{CC}. Because we do not know P_{CC}, we measure similarity with P_{Class}, which is an estimate of P_{CC}. We use the Jaccard similarity measure [24]. For any two sets A and B, the Jaccard similarity measure is defined as $\frac{|A \cap B|}{|A \cup B|}$. To every statement that is not executed by every failing test case, we assign a suspiciousness score of 0, because these statements cannot be faulty.

Based on the heuristic above, the suspiciousness score of a statement, s, is calculated by Equation 4.1. The term P_s denotes the set of passing test cases that execute the statement s. The terms F and F_s denote the set of all failing test cases in the test suite and the set of failing test cases that execute s, respectively.

$$susp_J(s) = \begin{cases} \frac{|P_s \cap P_{Class}|}{|P_s \cup P_{Class}|} & \text{if } F_s = F \\ 0 & \text{if } F_s \subset F \end{cases} \tag{4.1}$$

When multiple faults are present in the program, a faulty statement may be executed only by a subset of the failing test cases. Thus, we cannot assign a statement a suspiciousness score of 0 if it is executed by a subset of the failing test cases. An extension to our approach for calculating suspiciousness scores addressing multiple faults is presented in Section 4.6.

We also calculate suspiciousness scores using Masri et al.'s [37] approach that removes the classified test cases and recalculates scores using the remaining test cases. When we apply the approach, we calculate Ochiai suspiciousness scores instead of Tarantula suspiciousness scores because researchers have empirically established that the Ochiai approach results in more effective fault localization compared to the Tarantula approach [2]. We use the term $susp_{rm}$ to denote the Ochiai suspiciousness scores of statements calculated by removing classified test cases.

4.4 Runtime Complexity Analysis

Consider any approach in the family of fault localization approaches described in Section 4.1. Suppose that the program contains S statements and the test suite contains T test cases. Suppose E is the number of program elements. We present a runtime complexity analysis of our approach in terms of T, S, and E. We assume that the test cases have been already executed on the faulty program and the coverage information is available in the following form. For every test case, there is a list of program elements that are executed by the test case. We create two types of hash-maps from the information: one maps each test case to the set of program elements covered by the test case, and the other maps each program element to the set of passing test cases and the set of failing test cases that covers that element. Building the first hash-map requires $O(TE)$ time because the list of elements covered by each test case needs to be processed. Building the second hash-map also requires $O(TE)$ time. It is built by querying the first hash-map for every element and for every test case to check whether the test case covered the element. For every different type of program element, such as statement, branch, and function, a separate hash-map of each of the two types is maintained. Besides the hash-maps, we maintain a set of all the passing test cases and a set of all the failing test cases.

Given the above the data structures, one can find the following in constant time: (1) the set of elements executed by a given test case, (2) the set of passing test cases and the set of failing test cases that execute a given element, and (3) whether a given element is executed by a given test case. The sets are implemented using Java SDK's `HashSet`, which offers constant time performance for queries such as set membership and set size.

The first step in our approaches is to classify the test cases. The first task in classification is to derive the coverage vectors from the coverage information. Deriving the coverage vectors of all the test cases from the available coverage information requires $O(TE)$ time. It takes $O(E)$ time to select the program elements to be used for constructing the test vectors. Selecting these elements involves calculating for each element, the fractions of failing and

passing test cases executing that element, which can be performed in constant time using the constant time queries described above. Deriving the coverage vector of each test case requires $O(E)$ time. The size of a test vector is $O(E)$. To create a test vector, the value (0/1) of each of the $O(E)$ elements of the vector needs to be calculated. Calculating each value takes constant time because it takes constant time to determine if the corresponding program element was executed by the test case. Thus, creating the test vectors for all the T test cases requires $O(TE)$ time.

The second task in classification is performing k-means clustering. The k-means clustering technique requires $2^{\Omega(\sqrt{T})}$ iterations for classifying the T vectors[6]. Each iteration calculates at most $k \times T$ distances, where k is the number of clusters. In our case, $k = 2$. Each distance calculation requires $O(E)$ time because the size of each vector is $O(E)$. Thus, each iteration of the k-means clustering requires $O(TE)$ time. The entire clustering process requires $2^{\Omega(\sqrt{T})}TE$ time.

The identification of the cluster containing the coincidentally correct test cases requires $O(TE)$ time if the statement and branch coverage based clustering approaches are used, and $O(T^2E)$ time if the function coverage based clustering approach is used. For statement and branch coverage based clustering, calculating the average suspiciousness scores of covered elements for each test case requires $O(E)$ time, and calculating that average for the test cases in each of the clusters requires $O(TE)$ time. For function coverage based clustering, the distance between every passing and failing test case needs to be calculated. Thus, it requires $O(T^2)$ distance calculations, each taking $O(E)$ time because there are $O(E)$ elements in each test vector.

The second step in our fault localization approaches is to calculate suspiciousness scores for all the statements. Calculating the $susp_{rm}$ score for each statement using the failing test cases and the information obtained from the classification requires constant time. Calculating the $susp_{rm}$ scores for all statements requires $O(S)$ time.

Calculating the $susp_J$ score for each statement requires $O(T)$ time. It involves calculating the union and the intersection of sets of test cases. These operations are implemented using

the `addAll` and `retainAll` methods in the `HashSet` class, which take $O(N)$ time if the set size is $O(N)$. In our approaches, the size of the sets of test cases is at most T. Thus, calculating the $susp_J$ scores of all statements requires $O(TS)$ time. Once the suspiciousness scores are calculated, sorting the statements in decreasing order of suspiciousness scores requires $O(S \log S)$ time.

As opposed to our family of approaches, the Ochiai fault localization approach builds the data structures from the coverage information, calculates the suspiciousness score for each statement, and derives the ranked list. Thus, the runtime complexity of Ochiai is $O(TE + S \log S)$.

4.5 Extending the Approaches by Checking the Presence of Coincidentally Correct Test Cases

In the absence of coincidentally correct test cases or when there are few coincidentally correct test cases, there may not be any opportunity to improve fault localization effectiveness of the Ochiai approach by addressing coincidental correctness. In this situation, forcing classification may result in false positives, which, in turn, may cause reduction in the fault localization effectiveness. Thus, our family of approaches should be applied only when there is an opportunity for improving the fault localization effectiveness.

In Section 3.3, we presented an approach to estimate the lower bound for the number of coincidentally correct test cases by using tester feedback. We also concluded that repeatedly estimating the lower bound results in a classification that is too conservative. We present a simplified user feedback based check that is applied only once to identify whether there is an opportunity to improve fault localization effectiveness by addressing coincidental correctness. We apply our family of approaches on identifying such an opportunity. Below, we describe how the check is performed and how it is combined with our family of approaches.

The Ochiai suspiciousness scores of all the statements are calculated. The set of statements with the highest suspiciousness score is presented to the tester, who reports whether or not the faulty statement is found in the presented set of statements.

If the tester finds the faulty statement, fault localization ends. Otherwise, coincidentally correct test cases are classified by one of the approaches considering all statements. Then, $susp_J$ scores are calculated for the statements that have not been inspected by the tester.

4.6 Extending the Approaches for Multiple Faults

Mitigating the effect of coincidentally correct test cases in the presence of multiple faults requires modifications to our approach. In the presence of multiple faults, a faulty statement may not be executed by all failing test cases. We list two modifications to our approach and explain why they are needed.

- **Modification to approach for determining the need for addressing coincidental correctness:** In Section 4.5, we presented a tester feedback-based check to determine whether or not addressing coincidental correctness is required in a single fault environment. The check is based on the observation that if the faulty statement is not one of the statements having the highest suspiciousness score, then the test suite must contain coincidentally correct test cases. However, the above statement is not always true in a multiple fault environment. In the presence of multiple faults, a faulty statement may be executed by only a subset of the failing test cases and thus, may not obtain the highest suspiciousness score.

  Therefore, we modify the check by having the tester inspect all the statements that are executed by at least one failing test case but no passing test case. If the tester does not detect the faulty statement during the inspection, we infer that there exists some coincidentally correct test cases.

- **Modification to the calculation of suspiciousness scores**: Equation 4.1 assigns a suspiciousness score of 0 to any statement that is not executed by every failing test case. Because in a multiple fault environment, a faulty statement may not be executed by all failing test cases, we modified the equation to include a term that represents the fraction of failing test cases that execute a statement. The new term is formulated

based on the heuristic that the faulty statements will be executed by a high fraction of the failing test cases. Equation 4.2 is the modified equation. Te terms $failed(s)$ and $totalFailed$ respectively denote the number of failing test cases that execute the statement s and the total number of failing test cases.

$$susp_J(s) = \frac{|P_s \cap P_{Class}|}{|P_s \cup P_{Class}|} + \frac{failed(s)}{totalFailed} \tag{4.2}$$

For multiple faults, we use the same classification approaches as for a single fault. Test cases that are coincidentally correct with respect to a fault are expected to be similar to the failing test cases that fail due to that fault. Because of that similarity, we expect that the classification approaches will classify test cases that are coincidentally correct with respect to at least one fault.

Chapter 5

Implementation

We implemented a prototype framework for localizing faults by mitigating the effect of coincidentally correct test cases. The framework provides a concrete implementation of the proposed fault localization approaches, and also offers abstractions for implementing new fault localization approaches. The framework can be used both by testers to localize program faults and by researchers to perform empirical studies of fault localization effectiveness. In this chapter, we describe how the framework can be used by testers and researchers. We present the component architecture of the framework, and describe the key classes in each component. We also describe the extension mechanisms provided in the framework.

5.1 Using the Framework

A tester can use the framework to perform fault localization on a program that fails for a given test suite. We assume that the tester has access to the source code of the faulty program and the test cases in the test suite. We also assume that the tester knows the expected output of each test case so that the pass/fail outcome of the execution of each test case can be determined. Given the faulty program and the test suite, the framework applies the selected fault localization approach to calculate a list of statements sorted in decreasing order of suspiciousness scores.

A researcher can use the framework to perform studies on the effectiveness of fault localization approaches. A study involves multiple subject programs. For each subject program, there are multiple faulty versions and multiple test suites. Each test suite is executed on each faulty version and the effectiveness of the fault localization approaches is calculated for each (faulty version, test suite) pair. Therefore, the researcher needs to have access to the source code of all the faulty versions and the test suites for each subject program. The

researcher also needs to have access to the source code of the correct version of each subject program that can be executed to obtain the expected output of each test case. The expected output of a test case can be used to determine the pass/fail outcome of the test case. In order to calculate the fault localization effectiveness, the researcher needs to know additional information about each subject program, such as the size of the program and the line number of the faulty statement for each faulty version of the program. For ease of use, a researcher should be able to perform studies using multiple subject programs, faulty versions, and test suites by executing a single command.

5.1.1 Use of the Framework by Testers to Localize Faults

In the following description, we assume that the tester has already executed the test cases and determined for each test case whether the test case passed or failed. The tester needs to provide the following inputs to the framework:

- `FaultyDirName`: Name of the directory that contains the source files of the faulty program.
- `TestSuiteFileName`: Name of the file that contains the commands for executing the test cases in a test suite. Each line in the file represents one of the following:
  - A comment that starts with a '#' and is ignored.
  - A set-up command that starts with a ':' and precedes a command for executing a test case. The set-up command is run to set up the environment for the execution of the test case.
  - A command for executing a test case. Each test case in a test suite requires one command. Each test case is identified by the index of the corresponding command in the file. Thus, the i^{th} test case is the one that is executed by the i^{th} command in the file.
  - A clean-up command that starts with a ':', follows a command for executing a test case, and is run to clean up the effects of the execution of the test case.

Figure 5.1.1 shows an example file containing the commands for running two test cases for the *gzip* program. The first test case is executed by running the command `gzip /home/gzip/inputs/file9.z -d`, which runs `gzip` to decompress the file named `file9.z`. The effect of the execution of the test case is cleaned up by running `/home/gzip/scripts/cpoptd.sh`, which moves the decompressed file from its present location to another directory where output files are stored so that the framework can compare the generated output file with the correct output file.

```
#test1
gzip /home/gzip/inputs/file9.z -d
:/home/gzip/scripts/cpoptd.sh

#test2
gzip --decompress < /home/gzip/inputs/file5.z
```

Figure 5.1: Example File Containing Commands for Running Tests

- `TraceDirName`: Name of the directory where a trace file for each test case is stored. The trace file for a test case records what statements, branches, and functions are covered by the test case.

- `TestResultFileName`: Name of the file that contains information about which test case passed and which test case failed. This file contains a line for each test case that has a corresponding execution command in the file specified by `TestSuiteFileName`. The i^{th} line in the file specified by `TestResultFileName` is `P` or `F` based on whether the i^{th} test case is a passing or a failing test case.

- `ApproachName`: Name of the fault localization approach to be used. For localizing a single fault, one of the following names can be used: *StRm*, *BrRm*, *FnRm*, *StJ*, *BrJ*, *FnJ*, *Check+StJ*, *Check+BrJ*, and *Check+FnJ*. These approaches are described in Chapter 4. For localizing multiple faults, one of the following names can be used: *Check+StJ+Multi*, *Check+BrJ+Multi*, and *Check+FnJ+Multi*. These approaches are extensions to the approaches *Check+StJ*, *Check+BrJ*, and *Check+FnJ*, respectively, as described in Section 4.6.

- `ReportFileName`: Name of the file where the list of statements ranked according to their suspiciousness scores will be stored.

A Python script, `traceAndLocalize.py`, is executed by running the following command:

```
python traceAndLocalize.py <ApproachName> <FaultyDirName> <TestSuiteFileName>
 <TraceDirName> <TestResultFileName> <ReportFileName>
```

The script first instruments and compiles the instrumented faulty program. By default, it is assumed that the source directory contains a `Makefile` that uses `gcc` compiler commands with appropriate flags for instrumenting the program with the coverage tool, `gcov`. The script calls the `Makefile` to perform instrumentation and compilation. Custom mechanisms for instrumenting and compiling the instrumented programs can be provided in the script by implementing the functions, `instrument(dirName)` and `compile(dirName)`, respectively. The parameter, `dirName`, denotes the name of the directory containing the source code.

The script executes each test case on the instrumented faulty program, saves the generated traces for each test case, and then invokes the framework to perform fault localization. If one of the approaches that use tester feedback is selected, the script prints a set of statements in the standard output and prompts the tester to type "yes" or "no" to report whether or not the set of statements contains the faulty statement. The tester needs to inspect the statements and type the response in the standard input.

The output produced by the framework is stored in CSV format in the file specified by `ReportFileName`, which contains a row corresponding to every program statement. The rows are arranged in decreasing order of suspiciousness scores of the statements. For any statement, the first column shows the line number and the source code of the statement, and the second column shows the suspiciousness score of the statement. The third and the fourth column show the list of the passing test cases and that of the failing test cases that execute the statement, respectively. A list of n test cases is represented by the sequence of n numbers, where any number, i, denotes the test case that is executed by the i^{th} command in

the file named `TestSuiteFileName`. The output file can be used for localizing both single faults and multiple faults.

5.1.2 Use of the Framework by Researchers to Perform Studies

As described before, a researcher performing fault localization studies needs to use more features of the framework and also needs to provide more information to the framework. To facilitate supplying the additional information to the framework, we provide an abstract Python class, `BenchmarkConstants`, as an abstraction of all the necessary data and functions associated with any subject program. For every subject program, a researcher needs to a create concrete subclass of this class. The data specific to the subject program is provided by setting the attribute values of the subclass, and the functions specific to the subject program are provided by overriding the abstract methods of `BenchmarkConstants` in the subclass. Alternatively, the framework could have been designed to accept the subject-specific functions using overridden methods and the subject-specific data through the command line or configuration data files. However, our goal was to use a single artifact (the concrete subclass) to provide both the functions and the data.

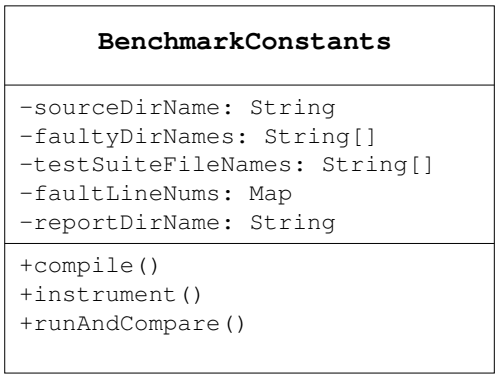


Figure 5.2: `BenchmarkConstants` class

Figure 5.2 shows the attributes and the methods of the `BenchmarkConstants` class. We describe the attributes below:

- `sourceDirName:` Name of the directory containing the source code of the correct version of the subject program.

- `faultyDirNames:` Names of the directories, where each directory contains the source code of a faulty version of the subject program. Because each faulty version containing multiple faults is constructed by combining the faults from multiple single fault versions, the name of the directory containing a multiple fault version must reflect the names of the directories containing the corresponding single fault versions. If the names of the directories for two single fault versions are 'f1' and 'f2', the name of the directory containing the multiple fault version constructed from the single fault versions in 'f1' and 'f2' must be 'f1-f2'. Thus, the name of the directory containing a multiple fault version can be used to determine the corresponding single fault versions.

- `testSuiteFileNames:` Names of the files, where each file contains the commands for executing the test cases in a test suite for the subject program. The format of each file is the same as that of the file specified by `TestSuiteFileName` described in Section 5.1.1.

- `faultyLineNums:` A Python dictionary that maps each faulty version to the line number of the faulty statement in that version. The dictionary has an entry for each single fault version. For a multiple fault version, first the corresponding single fault versions are determined given the naming convention described above, and then the line numbers of all the faulty statements in the multiple fault version are determined.

- `reportDirName:` Name of the directory where the results of the study for the subject program are stored.

The methods, `instrument` and `compile`, implement the mechanisms to instrument and compile the program, respectively. By default, these methods call `make` assuming that a `Makefile` using `gcc` compiler commands with `gcov` instrumentation flags is present in each directory containing the source code of the program. The methods can be overridden to

implement a different instrumentation and a compilation mechanism for every program. The method, `runAndCompare`, implements the comparison of the outputs of the execution of the correct program and a faulty version of the program to determine whether or not a test passed. By default, this method compares the outputs produced in the standard output. The method can be overridden, for example, to compare the generated error logs along with the outputs produced in the standard output.

A function, `runExperiment(constantsArray)`, is provided in the Python script, `traceAndLocalize.py`, to perform fault localization with multiple subject programs, faulty versions, and test suites. For example, in order to run fault localization experiments on three benchmarks, such as *flex*, *grep*, and *gzip*, a researcher first creates three concrete subclasses of `BenchmarkConstants`, namely, `FlexConstants`, `GrepConstants`, and `GzipConstants`. The code shown in Figure 5.3 is used to run experiments with the benchmarks.

```
constantsArray = [FlexConstants(), GrepConstants(), GzipConstants()]
runExperiments(constantsArray)
```

Figure 5.3: Code for Running Experiments

The `constantsArray` is an array of `BenchmarkConstants` instances. The first line in Figure 5.3 creates an instance of each of the classes, `FlexConstants`, `GrepConstants`, and `GzipConstants`, by calling the corresponding constructors, and stores the instances in `constantsArray`. The second line calls `runExperiments` using `constantsArray` as an argument. The implementation of `runExperiments` spawns a separate thread for localizing faults in each of the benchmarks. For each benchmark, every test suite specified in the `TestSuiteFileNames` attribute is executed on every faulty version located in the directories specified by `faultyDirNames` attribute. The framework is invoked to perform fault localization for every (faulty version, test suite) pair. For approaches that require feedback on the correctness of a given set of statements, the framework simulates the feedback by checking whether or not the faulty statement belongs to the set of statements.

The result of the study for the subject program is stored in several files in the directory specified by `reportDirName`. A file for each (faulty version, test suite) pair is created in the directory to store the ranked list of statements for the (faulty version, test suite) pair in the same format as described in Section 5.1.1. Additionally, another text file is created in the directory to store for each (faulty version, test suite) pair the data showing the: (1) effectiveness of the fault localization approaches, (2) actual set of coincidentally correct test cases, (3) set of classified coincidentally correct test cases, and (4) recall and precision of the classification approaches.

5.2 Framework Architecture

Figure 5.4 shows the framework components and their usage dependencies. The component `ProgramInstrumenter` is a third party component. We implemented the rest of the components, among which the component `FaultLocalizer` was implemented in Python, while all the others were implemented in Java. In the following sections, we discuss the key classes and associations of each component.

Component *FaultLocalizer*:

This component acts as a driver of the framework. It uses the third party component *ProgramInstrumenter* to collect the traces of test executions. Then it invokes other components in the framework to obtain ranked lists of statements or to perform studies of fault localization effectiveness using the collected traces. This component is implemented in the Python script, `traceAndLocalize.py`, and the Python class, `BenchmarkConstants`, and its subclasses described in Section 5.1.

Components *ProgramInstrumenter*, *CoverageInterpreter*, and *CoverageMatrix*:

These components together obtain the coverage of the test cases and convert the coverage information to a suitable representation. The *ProgramInstrumenter* component represents

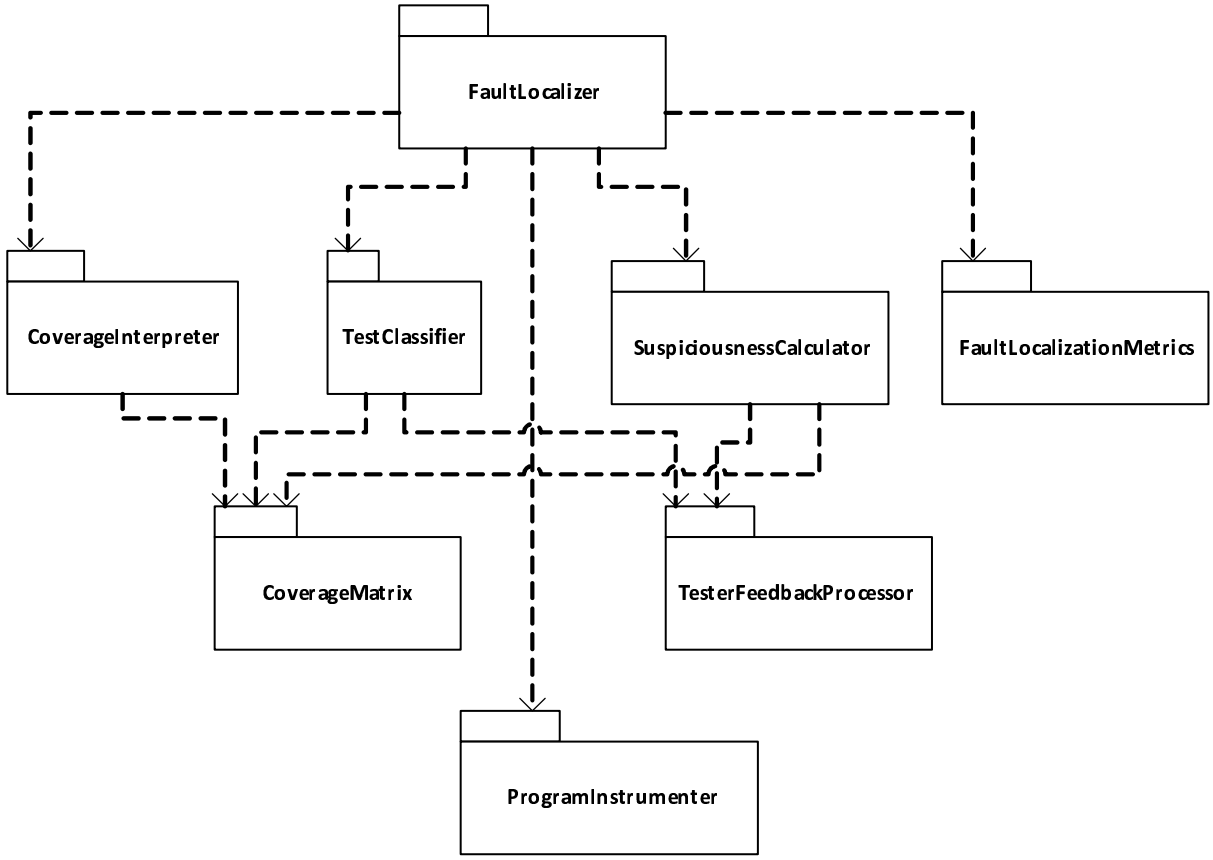


Figure 5.4: Framework Components

an external tool that can be used to instrument a program and generate coverage data for test executions.

The *CoverageInterpreter* component generates the representation of coverage information, defined by the *CoverageMatrix* component, from the trace files of test executions. A key interface in the *CoverageInterpreter* component is `CoverageAdapter`, shown in Figure 5.5. The interface defines a method, `getTestExecutionResult`. An implementation of the method creates a representation of the coverage information from the trace files. For example, the *GcovOutputAdapter* class implements the method to generate an internal representation of coverage information from the coverage data stored in the *'.gcov'* files, which are the trace files generated by the *gcov* coverage measurement tool.

The component, *CoverageMatrix*, defines a representation of the coverage information. The key classes and associations in the component are shown in Figure 5.6. The abstract

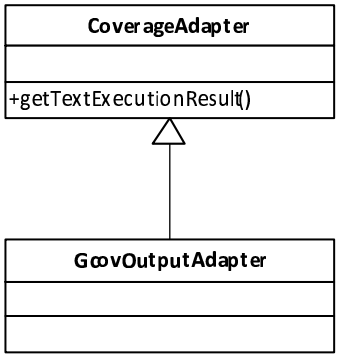


Figure 5.5: *CoverageInterpreter* Component Classes

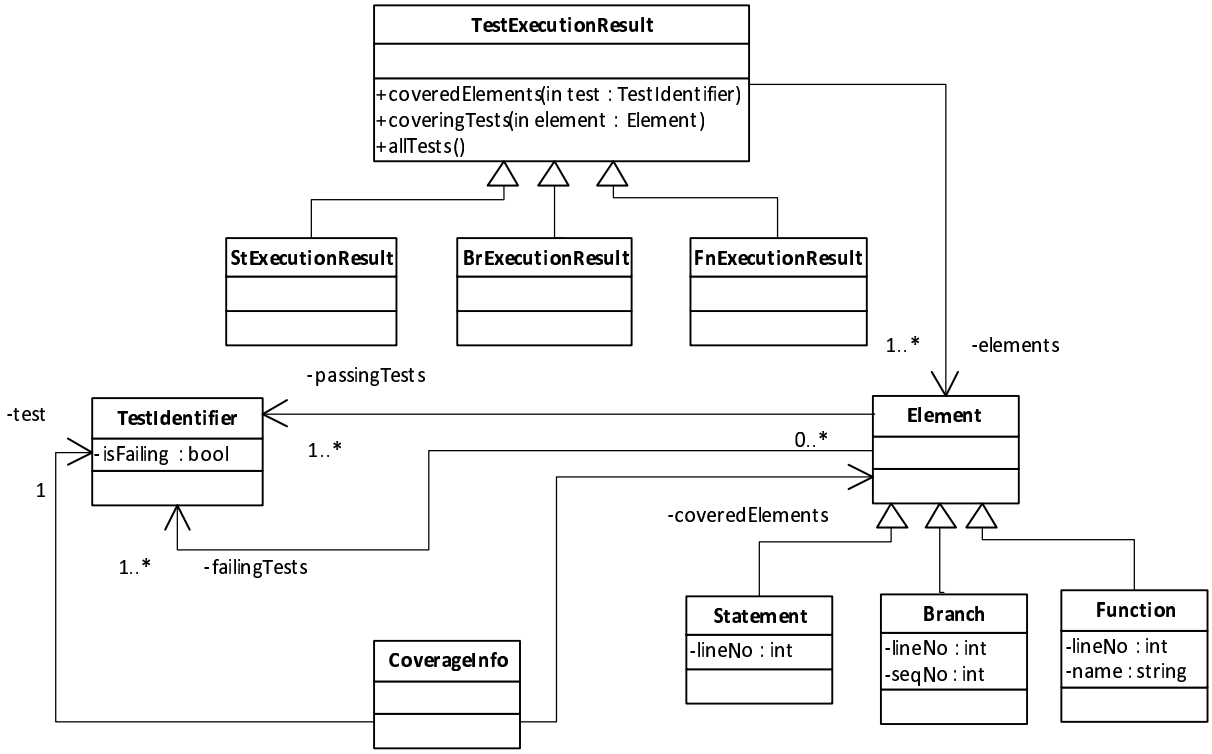


Figure 5.6: *CoverageMatrix* Component Classes

class, `Element`, represents the type of program elements. The concrete subclasses represent specific types of elements such as statements, branches, and functions. Each concrete subclass has its own set of attributes that characterize the type of program element the subclass represents. For example, the `Statement`, `Branch`, and `Function` classes have a `lineNo` attribute that represents the line number in the source code that the element is associated

with. The `Branch` class also contains a `sequenceNo` attribute that is used to identify the specific branch associated with a particular line in the source code.

A `TestIdentifier` instance represents a test case and contains a boolean attribute, `isFailing`, to denote whether the test case is failing or passing. The `TestIdentifier` class is associated with the `CoverageInfo` class, which in turn, is associated with the `Element` class. These associations are implemented as hash-maps to store the program elements covered by each test case. The `Element` class has two associations with the `TestIdentifier` class. These associations represent the set of passing test cases and a set of failing test cases that cover each element and are implemented as hash-maps.

`TestExecutionResult` is the abstract class acting as the container of all the test cases, elements and coverage information. `TestExecutionResult` provides methods for retrieving the set of passing/failing test cases, retrieving the passing/failing test cases that cover a specific element, and for retrieving the set of all the elements. Each concrete subclass of `TestExecutionResult` represents to a particular type of coverage. For example, `StExecutionResult` represents the statement coverage.

Component *TestClassifier*:

The *TestClassifier* component is responsible for classifying test cases. As Figure 5.7 shows, the component consists of the abstract class `TestClassifier` and its concrete subclasses. `TestClassifier` defines an abstract method `classifyCC()`, which returns the set of test cases classified as coincidentally correct. Each concrete subclass overrides `classifyCC()` by implementing a specific classification approach. For example, `FnBasedClustering` implements the function coverage based clustering approach for classifying coincidentally correct test cases. Each `TestClassifier` instance has a reference to a `TestExecutionResult` instance that is used for obtaining the coverage information.

Component *SuspiciousnessCalculator*:

This component provides the implementation of the approaches for calculating suspiciousness scores and also collaborates with the classes in the *TestClassifier* component to

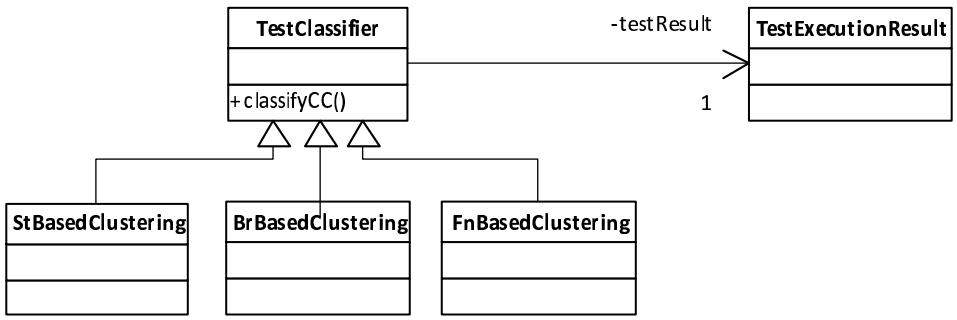


Figure 5.7: *TestClassifier* Component Classes

implement the family of fault localization approaches. Figure 5.8 shows the key classes of this component. `SuspiciousnessCalculator` is an abstract super class of the classes that implement the approaches for calculating suspiciousness scores. It defines a method `getScore(Statement st)` for calculating the suspiciousness score of a given statement, `st`. The concrete subclasses, `RemovalBased` and `JaccardBased`, override the method to implement $susp_{rm}$ and $susp_J$ approaches, respectively. `SuspiciousnessCalculator` is associated with `TestExecutionResult`, which is used to obtain the coverage information for calculating suspiciuosness scores. It is also associated with a collection of test cases that are classified as coincidentally correct by the `TestClassifier` class.

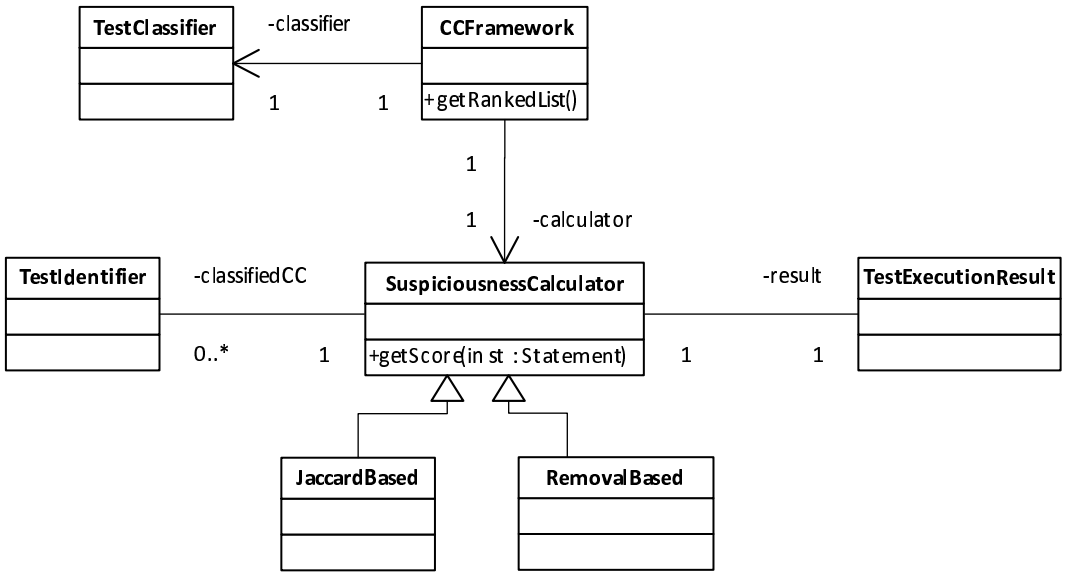


Figure 5.8: *SuspiciousnessCalculator* Component Classes

The `CCFramework` class integrates the classification approaches and the approaches of calculating suspiciousness scores. It is associated with the `TestClassifier` class and the `SuspiciousnessCalculator` class. It also has a method `getRankedList()`, which is implemented by first using the `TestClassifier` class to obtain the set of classified coincidentally correct test cases, then using the `SuspiciousnessCalculator` class to calculate the suspiciousness scores of all the statements, and finally by sorting the statements according the suspiciousness scores.

Component *TesterFeedbackProcessor*:

This component implements the functionality for collecting a feedback from the tester, processing the feedback, and incorporating the feedback in fault localization. Figure 5.9 shows the classes in the component.

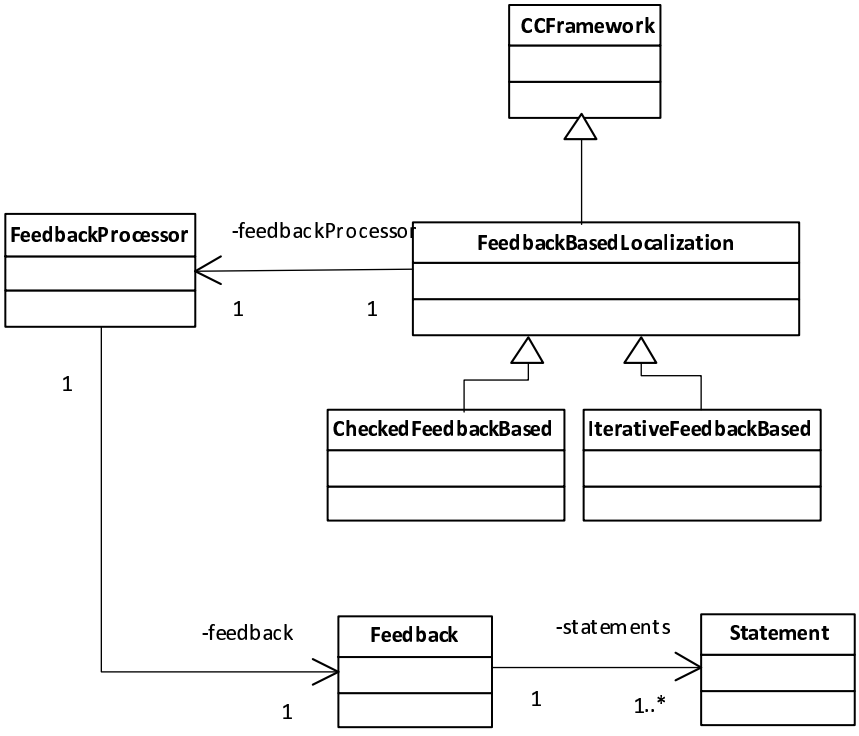


Figure 5.9: Classes of *FeedbackBasedFL* Component

`FeedbackBasedLocalization` extends the `CCFramework` class and represents fault localization approaches for mitigating the effect of coincidental correctness by incorporating

tester feedback. It is associated with the `FeedbackProcessor` class that obtains tester feedback and processes the feedback. The result of the processing is represented by the `Feedback` class. At present, the feedback is the set of statements inspected and reported correct by the tester. Thus, a `Feedback` instance contains references to multiple `Statement` instances. The subclasses of `FeedbackBasedLocalization` implement different methods of incorporating tester feedback. For example, the `CheckedFeedbackBased` class implements the approach described in Section 4.5 by obtaining the feedback once and performing fault localization if an opportunity for improving effectiveness is determined. The `IterativeFeedbackBased` class implements the approach presented in Section 3.3 by iteratively obtaining tester feedback and classifying coincidentally correct test cases.

Component *FaultLocalizationMetrics*:

This component is used in the studies performed by researchers to calculate various metrics, such as the effectiveness of fault localization approaches, and the recall and precision of classification approaches. The component is invoked by *FaultLocalizer*, which provides it with information about a faulty program, such as the program size and the line number of the faulty statement. *FaultLocalizationMetrics* uses *SuspiciousnessCalculator* to obtain a ranked list of statements and calculates fault localization effectiveness using the ranked list, the line number of the faulty statement, and the program size. In order to calculate recall and precision, *FaultLocalizationMetrics* uses *TestClassifier* to obtain the classified coincidentally correct test cases, and uses *CoverageMatrix* and the line number of the faulty statement to determine the actual coincidentally correct test cases.

5.3 Extending the Framework

The framework can be easily extended to support: (1) a new classification approach that uses an already supported coverage spectrum, (2) a new approach for calculating suspiciousness scores, (3) a new type of coverage spectrum to support classification using the coverage

spectrum, and (4) other programming languages. In the following subsections, we discuss the steps in implementing the extensions.

5.3.1 Adding a New Classification Approach

For this extension, the only required step is to create a concrete subclass of the `TestClassifier` class that implements the new classification approach. Any existing approach for calculating suspiciousness scores can be combined with the new classification approach to perform fault localization. For example, the lines of code shown in Figure 5.10 perform fault localization by combining the new classification approach implemented in the `NewClassifier` class with the Jaccard similarity based approach for calculating suspiciousness scores.

```
TestClassifier newClassifier = new NewClassifier(testExecutionResult);
SuspiciousnessCalculator jaccardBased = new JaccardBased();
CCFramework framework = new CCFramework(newClassifier, jaccardBased);
framework.getRankedList();
```

Figure 5.10: Code for Adding a New Classification Approach

5.3.2 Adding a New Approach for Calculating Suspiciousness Scores

For this extension, first, a concrete subclass of `SuspiciousnessCalculator` must be created to implement the new approach for calculating suspiciousness scores. Then, the new approach can be combined with any classification approach to perform fault localization. For example, the lines of code shown in Figure 5.11 perform fault localization using the function coverage based classification with the new approach for calculating suspiciousness scores implemented in the class `NewCalculator`.

5.3.3 Adding a New Coverage Type

Suppose that the framework needs to be extended to support the classification of test cases based on def-use pair coverage. In the *CoverageMatrix* component, a new concrete

```
TestClassifier classifier = new FnBasedClustering(testExecutionResult);
SuspiciousnessCalculator calculator = new NewCalculator();
CCFramework framework = new CCFramework(classifier, calculator);
framework.getRankedList();
```

Figure 5.11: Code for Adding a New Approach for Calculating Suspiciousness Scores

subclass, `DUPair`, of the `Element` class, needs to be created to represent the def-use pairs. Then, a new concrete subclass, `DUPairExecutionResult`, of `TestExecutionResult`, needs to be created to represent information about the coverage of def-use pairs. If collecting def-use pair coverage does not require using a new coverage tool, a new method in the subclass of `CoverageAdapter` corresponding to the coverage tool should be implemented for creating `DUPairExecutionResult` instances from the trace files of the test cases. If a new coverage tool is used for collecting def-use pair coverage, a new concrete subclass of `CoverageAdapter` needs to be created to process the output of the coverage tool.

Then, the classification approach based on def-use pair coverage information needs be implemented as a subclass, `DUPairBasedClassification`, of the `TestClassifier` class. To implement a fault localization approach that uses the new classification approach, `DUPair-BasedClassification` can be used with any existing class that implements an approach for calculating suspiciousness scores. The required code will be similar to the code shown in Figure 5.10.

5.3.4 Adding a New Programming Language

Adding a new programming language requires implementing a new subclass for `Coverage-Adapter` to process the output of a coverage tool that collects the coverage data obtained from the execution of programs in the new language. The rest of the framework is generic with respect to programming languages. Thus, once the new `CoverageAdapter` subclass creates an appropriate `TestExecutionResult` instance, the framework classes can be used to perform fault localization for the program in the new language.

Chapter 6

Evaluation

In this chapter, we present three studies of our family of approaches. Section 6.1 presents the first study, in which we compared the classification approaches for their precision and recall, and the fault localization approaches for their effectiveness. The fault localization approaches studied include (1) our family of fault localization approaches described in Section 4.1, (2) approaches for calculating suspiciousness scores, described in Section 4.3, combined with perfect classification, and (3) our fault localization approaches combined with the check for the presence of coincidentally correct test cases described in Section 4.5.

Section 6.2 presents the second study, in which we assessed the effect of several confounding variables on the effectiveness of our family of approaches: (1) percentage of passing test cases that are coincidentally correct, (2) test suite type, such as random, statement adequate, and branch adequate, and (3) size of test suites.

Section 6.3 presents the results of the third study, in which we evaluated the fault localization effectiveness of our approaches in the presence of two faults.

6.1 Evaluation of approaches for classification and calculation of suspiciousness scores

This group of evaluations are based on the following research questions:

RQ1: What are the recall and precision of the classification approaches?

Recall is defined as the ratio of the number of coincidentally correct test cases classified correctly to the total number of actual coincidentally correct test cases. Suppose that P_{Class} and P_{CC} denote the set of test cases classified to be coincidentally correct and the set of actual coincidentally correct test cases, respectively. Then recall is measured by the expression $\frac{|P_{Class} \cap P_{CC}|}{|P_{CC}|}$.

Precision is defined as the ratio of the number of test cases correctly classified as coincidentally correct to the total number of test cases classified as coincidentally correct. Thus, it is measured by the expression $\frac{|P_{Class} \cap P_{CC}|}{|P_{Class}|}$.

Fault localization effectiveness depends on the recall and precision of the classification of coincidentally correct test cases. Thus, evaluating the recall and precision of the classification approaches will facilitate the understanding of the effectiveness of the fault localization approaches.

RQ2: When coincidentally correct test cases are present, how does the fault localization effectiveness of the approaches compare with each other and with the Ochiai approach that does not address coincidental correctness?

We compare the approaches with the Ochiai approach because Abreu et al. [2] showed that Ochiai is the best fault localization approach that does not address coincidental correctness. Effectiveness of a fault localization approach is measured as follows. We measure the average number of statements a tester using the approach needs to examine in the ranked list of statements before finding the faulty statement. The statements to be examined include all the statements that have a higher suspiciousness score than the faulty statement, and on average, half the statements that share the same suspiciousness score as the faulty statement. We divide the number of statements examined by the total number of executable program statements to calculate the percentage of program statements examined. Higher is the percentage, lower is the effectiveness.

Inspecting each statement and determining whether or not it is faulty may require a tester to perform additional tasks such as setting breakpoints and analyzing execution states. Our measure does not account for the additional effort for inspecting each statement. To compare the effectiveness of each of our approaches with that of the Ochiai approach, we measure the difference between the effectiveness of the Ochiai approach and that of the approach.

The effectiveness of each approach is limited by the recall and precision of the classification approach used. Thus, the effectiveness that results when all coincidentally correct

test cases are correctly classified defines an upper bound for the effectiveness of each of our approaches. We calculate two types of upper bounds, *OptRm* and *OptJ*, which denote the effectiveness resulting from correctly classifying all coincidentally correct test cases and localizing the fault using $susp_{rm}$ scores and $susp_J$ scores, respectively.

RQ3: How does the fault localization effectiveness of the approaches compare with each other and with the Ochiai approach in the absence of coincidentally correct test cases?

When there are no coincidentally correct test cases, fault localization effectiveness cannot be further improved by addressing coincidentally correct test cases. Applying our approaches may even reduce the effectiveness if passing test cases are incorrectly classified as coincidentally correct.

RQ4: How does the effectiveness of the approaches combined with check compare with each other?

The goal of this research question is to investigate whether or not combining the approaches with the check for the presence of coincidentally correct test cases improves the fault localization effectiveness.

6.1.1 Benchmarks and Test Suites

We used three benchmarks obtained from Software-artifact Infrastructure Repository [1]. They are the Unix utilities, *flex*, *grep*, and *gzip*, with several faulty versions of each program. Table 6.1 shows the benchmarks together with their description, size, and the number of faulty versions. We used only those faulty versions for which there was at least one failing test case.

Each faulty version contains exactly one fault. Each benchmark comes with a large pool of test cases. For each faulty version, we created 50 test suites by randomly selecting between 1–4 failing test cases and 15 passing test cases from the pool. We kept the ratio of the number

Table 6.1: Benchmark Characteristics

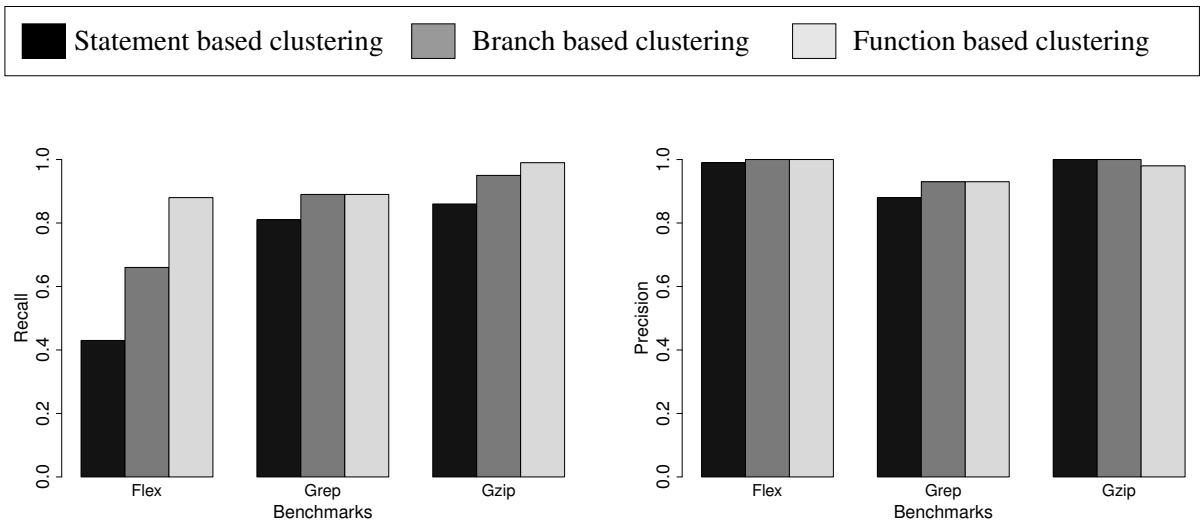
Program	Description	LOC	#Faulty Versions
gzip-1.1.2	Compression utility	1743	16
gzip-1.2.2		2045	7
gzip-1.2.3		1889	10
gzip-1.2.4		1918	12
gzip-1.3		2018	14
grep-2.2	Text search utility	3197	18
grep-2.3		3363	8
grep-2.4		3445	18
grep-2.4.1		3465	12
flex	Lexical analyzer generator	10459	6

of failing test cases to the number of passing test cases low. We assume that our approaches are applied when all the faults that are easy to detect have already been removed. Thus, test failure is less frequent than test success.

6.1.2 RQ1: Recall and Precision of the Classification Approaches

We executed each faulty version with each test suite. For each *(faulty version, test suite)* pair, we classified the test cases using the clustering approaches and then calculated the recall and the precision. We only considered the *(faulty version, test suite)* pairs having at least one coincidentally correct test case because recall is undefined in the absence of coincidentally correct test cases. Figure 6.1(a) and 6.1(b) show the average recall and precision of each clustering approach and each benchmark, respectively.

The recall values of the coarser grained coverage based clustering approaches were higher. Function coverage based clustering approach had the highest recall, followed by the branch coverage based and the statement coverage based clustering approaches. Recall increases if more actual coincidentally correct test cases are correctly classified. Coincidentally correct test cases that were similar to the failing test cases were classified correctly by all the classification approaches. As explained in Section 2.3, the approaches are expected to correctly classify the passing test cases that are similar to the failing test cases. However, for the benchmarks we studied, there were also coincidentally correct test cases that were



(b) Recall (c) Precision

Figure 6.1: Recall and Precision of the Classification Approaches for Random Test Suites

not similar to the failing test cases. These test cases were more often correctly classified by a coarse-grained coverage based approach than a fine-grained coverage based approach. A coarse-grained coverage based approach abstracts away the differences between two test cases by a greater extent than a fine grained coverage based approach. For example, the function coverage based approach abstracts away the differences in the branches covered by two test cases within the same function. Thus, a coincidentally correct test case dissimilar to the failing test cases with respect to branch coverage often was similar to the failing test cases with respect to function coverage. As a result, the branch coverage based clustering approach failed to classify such a test case as coincidentally correct, while the function coverage based clustering approach correctly classified the same test case, leading to a higher recall. A similar argument applies with statement coverage based clustering and branch coverage based clustering approaches.

All the approaches had lower recall values for *flex* compared to the other two benchmarks. For *flex*, there was a prevalence of faults for which the coincidentally correct test cases were not similar to the failing test cases. Such coincidentally correct test cases were not always correctly classified because the classification approaches are expected to classify coincidentally correct test cases that are similar to the failing test cases. For *flex*, many faults lie in the

`main` function or in functions directly called from `main`, such as the `flexinit` function that initializes *flex*, or the `readin()` function that reads the input file containing the specification of the language for which a lexical analyzer is to be generated. Many coincidentally correct test cases executed these faults but then executed paths that were different from the failing test cases. Our approaches failed to classify these test cases as coincidentally correct.

For *gzip*, the recall was higher than that for *flex* because there were fewer coincidentally correct test cases that were dissimilar to the failing test cases. The test cases in *gzip* were of two types: those that performed compression and those that performed decompression. These two types of test cases always formed their own clusters because the test cases of one type covered many different program elements compared to the test cases of the other type. Suppose that a faulty statement lay in the compression code. For this fault, (1) the failing test cases and the actual coincidentally correct test cases also performed compression and (2) the cluster containing the test cases that performed compression was identified as coincidentally correct because these test cases were more similar to the failing test cases. Thus, all coincidentally correct test cases were classified correctly, resulting in a recall of 1. For the same reason, when a fault was in the decompression code, the recall was 1. For *gzip*, most faults were either in the compression code or in the decompression code. Thus, a recall of 1 was obtained when the coincidentally correct test cases for those faults were classified.

For *grep*, most coincidentally correct test cases were similar to the failing test cases as well. The *grep* program first compiles a pattern then matches it with the input file. Test cases that used legal patterns executed the code for both pattern compilation and pattern matching. Test cases that used illegal patterns did not execute the pattern matching code due to unsuccessful pattern compilation. These two types of test cases always formed their own clusters due to their difference in the coverage of the pattern matching code. Most faults in *grep* were in the pattern matching code. In order to reach the faulty statement in the pattern matching code, a test case must have used a legal pattern. Thus, for these faults, both the failing and the actual coincidentally correct test cases used legal patterns. The cluster of

test cases that used legal patterns was also identified as the cluster of coincidentally correct test cases. Thus, a recall of 1 was obtained.

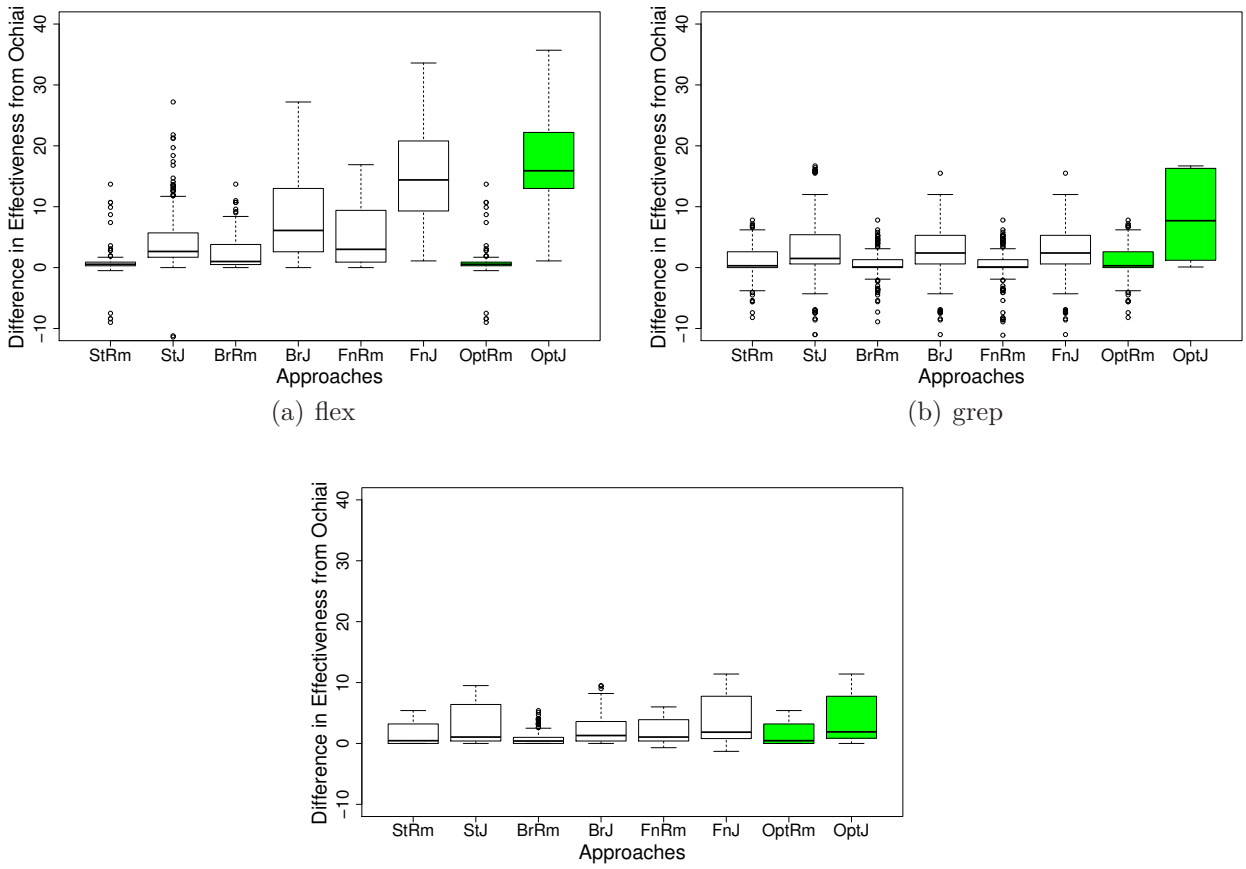
Precision decreases when a test case that is not coincidentally correct is classified as coincidentally correct. For our approaches, this happens if a passing test case is not coincidentally correct even though it is similar to the failing test cases. We observed that all our approaches resulted in a high precision for all benchmarks because passing test cases that are similar to the failing test cases were coincidentally correct in most cases.

6.1.3 RQ2: Fault Localization Effectiveness in the Presence of Coincidentally Correct Test Cases

Figure 6.2 shows the box-plots of the difference in effectiveness of the approaches from the Ochiai approach for all *(faulty version, test suite)* pairs for which there was at least one coincidentally correct test case. For *flex*, *grep*, and *gzip*, there were 254, 151, and 209 such pairs, respectively. The figure also shows the boxplots of the difference in effectiveness of *OptRm* and *OptJ* from the Ochiai approach. Each box-plot corresponds to a benchmark. The x-axis of each box-plot represents the fault localization approaches. The y-axis represents the difference in effectiveness from the Ochiai approach.

Most box plots lay above the value of 0. Thus, all approaches were more effective than the Ochiai approach when coincidentally correct test cases were present. The median of the difference in effectiveness varied from 0% to 15%.

Approaches using function coverage based clustering were more effective than the approaches using branch coverage based clustering, which, in turn, were more effective than the approaches using statement coverage based clustering. Function coverage based clustering resulted in the highest recall while achieving similar precision values compared to statement and branch coverage based clustering. A classification with a low recall value means that only a subset of all coincidentally correct test cases were classified correctly. As a result, non-faulty statements executed by the classified subset of the coincidentally correct test cases obtain a suspiciousness score equal or higher to that of the faulty statement.



(a) flex

(b) grep

(c) gzip

Figure 6.2: Difference in Effectiveness from the Ochiai approach in the Presence of Coincidentally Correct Test Cases

Thus, a lower recall results in lower effectiveness. Therefore, function coverage clustering based approaches were more effective than the other approaches. For *grep*, the effectiveness of the function coverage clustering based approaches were not appreciably higher than that of the other approaches because the recall of function coverage based clustering was not appreciably higher than that of the other classification approaches.

Based on the pairwise Student's t-test, for the benchmarks *flex* and *gzip*, approaches using function coverage based clustering were significantly more effective than those using branch and statement coverage based clustering, within 95% confidence interval. However, such statistical significance was not observed for *grep*.

Approaches using $susp_J$ were more effective than the ones using $susp_{rm}$. We explain the reason assuming a perfect classification. For such a classification, $susp_J$ assigned the faulty statement and the non-faulty statements executed by every coincidentally correct test case the highest suspiciousness score. However, $susp_{rm}$ assigned the faulty statement and non-faulty statements executed by any subset of the coincidentally correct test cases the highest suspiciousness score. Thus, while both $susp_{rm}$ and $susp_J$ assigned the faulty statement the highest score, $susp_{rm}$ assigned more non-faulty statements the highest score compared to $susp_J$ and resulted in a lower effectiveness. Based on the pairwise Student's t-test, approaches using $susp_J$ were significantly more effective than those using $susp_{rm}$, within 95% confidence interval, for all the benchmarks.

An example of a *gzip* fault shown in Table 6.2 illustrates why $susp_J$ based approaches performed better than $susp_{rm}$ based approaches. The table shows two statements, the passing and failing test cases that execute the statements, and the $susp_{rm}$ and $susp_J$ scores assigned to the statements, assuming that both the recall and the precision of the classification is 1. The statement shown in bold is faulty. Test cases 36, 56, 86, 126, and 176 are coincidentally correct. The statement in the first row is executed by a subset of coincidentally correct test cases. When all the coincidentally correct test cases are removed to calculate $susp_{rm}$, neither statement is executed by any passing test case. Thus, they both have a suspiciousness score of 1. However, when $susp_J$ is calculated, the faulty statement obtains a higher score than the non-faulty statement.

Table 6.2: Comparison of $susp_{rm}$ and $susp_J$

Statements	**Passing Tests**	**Failing Tests**	**susp$_{rm}$**	**susp$_J$**
`if (len >= nice_match) break;`	56, 86, 126, 176	13, 113	1.0	0.8
`if (compr_level >= 3)`	36, 56, 86, 126, 176	13, 113	1.0	1.0

The *FnJ* approach resulted in the highest effectiveness for all the benchmarks, among which the highest effectiveness was observed for *flex*. This can be explained by comparing the *OptJ* effectiveness for the three benchmarks. The *OptJ* effectiveness for *flex* was higher

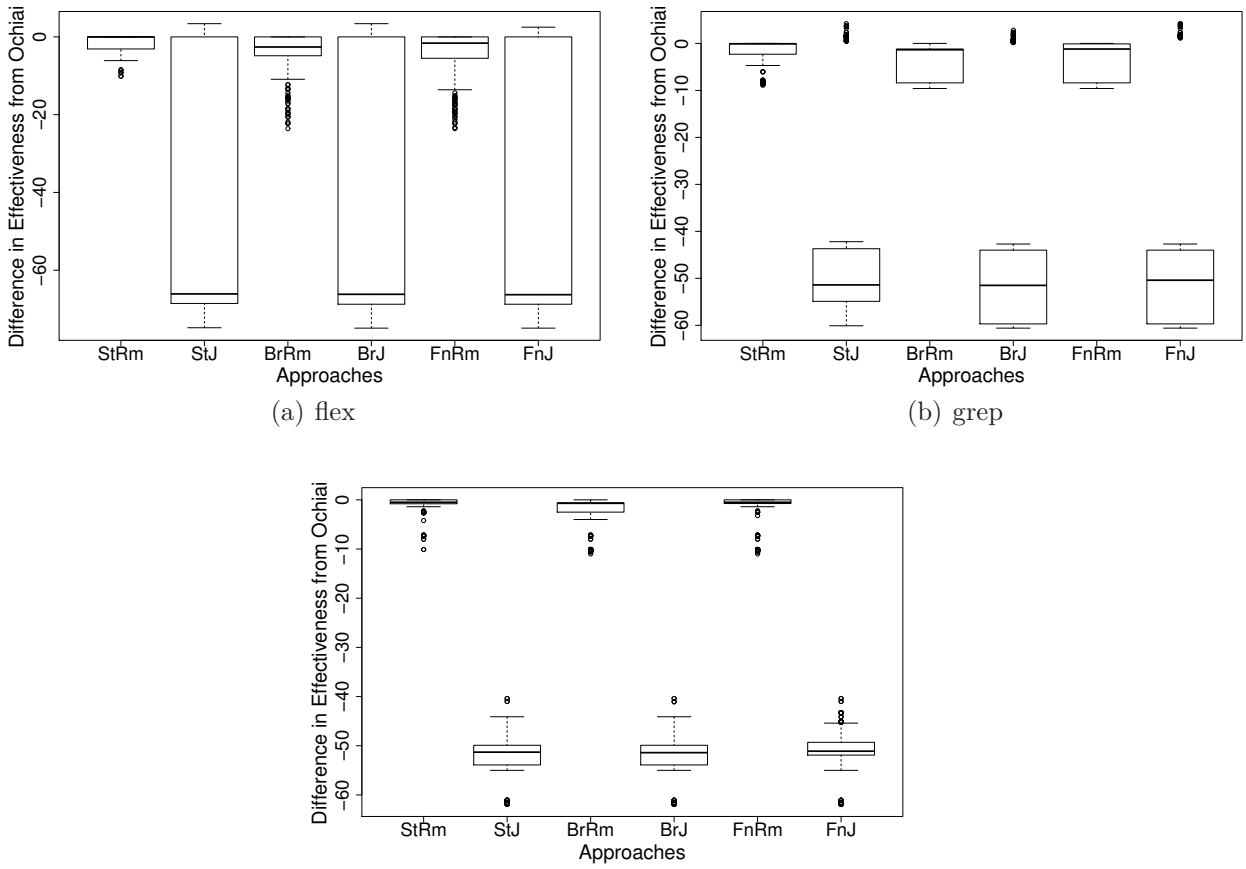
than that for the other benchmarks. This is because most *flex* faults were located close to the program entry, such as in the `main` function or functions directly called by `main`. Coincidentally correct test cases that executed these faults covered different paths in the remainder of the program. Thus, there were many non-faulty statements that were covered by some but not all coincidentally correct test cases. The suspiciousness score of the faulty statement assigned by the *OptJ* approach surpassed that of these non-faulty statements because statements that are executed by every coincidentally correct test case obtained the highest score in *OptJ*. For *grep* and *gzip*, the effectiveness of the *OptJ* approach was less than that for *flex* because the faulty statements were not close to the program entry. The *FnJ* approach performed nearly as effectively as *OptJ* because the recall and precision of function coverage based clustering were both close to 1. Thus, the effectiveness of *FnJ* was higher for *flex* compared to the other two benchmarks.

We also observed that for *flex*, the effectiveness of *OptRm* was lower than that of the other $susp_{rm}$ based approaches, such as *StRm*, even though *OptRm* assumes perfect classification. For *flex*, the number of coincidentally correct test cases in the test suites was high. In the *OptRm* approach, all the actual coincidentally correct test cases were removed and the resulting loss of information caused a reduction in effectiveness. However, for the $susp_{rm}$ based approaches, such as *StRm*, because the classification usually achieved an average precision close to 1 and an average recall of 0.4, only a subset of the actual coincidentally correct test cases was removed. Thus, the resulting loss in information was lower than that for *OptRm* and consequently, the effectiveness was higher.

6.1.4 RQ3: Fault Localization Effectiveness in the Absence of Coincidentally Correct Test Cases

We followed the same method as in the study of **RQ2** to obtain the box plots of the difference in effectiveness values for **RQ3**. For **RQ3**, we only considered the *(faulty version, test suite)* pairs for test suites that did not contain any coincidentally correct test case for the faulty version. For *flex*, *grep*, and *gzip*, there were 248, 251, and 389 such pairs, respectively.

Figure 6.3 shows the box plots. We do not show plots for *OptJ* and *OptRm* because it is not meaningful to measure the upper bound of effectiveness for **RQ3**.



(a) flex

(b) grep

(c) gzip

Figure 6.3: Difference in Effectiveness from to the Ochiai approach in the Absence of Coincidentally Correct Test Cases

All the approaches had a lower fault localization effectiveness compared to the Ochiai approach. This is expected because classifying coincidentally correct test cases when there is none can only result in false positives. Calculating suspiciousness scores using the classification results in non-faulty statements executed by these false positives to obtain higher suspiciousness scores compared to the faulty statement, thereby reducing fault localization effectiveness.

Approaches based on $susp_J$ reduce the effectiveness by a larger amount compared to the approaches based on $susp_{rm}$. In the absence of coincidentally correct test cases, $susp_J$

assigns the faulty statement a 0 suspiciousness score because P_s becomes ϕ in Equation 4.1 for the faulty statement. Thus, $susp_J$ causes the faulty statement to obtain the lowest suspiciousness score.

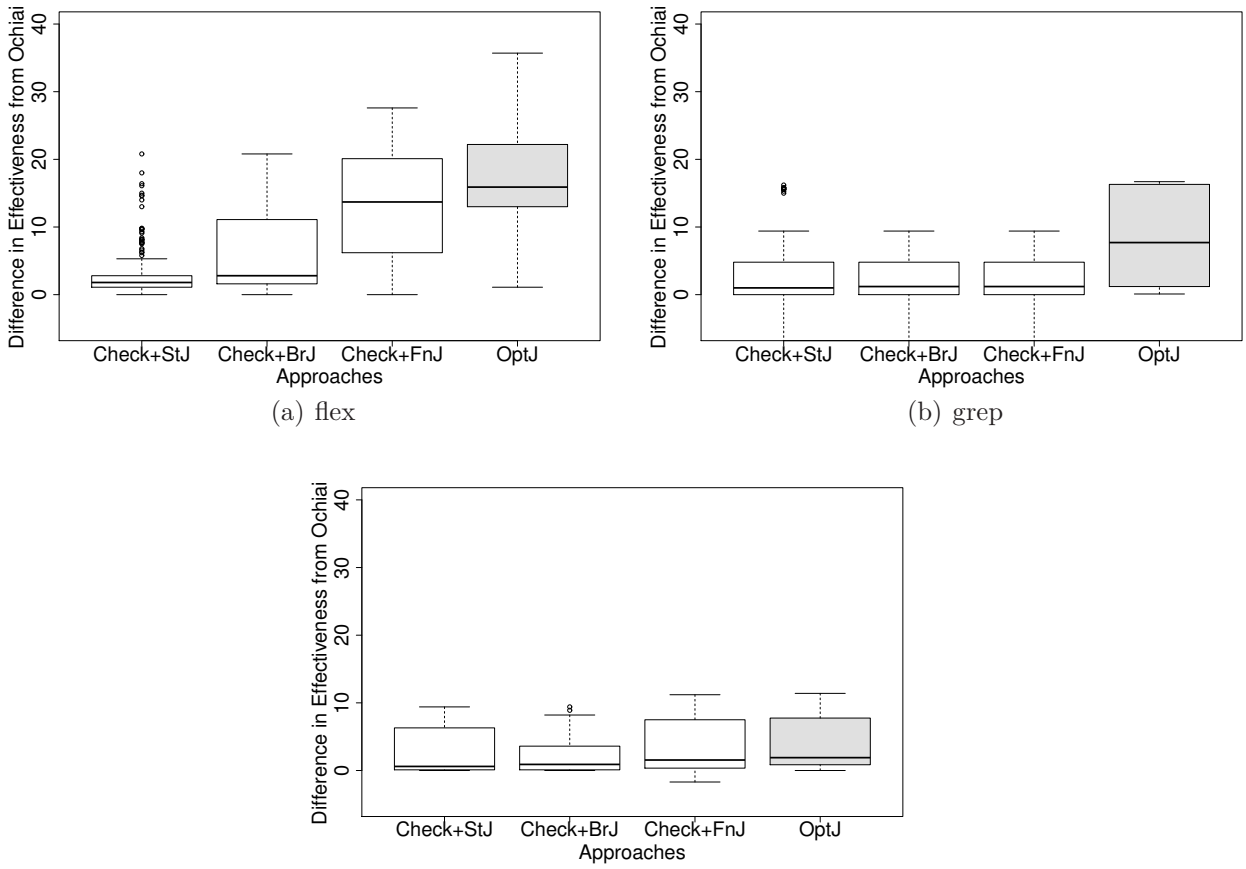
For $susp_{rm}$, the suspiciousness score of the faulty statement, f, remains 1, as $passed(f) = 0$ and $failed(f) = totFailed$ in Equation 1.1, after removing the test cases. However, after removing the test cases, the non-faulty statements that are executed by all failing test cases and the removed passing test cases also obtain a suspiciousness score of 1. Thus, $susp_{rm}$ causes these non-faulty statements to obtain the same suspiciousness score as the faulty statement, thereby reducing effectiveness. However, the extent of this reduction is lower than that for $susp_J$.

6.1.5 RQ4: Fault Localization Effectiveness with Tester Feedback Based Check

To study **RQ4**, we only considered the combination of the check with each of the three classification approaches and the Jaccard similarity based approach for calculating suspiciousness scores. We did not use $susp_{rm}$ scores as $susp_J$ scores have been shown to result in more effective fault localization compared to $susp_{rm}$ scores. This results in the following three approaches: *Check+StJ*, *Check+BrJ*, and *Check+FnJ*, which denote the check combined with statement, branch, and function coverage based clustering, respectively, along with the Jaccard similarity based suspiciousness score calculation.

Figure 6.4 shows the effectiveness of the three approaches for all *(faulty version, test suite)* pairs, which include both the pairs with coincidentally correct tests as well as those without coincidentally correct tests.

As the figures show, the lower-quartile of all boxes are greater than or equal to 0. Due to the introduction of the check for the presence of coincidentally correct tests, (1) the effectiveness was not reduced for the *(faulty version, test suite)* pairs without coincidentally correct test cases and (2) the effectiveness was improved for the pairs with coincidentally correct test cases. Approaches based on function coverage clustering performed better than the other



(a) flex

(b) grep

(c) gzip

Figure 6.4: Difference in Effectiveness of the Approaches Applied with Check from Ochiai

types of clustering based approaches. The effectiveness for *flex* was higher compared to the other two benchmarks. These results are consistent with **RQ2** results.

6.2 Evaluation Addressing Confounding Variables

We present further evaluation of *Check+StJ*, *Check+BrJ*, and *Check+FnJ* approaches to address the effects of three confounding variables. We state the following research questions to define our goals.

RQ5: How does the effectiveness of each approach vary with varying levels of percentage of the passing test cases that are actually coincidentally correct?

Wang et al. [48] showed that if a higher percentage of the passing test cases is coincidentally correct, the fault localization effectiveness of the Ochiai approach is lowered. Thus, with a higher percentage, there is more opportunity to improve the fault localization effectiveness by applying our approaches.

RQ6: How does the effectiveness of each approach vary for different types of test suites, such as random, statement coverage adequate, and branch coverage adequate?

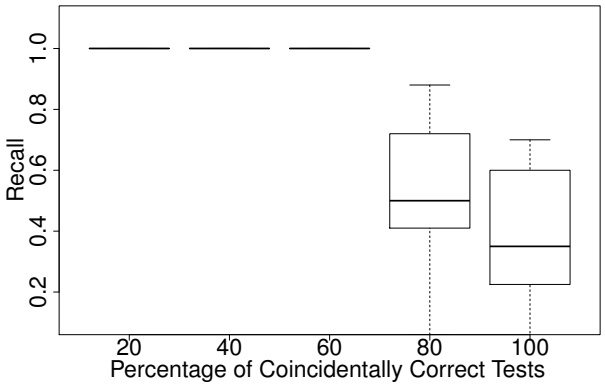
We consider the type of test suites to be a confounding factor because previous studies have shown that fault localization effectiveness varies with the type of the test suites. We consider three types of test suites: (1) random, (2) statement coverage adequate, and (3) branch coverage adequate. These types of test suites are commonly used in industry and research.

RQ7: How does the effectiveness of each approach vary for test suites of different sizes?

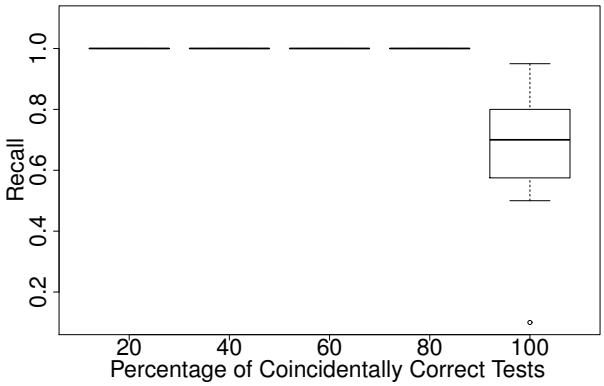
A test suite containing more test cases can achieve higher coverage, which can provide more information for fault localization. We suspect that when large test suites are used, fault localization effectiveness using the Ochiai approach may already be high, leaving less opportunity for improving the effectiveness by addressing coincidental correctness.

6.2.1 RQ5: Effect of Percentage of Coincidentally Correct Test Cases

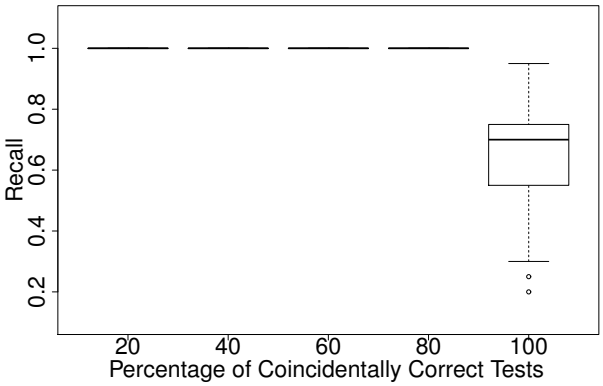
We evaluated our approaches with test suites containing five levels of percentage of coincidentally correct test cases: 20%, 40%, 60%, 80%, and 100%. To create a test suite containing k% coincidentally correct test cases, we randomly selected 1–5 failing test cases and 20 passing test cases among which $k/5$ were coincidentally correct. We used 10 test suites for each percentage level and for each faulty version.



(a) Statement Coverage Based Clustering



(b) Branch Coverage Based Clustering

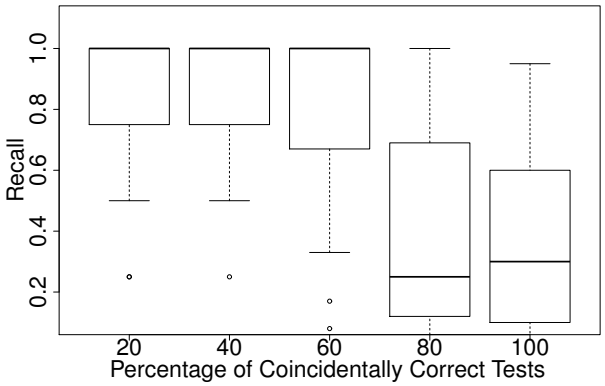


(c) Function Coverage Based Clustering

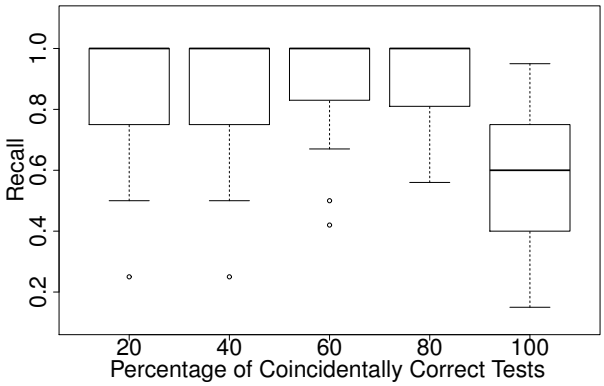
Figure 6.5: Variation of Recall for flex with Percentage of Coincidentally Correct Test Cases

The box-plots in Figures 6.5, 6.6, and 6.7 show how the recall of each classification approach varied with the percentage of coincidentally correct test cases for each benchmark. The x-axis of each box-plot represents the percentage of passing test cases that are coincidentally correct, and the y-axis represents recall. We explain with an example what each box-plot represents. In Figure 6.5(a), the box-plot corresponding to the value 80 on the x-axis shows the distribution of the recall values obtained when coincidentally correct test cases were classified using the statement coverage based clustering approach, for the faulty versions of *flex*, using test suites for which 80% of the passing test cases were coincidentally correct.

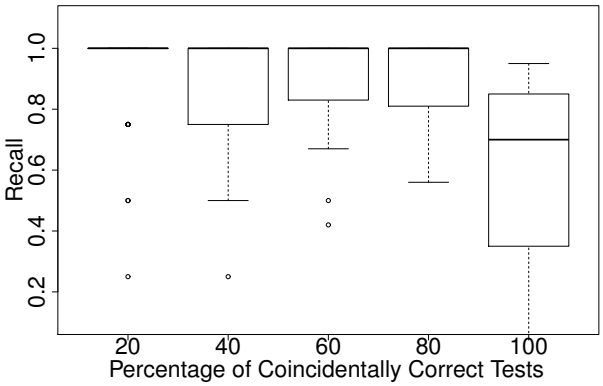
As the figures show, the recall was generally reduced when the percentage of coincidentally correct test cases increased. The test suites with higher percentage levels contained more



(a) Statement Coverage Based Clustering



(b) Branch Coverage Based Clustering



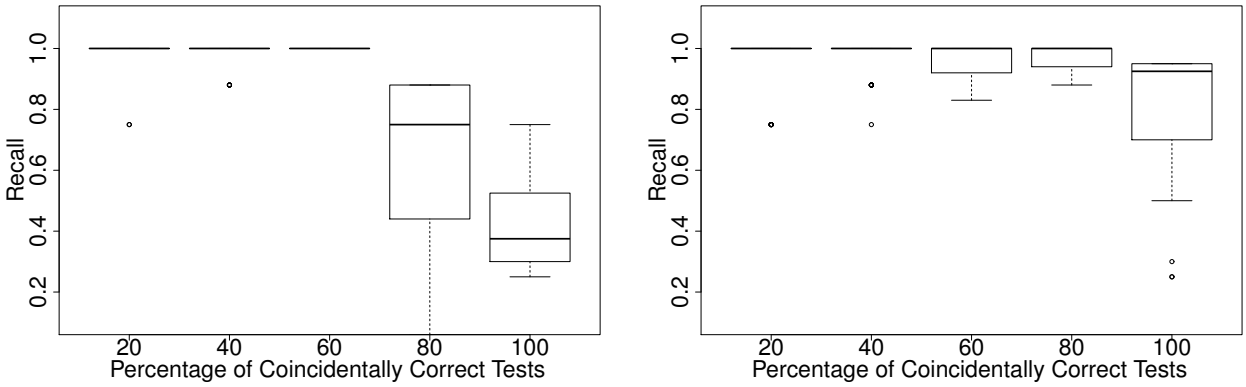
(c) Function Coverage Based Clustering

Figure 6.6: Variation of Recall for grep for Varying Percentage of Coincidentally Correct Test Cases

coincidentally correct test cases that were not similar to the failing test cases. As explained in Section 6.1, such coincidentally correct test cases were not correctly classified by the approaches, thereby reducing recall. Selecting more coincidentally correct test cases in the test suites with the higher percentage levels increased the likelihood of selecting coincidentally correct test cases that are dissimilar to the failing test cases.

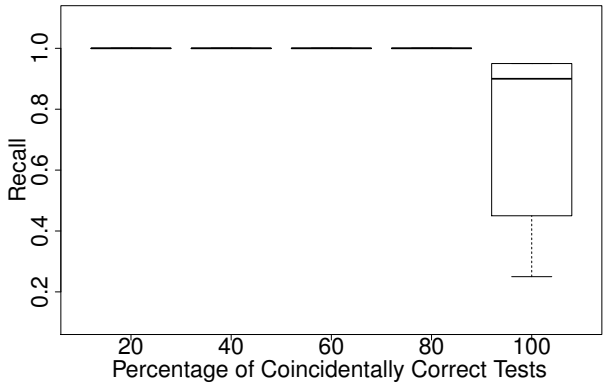
The recall of statement coverage based clustering was lower than the other two clustering approaches. This is consistent with our earlier observation presented in Section 6.1.

The box-plots in Figures 6.8, 6.9, and 6.10 show how the precision of each classification approach varied with the percentage of coincidentally correct test cases for each benchmark. These box-plots can be interpreted in a similar way as the box-plots for recall. As the



(a) Statement Coverage Based Clustering

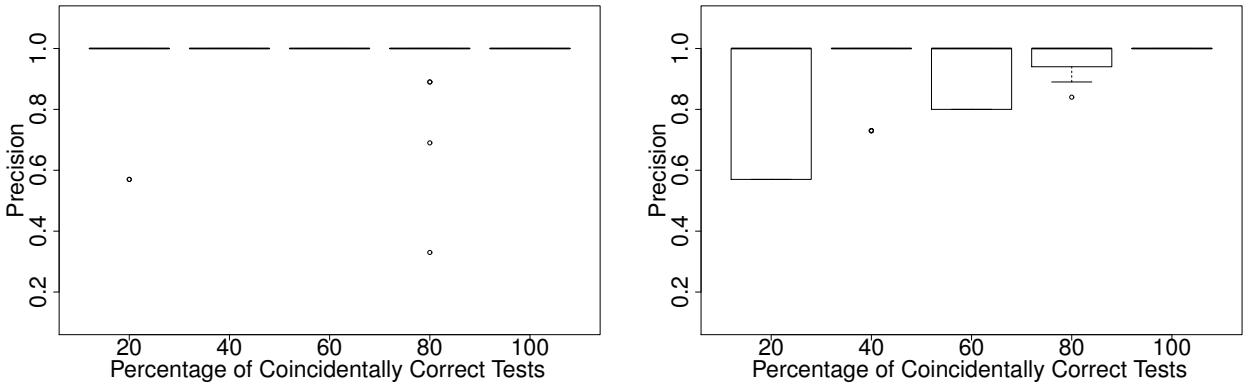
(b) Branch Coverage Based Clustering



(c) Function Coverage Based Clustering

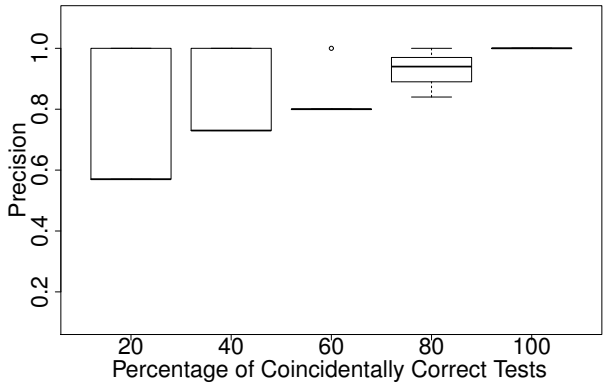
Figure 6.7: Variation of Recall for gzip for Varying Percentage of Coincidentally Correct Test Cases

figures show, the precision generally increased for all the approaches with an increase in the percentage of coincidentally correct test cases. This is expected because when $k\%$ of the passing test cases are coincidentally correct, the probability that even a randomly selected test case is coincidentally correct is $k/100$. Additionally, for lower percentage levels, the test suites contained more passing test cases that were not coincidentally correct but still similar to the failing test cases. This type of passing test cases were incorrectly classified as coincidentally correct because the classification approaches always classify passing test cases that are similar to failing test cases. Thus, this type of test cases resulted in reduced precision. Because test suites with low percentage levels selected more test cases that were



(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering



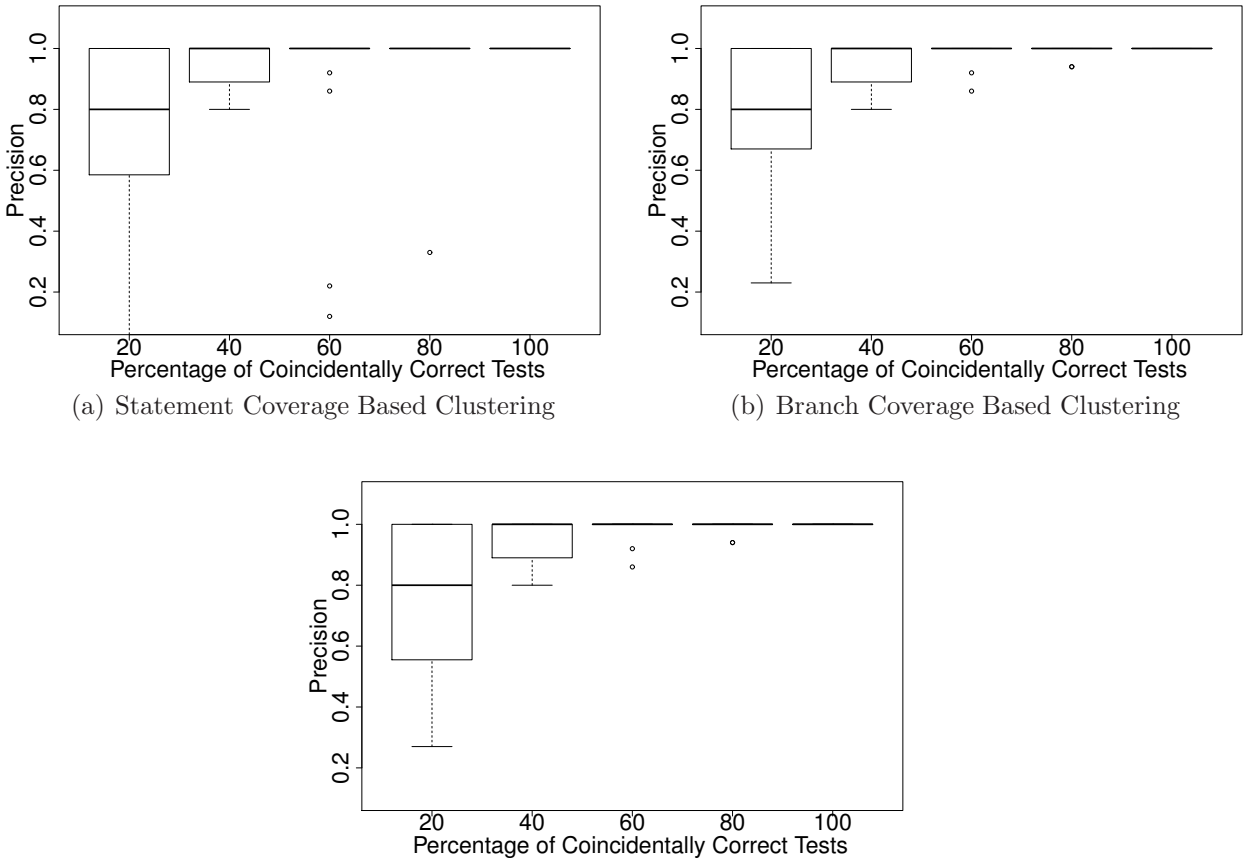
(c) Function Coverage Based Clustering

Figure 6.8: Variation of Precision for flex with Percentage of Coincidentally Correct Test Cases

passing but not coincidentally correct, these test suites were more likely to contain the type of passing test cases that caused a reduction in precision.

Figures 6.11, 6.12, and 6.13 show the effectiveness of the *Check+StJ*, *Check+BrJ*, *Check+FnJ*, and *OptJ* approaches for the benchmarks *flex*, *grep*, and *gzip*, respectively. The plots corresponding to *OptJ* are shaded grey. The x-axis of each plot shows the percentage of coincidentally correct test cases and the y-axis shows the difference in effectiveness from to the Ochiai approach.

For *flex*, the medians and quartiles of the *OptJ* box-plots for levels of the percentage of coincidentally correct test cases between 40% and 80% were nearly the same. Thus, the *OptJ* effectiveness did not increase with the increase in the percentage of coincidentally correct



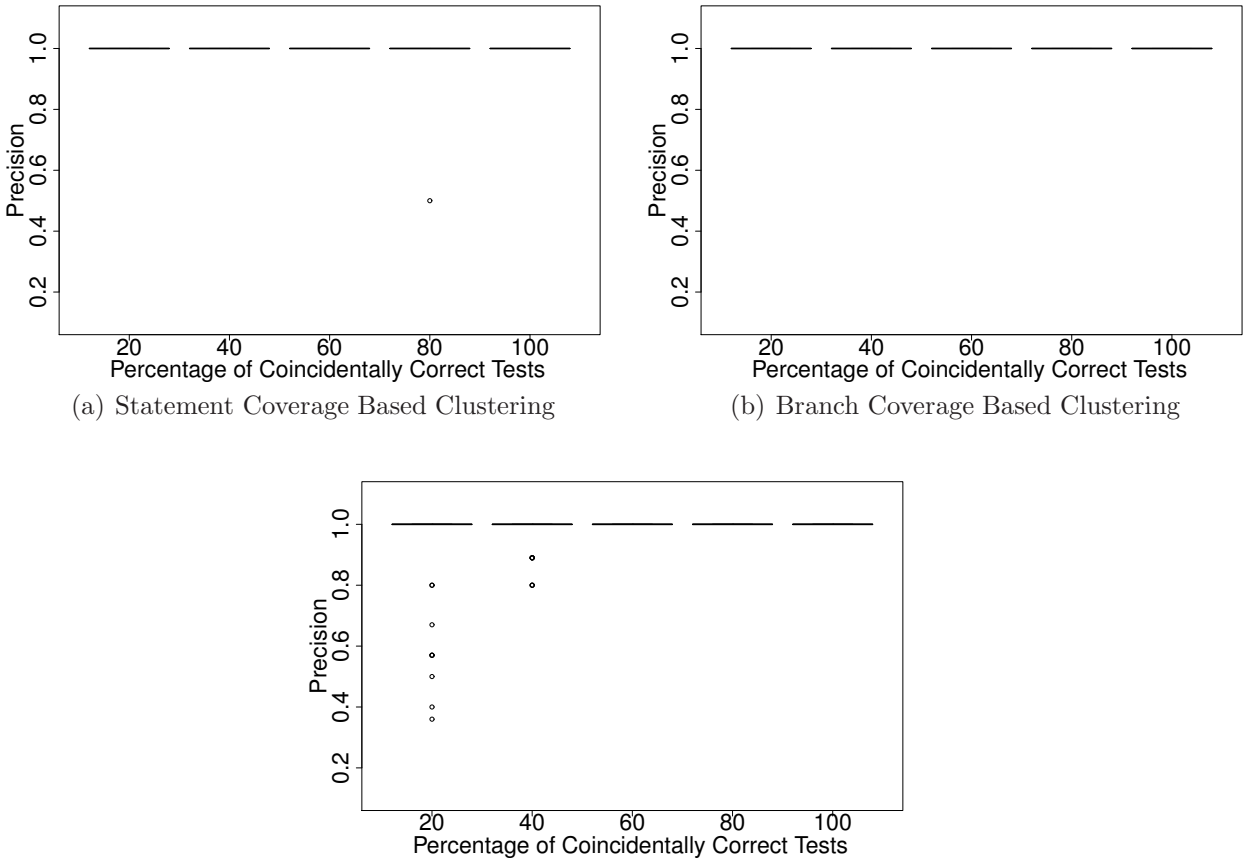
(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering

(c) Function Coverage Based Clustering

Figure 6.9: Variation of Precision for grep for Varying Percentage of Coincidentally Correct Test Cases

test cases. This is due to the nature of *flex* faults. Most faults in *flex* were in the `main` function or in functions directly called by `main`. Most passing test cases were coincidentally correct for these faults. The passing test cases that were not coincidentally correct were the ones that printed usage and covered only a few statements. Coincidentally correct test cases covered most of the program. Thus, increasing the percentage of coincidentally correct test cases decreased the Ochiai suspiciousness score of most statements and the rank of the faulty statement was not altered. Thus, the opportunity of improving the rank by addressing coincidental correctness was also unaltered. The difference in effectiveness of *OptJ* from the Ochiai approach did not increase with an increase in the percentage of coincidentally correct test cases. Because *OptJ* defines the upper bound on the effectiveness, the difference in



(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering

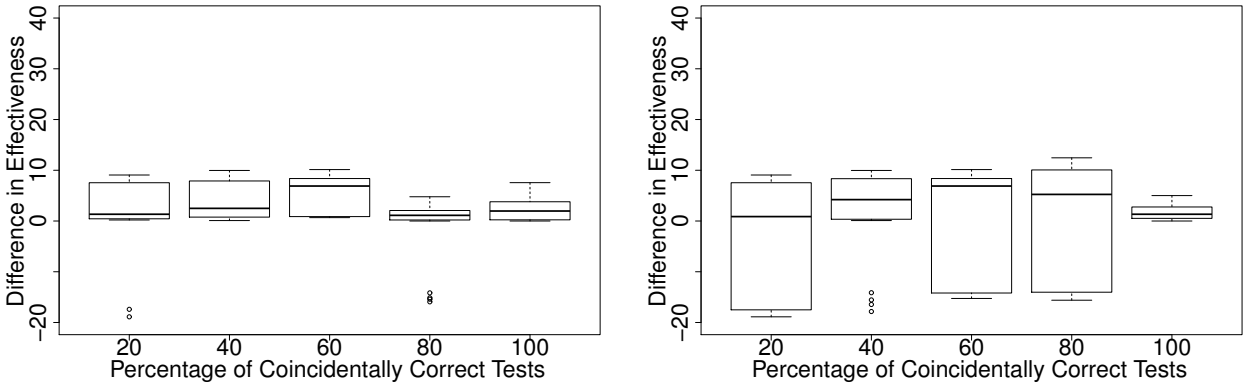
(c) Function Coverage Based Clustering

Figure 6.10: Variation of Precision for gzip for Varying Percentage of Coincidentally Correct Test Cases

effectiveness from the Ochiai approach did not increase for the *Check+StJ*, *Check+BrJ*, and *Check+FnJ* approaches as well.

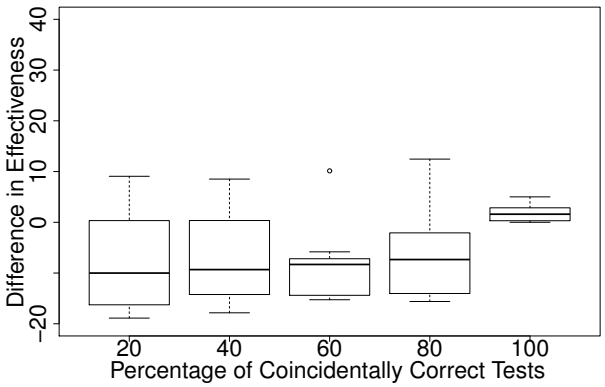
For both *grep* and *gzip*, the difference in effectiveness of *OptJ* from the Ochiai approach increased with the percentage of coincidentally correct test cases. Unlike *flex*, most *gzip* and *grep* faults were not located close to the program entry, such as in the `main` function. Thus, coincidentally correct test cases did not cover most of the program and the opportunity for improving the rank of the faulty statement increased with the increase in the percentage of coincidentally correct test cases.

The difference in effectiveness from the Ochiai approach of the *Check+StJ*, *Check+BrJ*, and *Check+FnJ* approaches initially increased with an increase in the percentage of coinci-

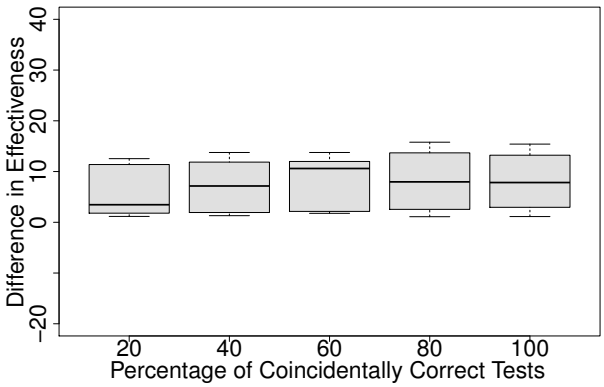


(a) Check+StJ

(b) Check+BrJ



(c) Check+FnJ

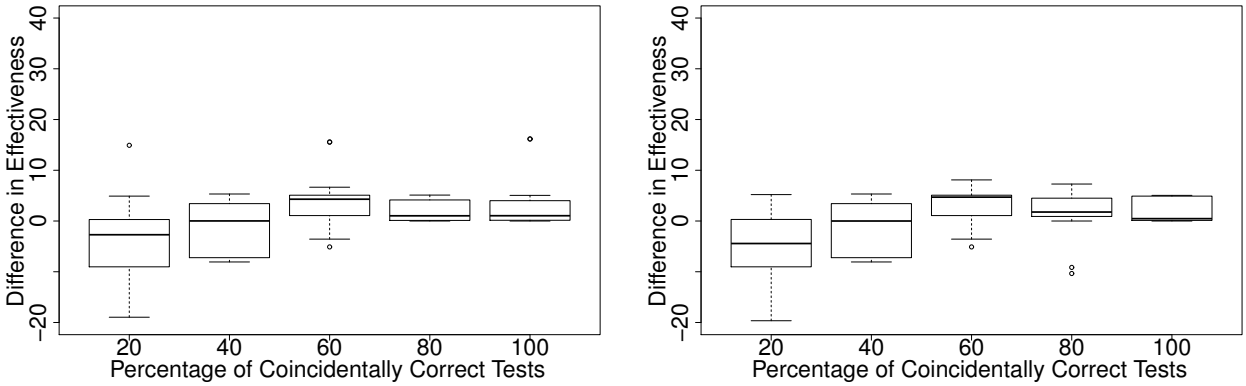


(d) OptJ

Figure 6.11: Variation of Effectiveness for flex for Varying Percentage of Coincidentally Correct Test Cases

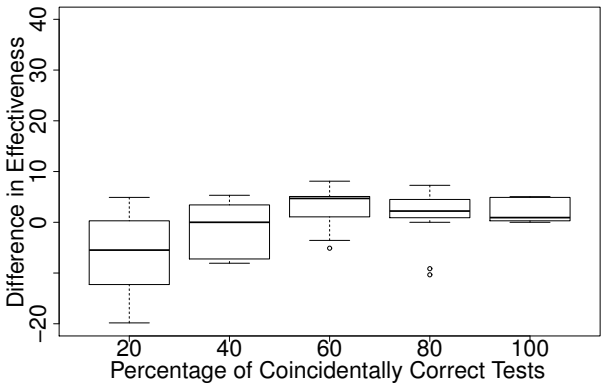
dentally correct test cases. However, when the percentage was above 60 or 80, the difference in effectiveness of the approaches was reduced. The difference in effectiveness initially improved because with more coincidentally correct test cases there was more opportunity for improvement. At high percentages our approaches could not capitalize on the opportunity because when most test cases were coincidentally correct, most non-faulty statements were also executed by coincidentally correct test cases. Thus, addressing coincidental correctness increased the suspiciousness score of both the faulty and non-faulty statements. Therefore, the rank of the faulty statement did not improve relative to the non-faulty statements.

We did not perform pairwise Student's t-test to investigate whether or not the observations above were statistically significant. Student's t-test is not applicable because the test

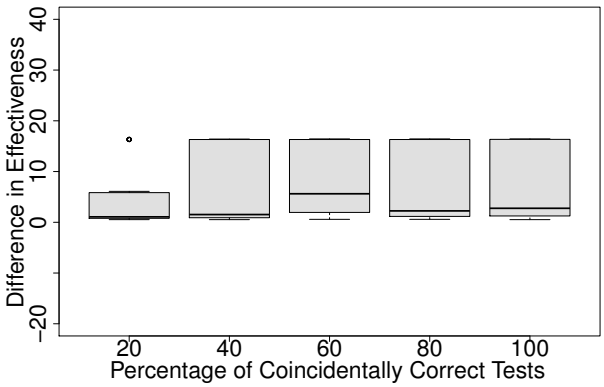


(a) Check+StJ

(b) Check+BrJ



(c) Check+FnJ



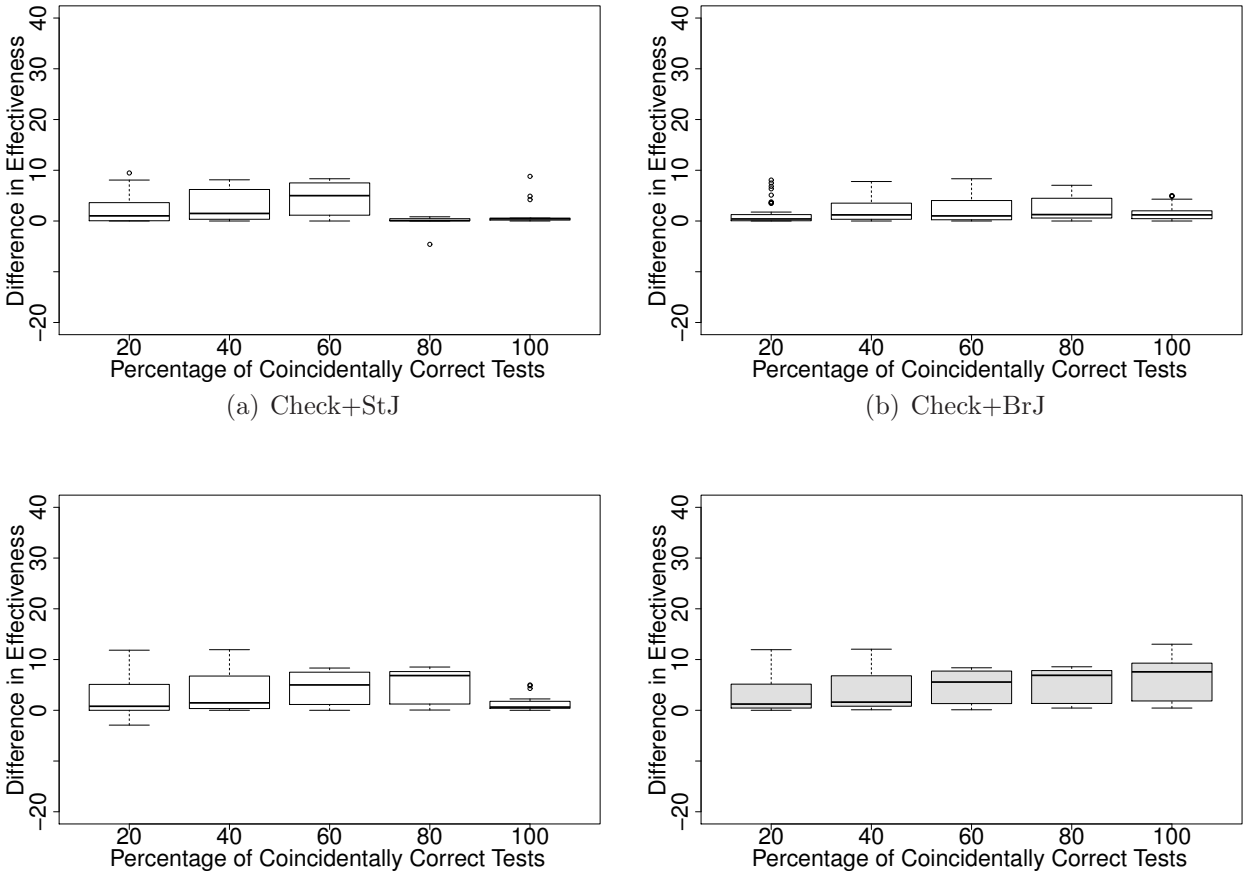
(d) OptJ

Figure 6.12: Variation of Effectiveness for grep for Varying Percentage of Coincidentally Correct Test Cases

suites used for the different levels of the percentage values were different. Therefore, two effectiveness values obtained using test suites containing two different levels of the percentage of coincidentally correct test cases cannot be compared in pairs.

6.2.2 RQ6: Effect of Test Suite Type

We performed a study with random, statement adequate, and branch adequate test suites. We created coverage adequate test suites of size similar to the random test suites. Each coverage adequate test suite contained 1–5 failing test cases and 20 passing test cases except when satisfying the coverage criteria required more test cases. We did not control the percentage of coincidentally correct test cases and the ratio of the number of failing test



(a) Check+StJ

(b) Check+BrJ

(c) Check+FnJ

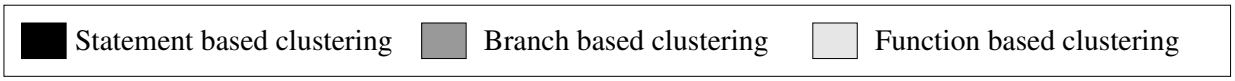
(d) OptJ

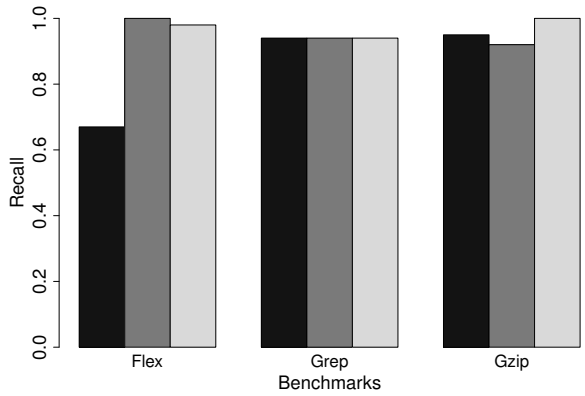
Figure 6.13: Variation of Effectiveness for gzip for Varying Percentage of Coincidentally Correct Test Cases

cases to the number of passing test cases. It was not always possible to control the above variables while satisfying the coverage criteria.

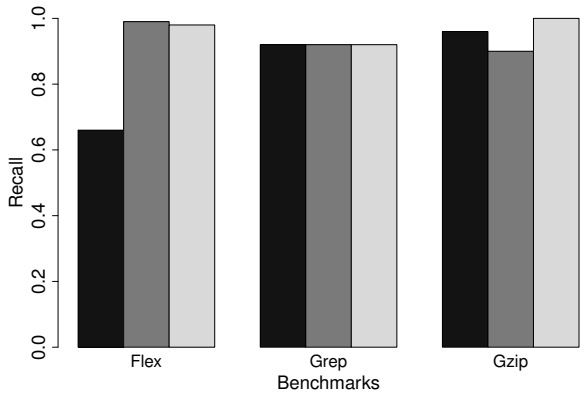
In Section 6.1, we described the method used to select random test suites. Below, we describe the method used to select the coverage adequate test suites. We selected each statement (or branch) coverage adequate test suite such that it achieved the same statement (or branch) coverage as the entire pool of test cases. The pools of test cases contained between 75% to 85% statement and branch coverage. To generate each test suite, we first selected a random failing test case. We continued selecting one additional failing test case at each step until the selected failing test cases achieved the same statement (or branch) coverage as all the failing test cases in the pool. At each step the failing test case was selected using a

greedy strategy. We selected the test case that covered the maximum number of statements (or branches) that were not covered by the already selected test cases. We applied the same strategy to select a set of passing test cases that covered the statements (or branches) executed by the passing test cases in the pool. Sometimes test adequacy was achieved with less than 20 passing test cases. We randomly selected additional passing test cases until the test suite contained 20 passing test cases.

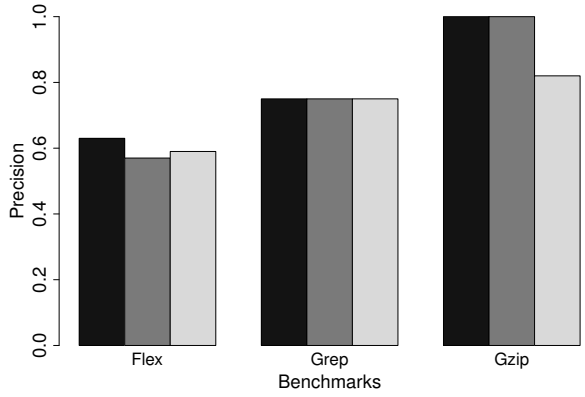




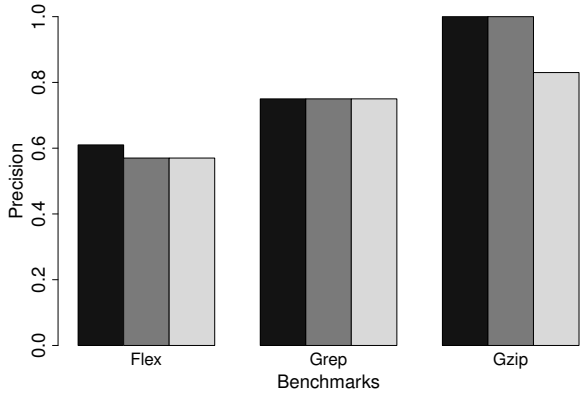
(b) Recall Statement Adequate



(c) Recall Branch Adequate



(d) Precision Statement Adequate



(e) Precision Branch Adequate

Figure 6.14: Recall and Precision of the Classification Techniques for Coverage-Adequate Test Suites

In Figure 6.1, we presented the recall and precision of the classification approaches using random test suites. Figure 6.14 shows the recall and precision of the classification approaches

using statement coverage adequate and branch coverage adequate test suites. Comparing with random test suites, we observed that when coverage adequate test suites were used, the recall remained high but the precision was reduced. The reduction in precision can be attributed to one or more of the following reasons:

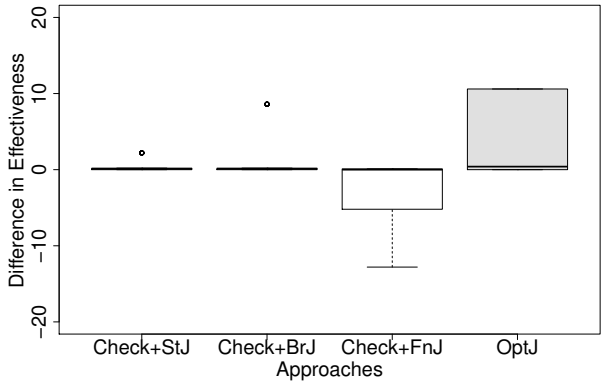
- In many statement coverage and branch coverage adequate test suites, only less than 10% of the passing test cases were coincidentally correct. When the number of coincidentally correct test cases is low, a classification approach is less likely to classify only the actual coincidentally correct test cases. This results in low precision. The low number of coincidentally correct test cases resulted from the greedy method of selecting test cases. If two test cases had similar coverage and one was selected, our method did not select the other. For many faults, the coincidentally correct test cases had similar coverage. Thus, selecting a few of them precluded the selection of the rest.

- During our test selection process, at each step, we selected a test case that had the maximum possible distance from the already selected test cases. This prevented the formation of natural clusters in the test vectors and resulted in the test vectors to be spread out in the space of vectors as much as possible. In the absence of natural clusters, k-means clustering was less accurate.

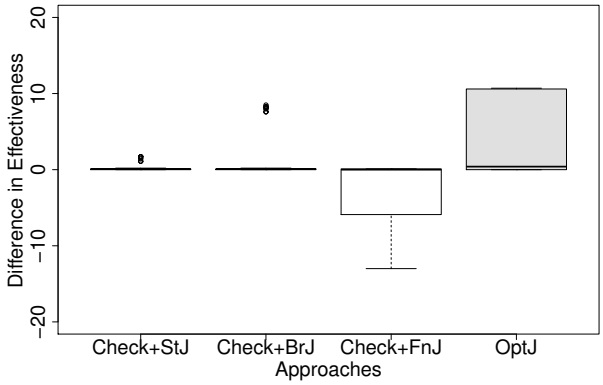
- The coverage adequate test suites often contained passing test cases that were not coincidentally correct but were similar to the failing test cases. Satisfying the coverage criteria often required the inclusion of these test cases while creating the test suites as these test cases often covered statements/branches that other test cases did not.

The precision of statement and branch coverage based clustering approaches were higher than that of function coverage based clustering approaches. This happened due to the presence of the passing test cases that were not coincidentally correct but were similar to the failing test cases. These test cases often executed the function containing the faulty statement but did not execute the faulty statement. Function coverage based clustering abstracted away the differences within individual functions and failed to distinguish these

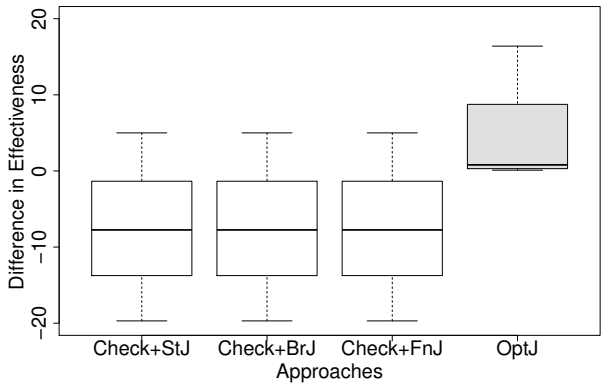
test cases from the actual coincidentally correct test cases. Statement and branch coverage based clustering approaches magnified the differences within each function and often were able to distinguish these passing tests from the actual coincidentally correct test cases.



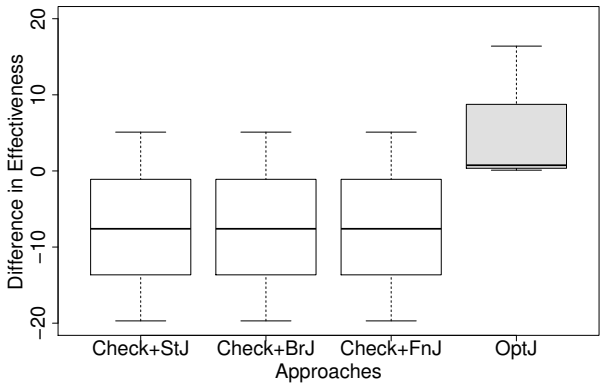
(a) flex Statement Adequate



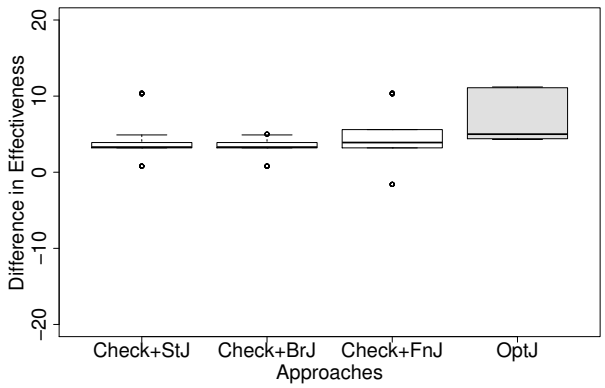
(b) flex Branch Adequate



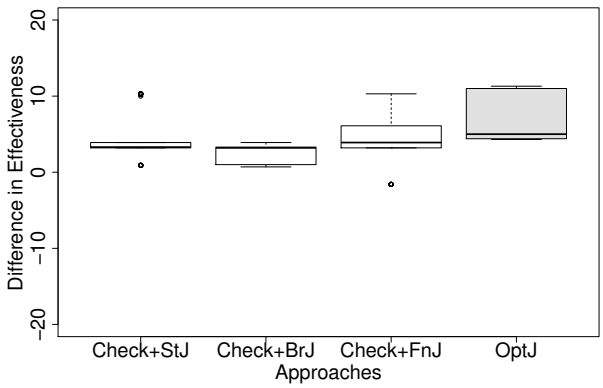
(c) grep Statement Adequate



(d) grep Branch Adequate



(e) gzip Statement Adequate



(f) gzip Branch Adequate

Figure 6.15: Difference in Effectiveness from the Ochiai Approach for Test Suite Types

In Figure 6.4, we presented the box-plots for the difference in effectiveness of the approaches from the Ochiai approach obtained by using random test suites. Figure 6.15 shows the difference in effectiveness when statement and branch adequate test suites were used. The effectiveness for *flex* did not change. This is because in most test suites the number of coincidentally correct test cases was low. The faulty statement already had the highest suspiciousness score and was identified during the *Checking* phase of our approach. For *grep*, the effectiveness was reduced for all the approaches due to the reduction in the precision of the classification. The incorrectly classified passing test cases had a high coverage and using them to calculate suspiciousness scores caused many non-faulty statements to be ranked higher than the faulty statement.

For *gzip*, the effectiveness of the approaches improved despite the reduction in precision because the incorrectly classified passing test cases had a low coverage and affected the suspiciousness scores of only a few statements. For example, each coverage adequate test suite in *gzip* contained three types of passing test cases: (1) test cases that caused the program to print usage information, (2) test cases that performed compression, and (3) test cases that performed decompression. Inclusion of all these three types of test cases was required to achieve the target coverage. When the faulty statement was in the code for compression, the coincidentally correct test cases were also the ones that performed compression. When clustered, test cases of type (2) clustered with test cases of type (1), while the test cases of type (3) formed a different cluster. The reason is that while the test cases of type (1) were dissimilar to both the test cases of types (2) and (3), the dissimilarity with test cases of type (3) was larger. The test cases of type (1) neither covered the program elements in the compression code, nor the ones in the decompression code, but the compression code comprised fewer program elements than the decompression code. Because the compression test cases were classified as coincidentally correct, test cases that caused printing usage information were also classified as coincidentally correct. However, this incorrect classification did not affect effectiveness because these incorrectly classified test cases covered only a few statements.

We did not perform pairwise Student's t-test to investigate whether or not the observations above were statistically significant. Student's t-test is not applicable because the test suites of different types contained different test cases. Therefore, two effectiveness values obtained using test suites of two different types cannot be compared in pairs.

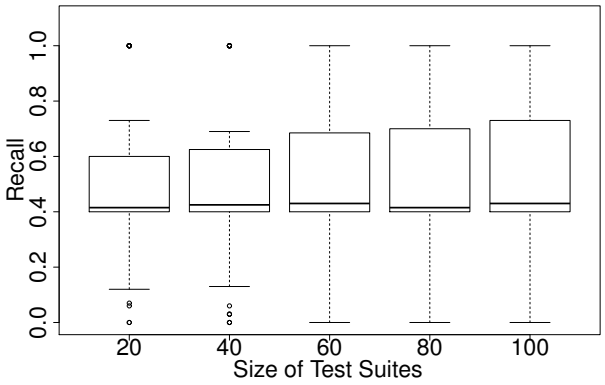
6.2.3 RQ7: Effect of the Size of Test Suites

We evaluated our approaches with test suites of 5 different sizes: 20, 40, 60, 80, and 100. Test suites of each type were constructed by randomly selecting passing and failing test cases from the test pools. We maintained the ratio of failing and passing test cases at 1/5 in all the suites. We did not control for the percentage of coincidentally correct test cases because for some faults there were not enough coincidentally correct test cases to maintain a fixed percentage for large test suites.

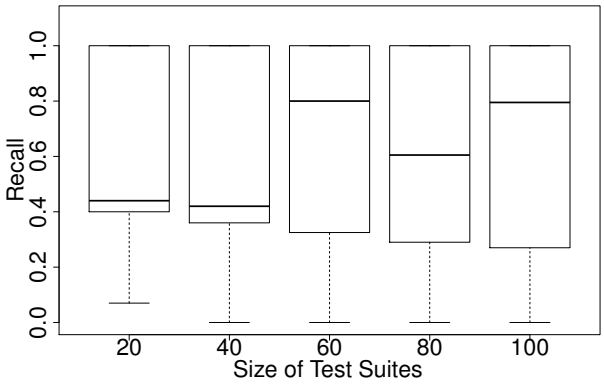
The box-plots in Figures 6.16, 6.17, and 6.18 show how the recall of each classification approach varied with the size of test suites for each benchmark. The box-plots in Figures 6.19, 6.20, and 6.21 show similar plots for precision. There was no strong evidence suggesting that test suite size impacts recall and precision.

Figures 6.22, 6.23, and 6.24 respectively show the effectiveness of the approaches for variation in the size of test suites for the benchmarks *flex*, *grep*, and *gzip*. There was no strong evidence suggesting that test suite size affects effectiveness. The *OptJ* effectiveness did not vary with test suite size, indicating that the opportunity for improving effectiveness by addressing coincidentally correct test cases also did not change with size of test suites.

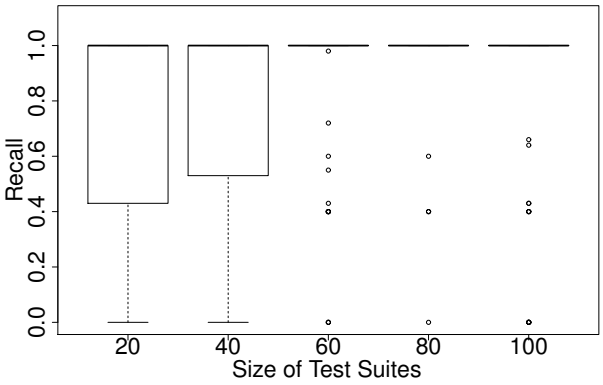
We did not perform pairwise Student's t-test to investigate whether or not the observations above were statistically significant. Student's t-test is not applicable because the test suites of different sizes contained different test cases. Therefore, two effectiveness values obtained using test suites of two different sizes cannot be compared in pairs.



(a) Statement Coverage Based Clustering



(b) Branch Coverage Based Clustering



(c) Branch Coverage Based Clustering

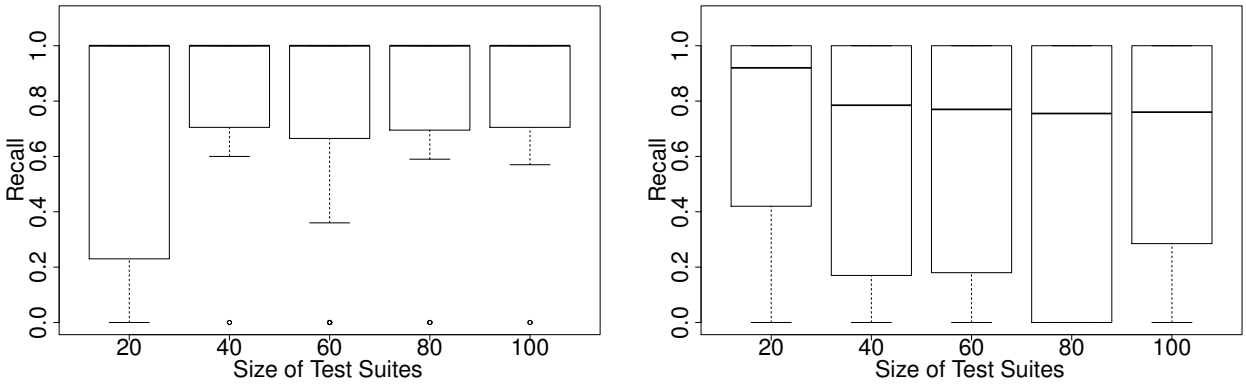
Figure 6.16: Variation of Recall for flex with Size of Test Suites

6.3 Evaluation in the Presence of Two Faults

We evaluated the fault localization effectiveness of *Check+StJ*, *Check+BrJ*, and *Check+FnJ* for programs containing two faults, based on the following research questions:

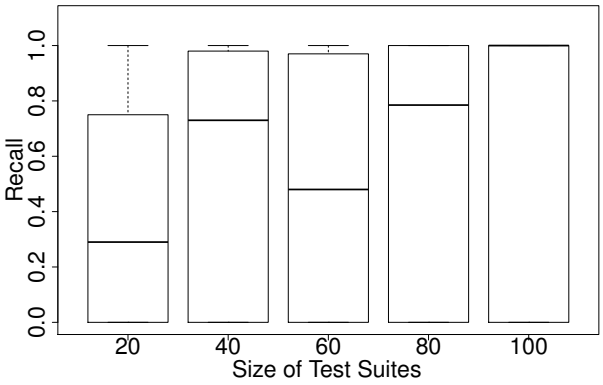
RQ8: What is the effectiveness of the approaches for each of the faults?

Our goal is to investigate how the effectiveness of localizing each fault improves by addressing coincidentally correct test cases. We assume that a tester will localize each fault separately based on the fault localization results.



(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering



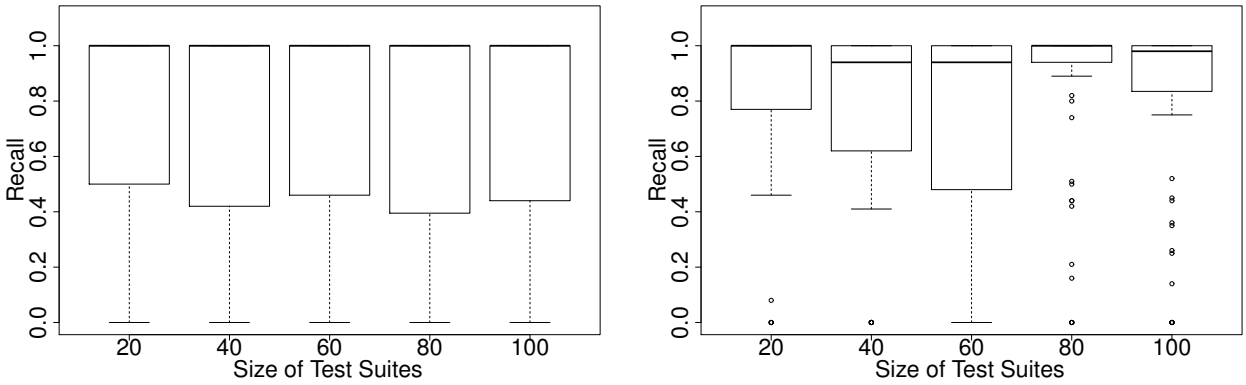
(c) Function Coverage Based Clustering

Figure 6.17: Variation of Recall for grep with Size of Test Suites

RQ9: How is the effectiveness for one fault associated with the effectiveness for the other fault?

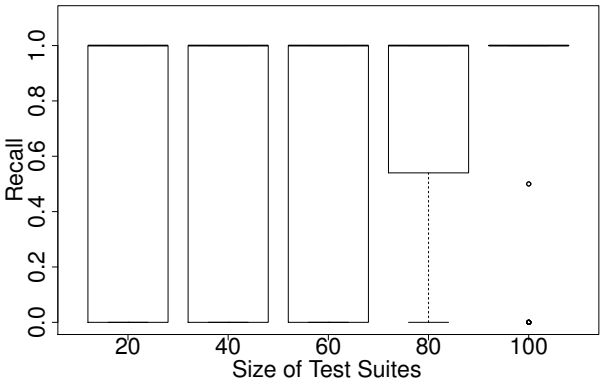
We want to answer this research question because we speculate that classifying test cases as either coincidentally correct or not coincidentally correct may favor one fault more than the other. Thus, upon addressing coincidentally correct test cases, the effectiveness for one fault may improve, while that for the other fault may be reduced.

We first created faulty versions each containing two faults, by activating all possible pairs of single faults in version-1 of *flex*, version-3 of *grep* and version-1 of *gzip*. We could not use version-1 and version-2 of *grep* because every test case failed for every multiple fault program created for those versions. We executed each 2-fault version with the large pool of test cases available for the corresponding benchmark. Finally, for each version, we created



(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering

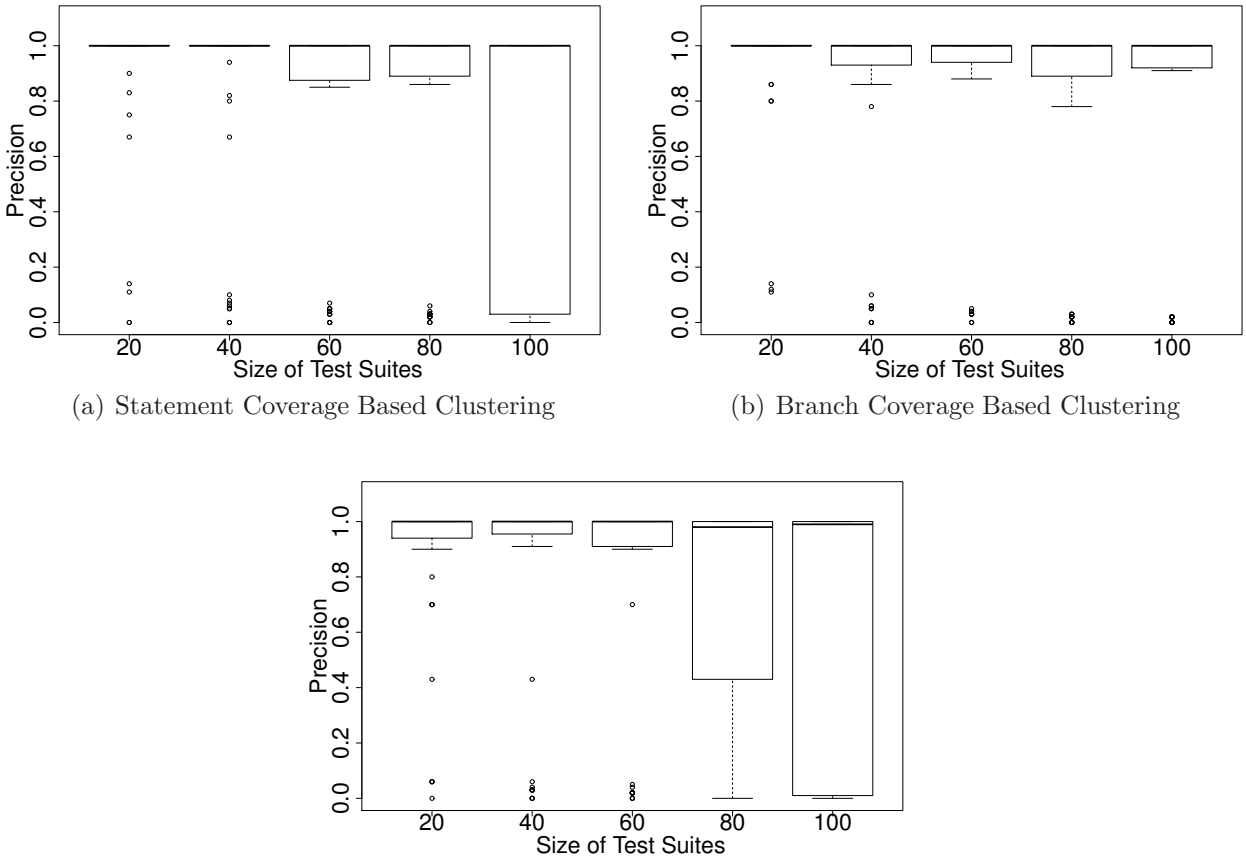


(c) Function Coverage Based Clustering

Figure 6.18: Variation of Recall for gzip for with Size of Test Suites

test suites containing 1–5 failing test cases and 20 passing test cases by randomly grouping failing and passing test cases from the large pool. For each *(faulty version, test suite)* pair, we applied the approaches *Check+StJ*, *Check+BrJ*, and *Check+FnJ*. We only considered the pairs for which each faulty statement was covered by at least one failing test case. This is because a fault cannot be localized if no failing test case that executes the faulty statement is available. In each *(faulty version, test suite)* pair, among the two faults, we denoted the faulty statement ranked higher with respect to the Ochiai suspiciousness scores by $fault_1$, and the fault ranked lower by $fault_2$.

Figure 6.25 shows the distribution of the difference in effectiveness of our approaches from the Ochiai approach for each of $fault_1$ and $fault_2$ in each benchmark. The figure shows that the approaches improved the effectiveness for both $fault_1$ and $fault_2$ in all the cases with



(a) Statement Coverage Based Clustering

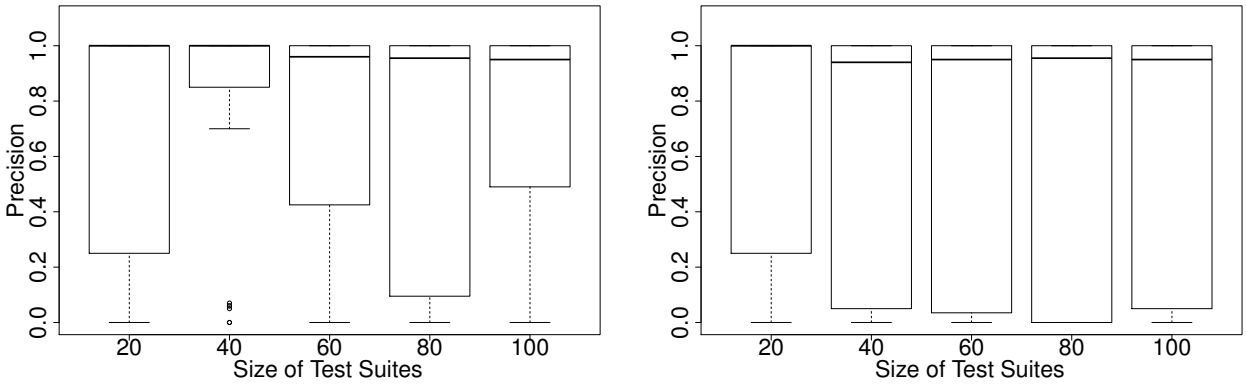
(b) Branch Coverage Based Clustering

(c) Function Coverage Based Clustering

Figure 6.19: Variation of Precision for flex with Size of Test Suites

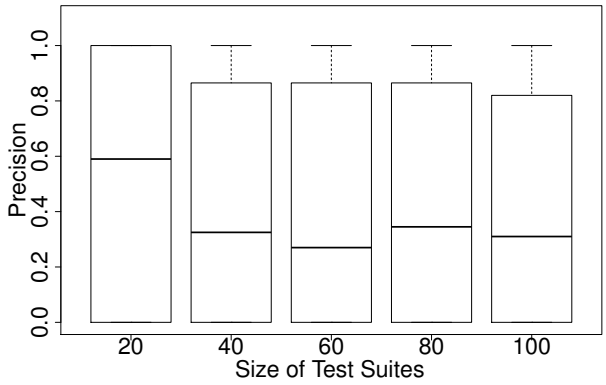
the exception that the effectiveness of *Check+FnJ* for *flex* was reduced for $fault_2$ compared to the Ochiai approach. For flex, 50% of the faults were located in `make_tables()` function. These faults are hard to reach because they are guarded by many conditions. Less than 1% of the passing test cases for these faults were coincidentally correct. Because the execution of `make_tables()` often did not result in the execution of the faults, function coverage based clustering resulted in incorrect classification of coincidentally correct test cases for these faults.

Given the above argument, function coverage based clustering should have performed poorly when these faults were present individually. However, in single fault cases, the coincidentally correct test cases often did not get selected in the test suites because there were many passing test cases. Among the test cases that passed in the presence of a single fault,



(a) Statement Coverage Based Clustering

(b) Branch Coverage Based Clustering



(c) Function Coverage Based Clustering

Figure 6.20: Variation of Precision for grep with Size of Test Suites

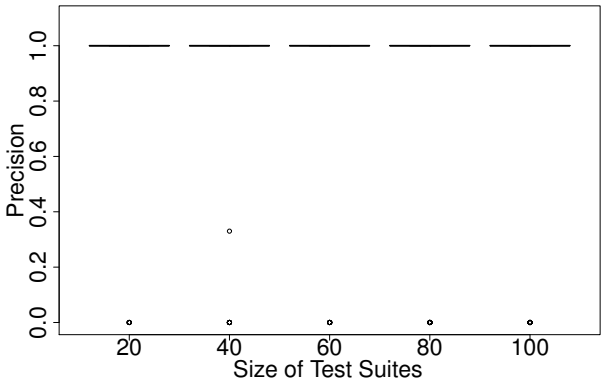
many failed when multiple faults were present. Thus, in the presence of multiple faults, because there were fewer passing test cases, the likelihood of selecting the coincidentally test cases increased while creating the test suites. The test suites contained the coincidentally correct test cases more often.

Figure 6.26 addresses the second research question. It shows the percentage of the following three types of cases for each benchmark:

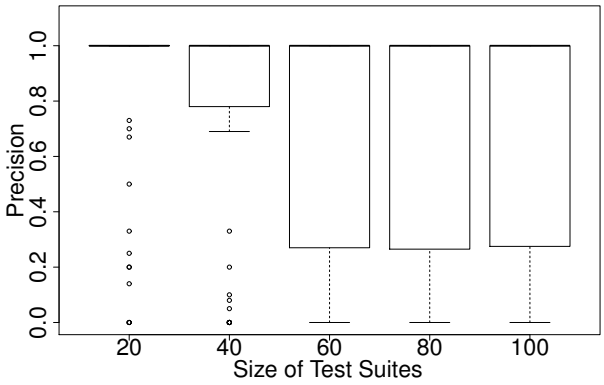
1. *Both improved*: Cases where the effectiveness for both faults improved at the same time.

2. *Exactly one improved*: Cases where the effectiveness improved for exactly one fault.

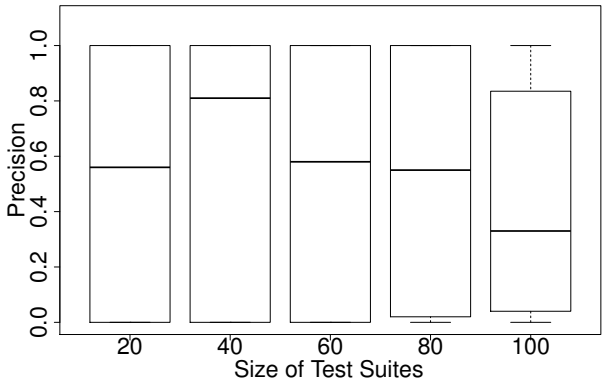
3. *None improved*: Cases where the effectiveness did not improve for any fault.



(a) Statement Coverage Based Clustering



(b) Branch Coverage Based Clustering

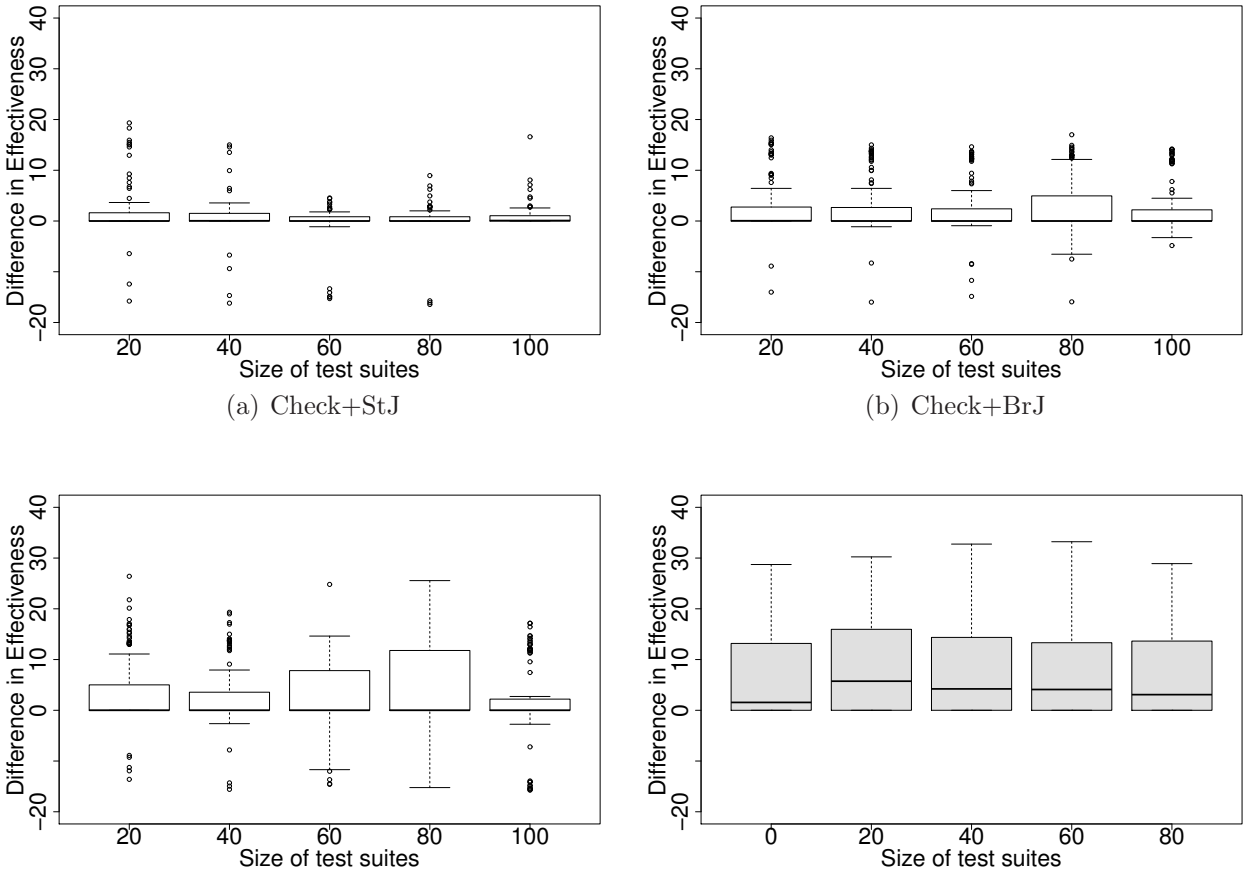


(c) Function Coverage Based Clustering

Figure 6.21: Variation of Precision for gzip with Size of Test Suites

The effectiveness for both faults improved simultaneously in the majority of the cases. For such cases, the passing test cases that are coincidentally correct for one fault were also coincidentally correct for the other. Thus, when the coincidentally correct test cases were classified with high recall and precision for one fault, the coincidentally correct test cases for the other fault were also classified with high recall and precision.

Deeper analysis of the above three cases revealed that cases (1) and (3) happened when the functions containing the two faults had a caller-callee relationship directly or through other intermediate calls. Under such situations, the coincidentally correct test cases that executed one faulty statement also executed the other. Therefore, test cases that were coincidentally correct with respect to one fault covered a similar set of statements compared to those that are coincidentally correct with respect to the other fault. Thus, the classification



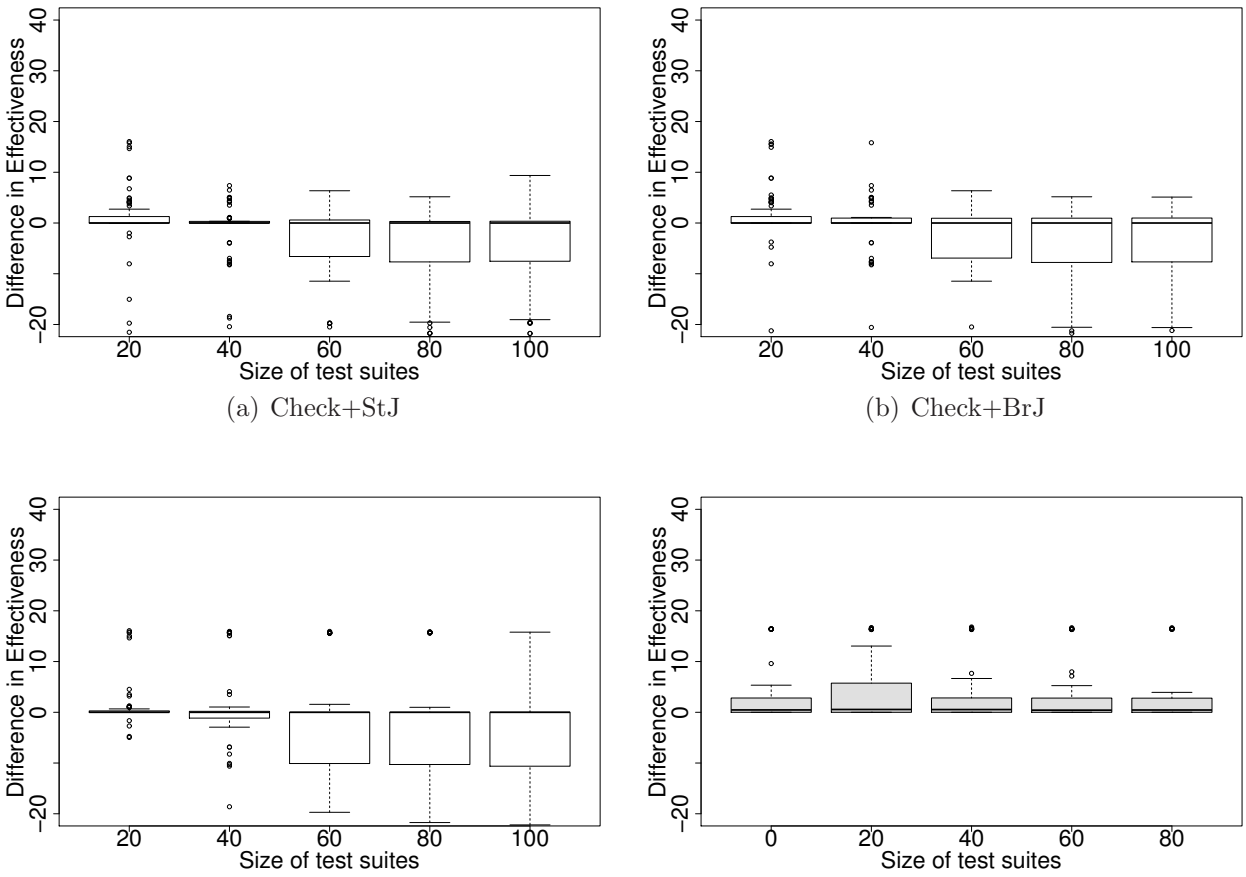
(a) Check+StJ (b) Check+BrJ

(c) Check+FnJ (d) OptJ

Figure 6.22: Variation of Effectiveness for with Size of Test Suites

approaches either correctly classified the coincidentally correct test cases for both faults resulting in case (1), or for none, resulting in case (3). Due to a similar reason, when the functions containing the faults did not have a direct or indirect caller-callee relationship, the coincidentally correct test cases for the two faults covered very different sets of statements. Thus, clustering was able to classify the coincidentally correct test cases for one fault but not for the other. This resulted in case (2).

Based on this observation, we recommend that a tester follow the strategy described below while localizing multiple faults. If a faulty statement is identified in the ranked list of statements, a tester has two choices: (1) continue inspecting the ranked list to find more faulty statements, or (2) fix the already found fault and perform fault localization again. We



(a) Check+StJ (b) Check+BrJ

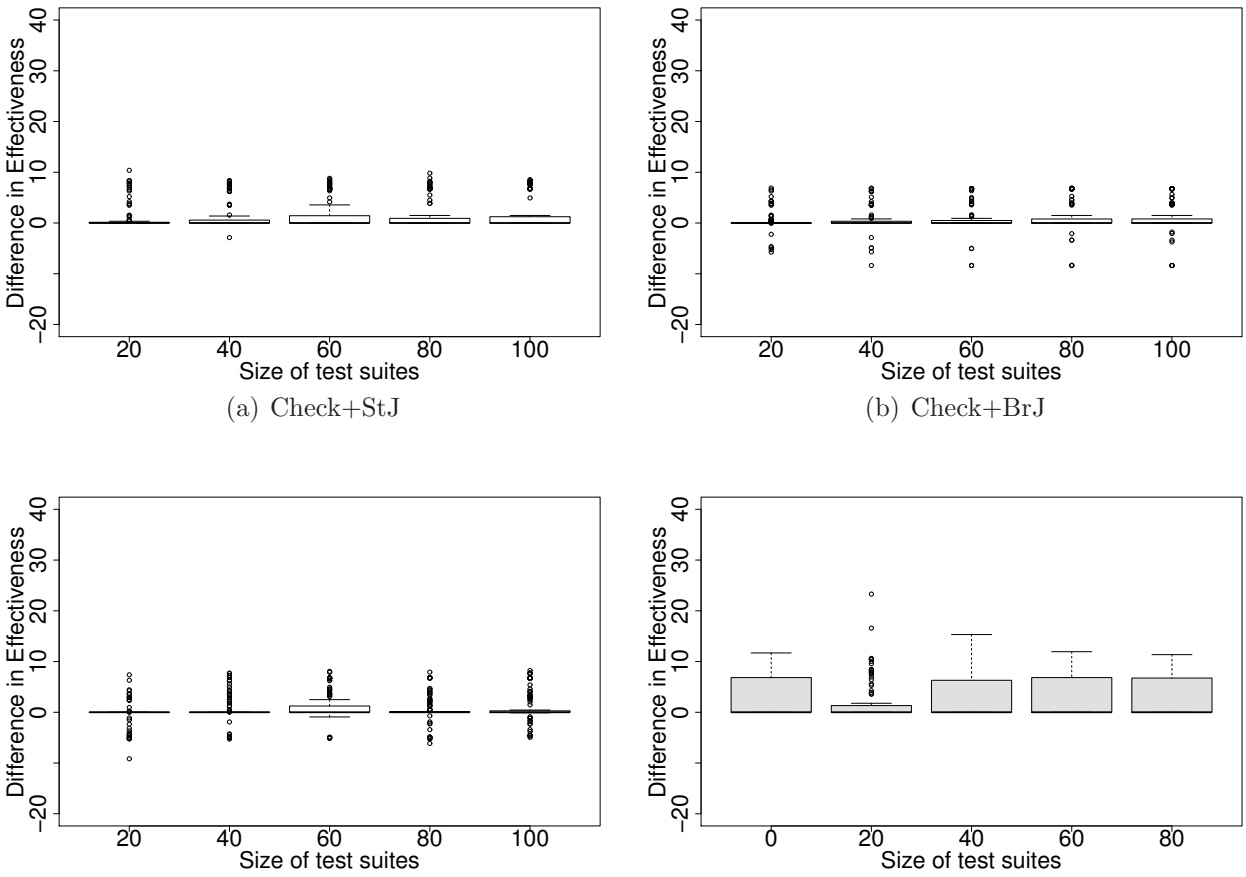
(c) Check+FnJ (d) OptJ

Figure 6.23: Variation of Effectiveness for grep with Size of Test Suites

recommend that the tester should inspect a few more statements to exploit the possibility of finding faults that are located in functions having direct or indirect caller-callee relationships. However, if none of the closely ranked statements are faulty, addressing coincidentally correct test cases for one faulty statement must have caused a reduction in the ranks of the other faulty statements. Therefore, the tester should fix the already identified fault and perform fault localization again.

6.4 Threats to Validity

External: Threats to external validity stem from the nature of benchmarks used. The programs are relatively small and may not represent all real world programs. The runtime



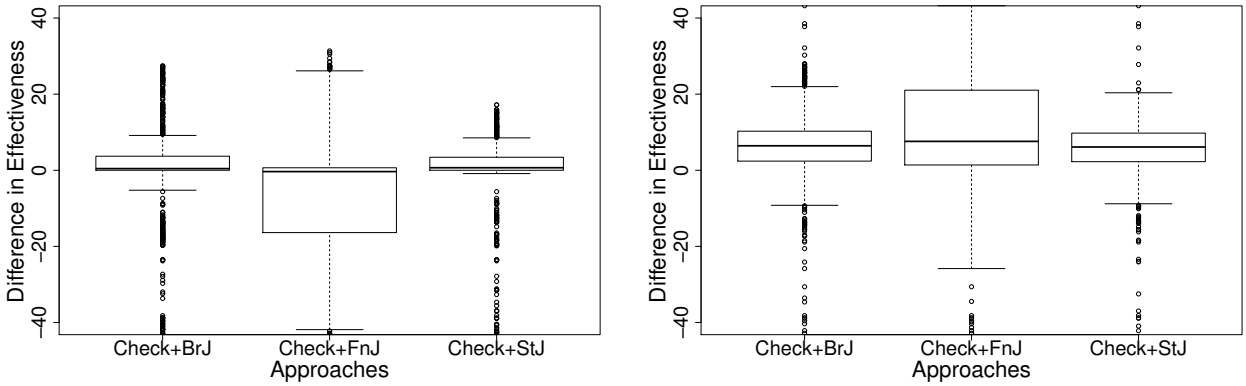
(a) Check+StJ (b) Check+BrJ

(c) Check+FnJ (d) OptJ

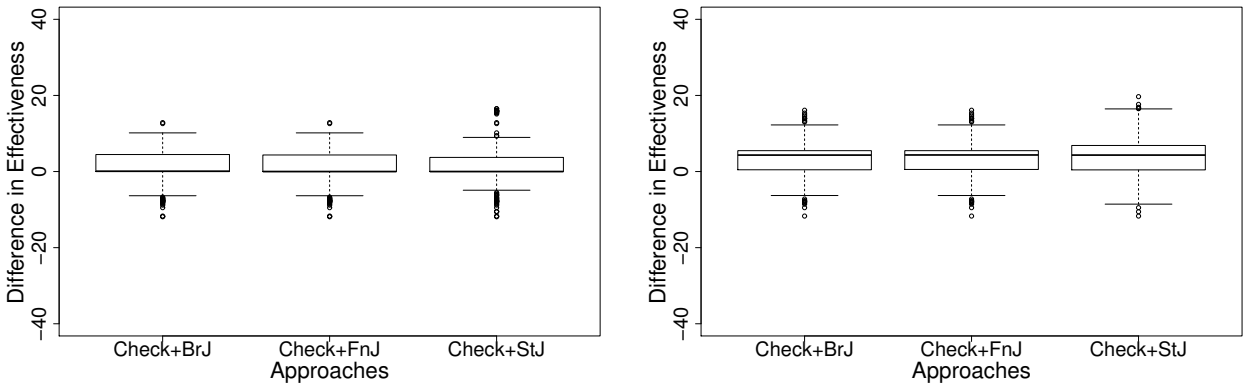
Figure 6.24: Variation of Effectiveness for gzip with Size of Test Suites

of our approaches depends on the number of program elements, which is large for large programs. Thus, the approaches, particularly the ones using fine-grained coverage spectra may not scale for large programs. However, this is true for any fault localization approach that uses program spectra.

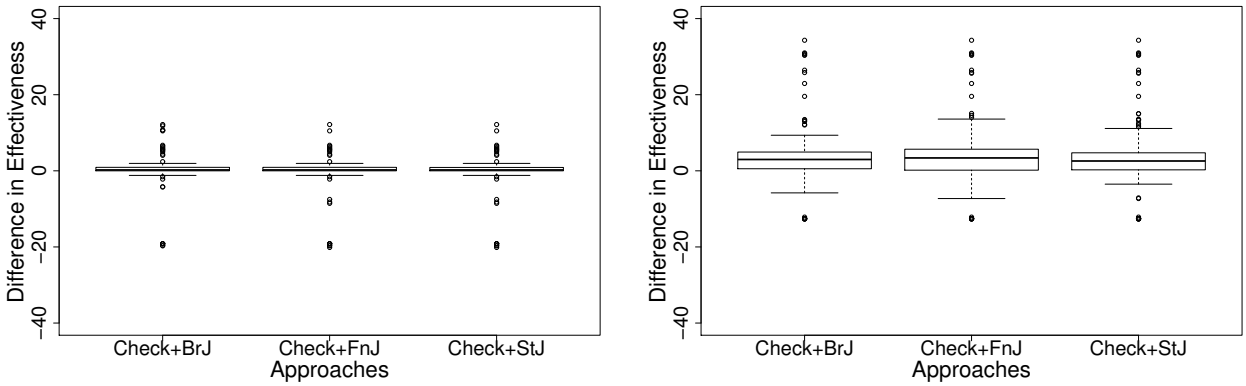
Internal: Distribution of the types of faults presents a threat to internal validity. Depending on the location and nature of faults, coincidentally correct test cases for some faults may be classified more accurately compared to other faults. We plan to study the effectiveness of our approaches for faults of different types at diverse program locations. A mutation tool such as MuJava [36] can be used for this purpose.



(a) flex, $fault_1$ (b) flex, $fault_2$



(c) grep, $fault_1$ (d) grep, $fault_2$



(e) gzip, $fault_1$ (f) gzip, $fault_2$

Figure 6.25: Difference in Effectiveness of the Approaches from the Ochiai approach for Each Fault in Programs Containing Two Faults

Another threat to internal validity arises from the nature of the test cases. All test cases that we used are system test cases. Clustering test cases based on a coarse-grained

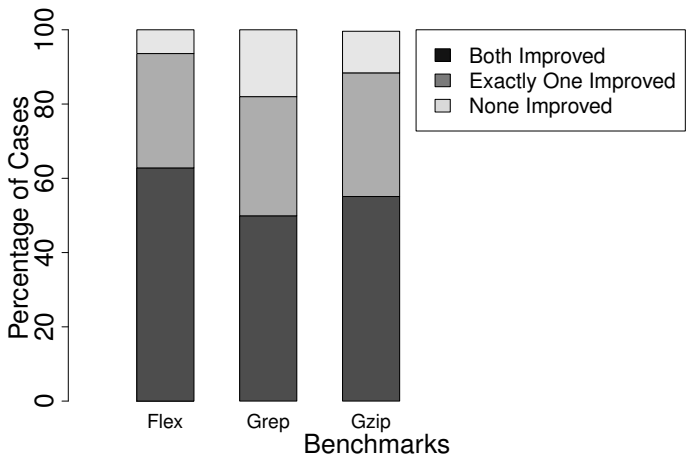


Figure 6.26: Distribution of the Cases When the Effectiveness Improved for Both Faults, Exactly One Fault, and for No Fault

coverage metric, such as function coverage may have been more accurate because of the use of system test cases. In the presence of unit test cases, a coarse-grained coverage may not be as accurate for the classification of coincidentally correct test cases. A unit test typically tests a single function, and will only cover the function under test and the functions called by the function under test. Thus, function coverage will not provide adequate information for distinguishing the coincidentally correct test cases.

Other threats to internal validity arise due to the characteristics of test suites that we did not control for. In our studies to evaluate the effect of test suite size and test objective on the fault localization effectiveness, we did not control the percentage of coincidentally correct tests. Maintaining a certain percentage was not always feasible because a sufficient number of coincidentally correct test cases was not always available. However, the percentage of coincidentally correct tests may have an effect on the fault localization effectiveness. In the study for evaluating the effect of test objectives, we did not control the ratio of the number of failing test cases to the number of passing test cases. It was not possible to control the ratio and satisfy the coverage criteria at the same time. The variation of the ratio could have an effect on the fault localization effectiveness. Abreu et al. [2] showed that increasing the number of failing test cases, up to a certain number, improves the effectiveness of the Ochiai

approach. Thus, increasing the ratio of the number of failing to the number of passing test cases may reduce the opportunity of improving fault localization effectiveness by addressing coincidental correctness.

In the study for evaluating the change in fault localization effectiveness with different types of test suites, we selected the test suites for each benchmark from a large pool of test cases for the benchmark such that each selected test suite achieved the same coverage as the pool of test cases. However, the pools for different benchmarks and program versions did not have the same level of coverage. Thus, the coverage of the test suites was different for different benchmarks, and the difference may have influenced the fault localization effectiveness for the benchmarks.

Construct: A threat to construct validity results from the fact that the percentage of code examined may not correctly represent the effort to locate the faulty statement. As a tester inspects statements in descending order of suspiciousness scores, the effect needed for determining the correctness of each statement may differ for different statements. The percentage of code examined does not address these differences in effort, although it is a currently accepted way of measuring effectiveness [8, 29].

Chapter 7

Conclusions

We summarize our conclusions on (1) the benefits of mitigating the effect of coincidentally correct test cases for fault localization, (2) the choice of the granularity of coverage spectra in classifying coincidentally correct test cases, (3) the comparison of the approaches for calculating suspiciousness scores, (4) the role of tester feedback in mitigating the effect of coincidental correctness, (5) the effect of the characteristics of test suites on our approaches, and (6) localizing multiple faults using our approaches.

Benefits of Mitigating the Effect of Coincidental Correctness: Overall, our approaches can improve the fault localization effectiveness. In our studies, the median improvement in fault localization effectiveness for all the approaches and benchmarks varied between 0% to 15%. For faulty statements that were close to the program entry point, the improvement in fault localization effectiveness was higher compared to other faulty statements. In general, the extent of the improvement depends on the accuracy of the classification of coincidentally correct test cases and on the extent to which the presence of coincidentally correct test cases lowers the rank of the faulty statement.

Choice of the Granularity of Coverage Spectra: For randomly selected test suites, classifying test cases with a coarse-grained coverage spectrum, such as function coverage, led to more accurate classification compared to other fine-grained coverage spectra, such as statement or branch coverage. The coarse-grained coverage spectrum resulted in a more accurate classification by abstracting away unimportant differences between the passing test cases and the actual coincidentally correct test cases. However, when coverage adequate test suites were used, the coarse-grained coverage spectrum failed to magnify the important dif-

ferences for some coincidentally correct test cases and resulted in less accurate classification compared to the fine-grained coverage spectra.

Comparison of the Approaches for Calculating suspiciousness Scores: Our Jaccard similarity based approach of using the classified coincidentally correct test cases to calculate suspiciousness scores is more effective compared to calculating suspiciousness scores by removing the coincidentally correct tests. The former approach resulted in a median improvement in effectiveness of up to 15%, while the latter approach resulted in a median improvement in effectiveness of only up to 2%.

Role of Tester Feedback: Utilizing tester feedback to determine the lower bound for the number of actual coincidentally correct test cases can result in a more precise classification. However, repeatedly using tester feedback to determine the lower bound can lead to classifications that are too conservative and result in many false negatives. Therefore, tester feedback can be used, but only once to check for the presence of coincidentally correct test cases.

Effect of the Characteristics of Test Suites: Addressing coincidental correctness can result in more improvement in effectiveness if a higher percentage of the passing test cases is coincidentally correct. We observed that effectiveness increased by up to 10% when the percentage of coincidentally correct tests increased by 40%. However, when most of the test suite was coincidentally correct, addressing coincidental correctness did not result in any improvement in fault localization effectiveness. Effectiveness was lower for coverage-adequate test suites compared to random test suites.

Localization of Multiple Faults: In the presence of two faults, addressing coincidentally correct test cases can improve the fault localization effectiveness for at least one fault. Effec-

tiveness for both faults improved in at least 50% of the cases. In at least 30% of the faulty programs, effectiveness improved for one fault but not for the other.

Chapter 8

Future Work

This dissertation unveils new research directions and poses new research questions. We discuss the short term research opportunities for improving the work presented in this dissertation as well as the long term opportunities for applying our research to improve the state of the art of software debugging.

8.1 Short Term Research

In the short term, our approaches for classifying test cases and for calculating suspiciousness scores can be improved. Our fault localization approaches can be extended to better support the localization of multiple faults. Large scale empirical studies addressing various confounding factors can also be performed.

8.1.1 Improving the Classification Approaches

We used the Euclidean distance metric with the k-means clustering technique for classifying coincidentally correct test cases. Other distance metrics that have been used by researchers for clustering, such as Minkowski metric [51] and mutual neighbor distance [20], can be explored. Different clustering approaches, such as hierarchical clustering [25], search-based clustering [31], and graph-theoretical clustering [55], can also be used.

We used the heuristic that coincidentally correct test cases are similar to the failing test cases. During our experiments, we observed exceptions to this heuristic. Therefore, rather than relying on the domain knowledge, a supervised learning based approach, such as support vector machines (SVM) [22], can be used to classify coincidentally correct test cases. SVMs can be trained with test cases labeled coincidentally correct or not coincidentally correct for a set of known faults. Then, the trained SVMs can be used for classifying coincidentally

correct test cases for unknown faults. One challenge is to find a suitable representation for the test cases for training and classifying. The representation should capture the differences between coincidentally correct test cases and the passing test cases that are not coincidentally correct. Another challenge is to select the known faults for training. The known faults used for training should cover a fault model such that the SVM is trained to classify coincidentally correct test cases for all types of faults.

8.1.2 Improving the Approaches for Calculating Suspiciousness Scores

Our current approaches for calculating suspiciousness scores use the number of passing, failing, and classified coincidentally correct test cases, and do not take into consideration what caused a test case to be coincidentally correct. Thus, a possible improvement would be to identify the statements/branches/conditions/run-time values that participate in suppressing the local failure states in coincidentally correct test cases through examining the differences between a failing and a coincidentally correct test case. The faulty statement must have some dependence on the elements that participate in suppressing the failure state. Therefore, these elements can be used as slicing criteria for calculating static and dynamic slices to localize the faulty statement. The resulting slices can be more accurate than slices derived using program outputs as the slicing criteria.

8.1.3 Localizing Multiple Faults

In the presence of multiple faults, the binary classification of passing test cases as coincidentally correct or not coincidentally correct is not adequate. A test case may be coincidentally correct with respect to one fault, but may not be coincidentally correct with respect to another fault. To mitigate the effect of coincidental correctness in the presence of multiple faults, passing test cases can be classified into multiple classes such that each class contains the test cases that are coincidentally correct with respect to one fault. Existing techniques for failure clustering [35] can be used to classify the failing test cases into classes $F_1, \ldots, F_N$, such that each class contains the failing test cases that failed due to the same fault. Then,

the passing test cases can be classified into classes $P_1, P_2, \ldots, P_{N+1}$ such that the class P_i contains the test cases that are coincidentally correct with respect to the same fault that caused the failing test cases in F_i, and the class P_{N+1} contains the passing test cases that are not coincidentally correct with respect to any fault.

Once the classes of failing and passing test cases are derived, fault localization can be performed. For each statement, for $i = 1, \ldots, N$, a suspiciousness score can be calculated using P_i, F_i, and P_{N+1}. The maximum suspiciousness score of each statement can be used to derive the ranked list of statements. This approach can be evaluated with programs containing multiple faults.

8.1.4 Further Studies

Our studies revealed that the fault localization effectiveness of our approaches depends on the type and the location of a fault. The fault localization effectiveness of our approaches needs to be evaluated by systematically varying the type and the location of faults. A mutation tool, such as MuJava [36], can be used to seed mutation faults. We have already developed concrete and measurable fault properties, such as accessibility and impact, which characterize fault type and location. The correlation of accessibility and impact of faults to the fault localization effectiveness of our family of approaches needs to evaluated.

The fault localization approaches also need to be evaluated with large scale benchmarks and using new types of test suites, such as data-flow coverage adequate test suites.

8.2 Long Term Research

In the long term, the use of spectrum-based fault localization approaches for localizing faults in large scale distributed systems can be explored. Analysis of coincidental correctness can be applied to improve the existing approaches for automatically repairing programs.

8.2.1 Localizing faults in Large Scale Distributed Systems

Spectrum-based fault localization approaches have primarily been studied with small scale standalone systems. Existing approaches [40] for fault localization in distributed systems collect and analyze traces to identify nodes and links that are most likely to be faulty. Research can address the following questions on localizing faults in distributed systems:

- How can spectrum based fault localization approaches be applied to large scale distributed systems?
- To what extent do coincidentally correct test cases affect the fault localization effectiveness in distributed systems?
- To what extent is our heuristic for classifying coincidentally correct test cases applicable to test cases in distributed systems?
- Can we leverage the knowledge of system topology, caching, and replication for classifying coincidentally correct traces?
- How can we adapt spectrum-based fault localization approaches to localize faults online by continuously monitoring a running system and analyzing the generated traces?

8.2.2 Automatic Program Repair Based on Coincidental Correctness

Researchers have proposed approaches [4, 19, 30, 49, 50] for automatically repairing a faulty program. These approaches represent the problem of repairing a program as a search problem to find the correct repair among all possible changes in a program. The approaches generate tentative repairs and evaluate the fitness of a tentative repair as a measure of how close the tentative repair is to the actual repair. These approaches require many program executions, and thus, often do not scale for large programs.

We believe that coincidentally correct test cases can be analyzed to derive possible repairs for a fault. Recall that a coincidentally correct test case can result if a local failure

state is suppressed. The statements/branches/conditions/run-time values that participate in suppressing the local failure states for a coincidentally correct test case can be determined through examining the differences between a failing and the coincidentally correct test case. Analyzing the effect of these statement/branches/conditions can provide the condition that needs to satisfied to suppress the effect of the fault. Possible repairs for the fault can be derived from the condition.

References

[1] Software-artifact infrastructure repository. `http://sir.unl.edu`.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, Windsor, UK, 2007.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, Auckland, New Zealand, 2009.

[4] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation*, pages 1427–1434, Dublin, Ireland, 2011.

[5] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pages 143–151, Toulouse, France, 1995.

[6] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the 22nd Annual Symposium on Computational Geometry*, pages 144–153, Sedona, Arizona, USA, 2006.

[7] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 49–60, Trento, Italy, 2010.

[8] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Fault localization for dynamic web applications. *IEEE Transactions on Software Engineering*, 38(2):314–335, 2012.

[9] George K. Baah, Andy Podgurski, and Mary Jean Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 189–200, Seattle, WA, USA, 2008.

[10] Aritra Bandyopadhyay and Sudipto Ghosh. On the effectiveness of the tarantula fault localization technique for different fault classes. In *Proceedings of the 13th IEEE International Symposium on High-Assurance Systems Engineering*, pages 317–324, Boca Raton, FL, USA, 2011.

[11] Aritra Bandyopadhyay and Sudipto Ghosh. Proximity based weighting of test cases to improve spectrum based fault localization. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 420–423, Lawrence, KS, USA, 2011.

[12] Aritra Bandyopadhyay and Sudipto Ghosh. Tester feedback driven fault localization. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pages 41–50, Montreal, QC, Canada, 2012.

[13] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering*, pages 82–91, Shanghai, China, 2006.

[14] Rafael Caballero, Christian Hermanns, and Herbert Kuchen. Algorithmic debugging of Java programs. *Electronic Notes in Theoretical Computer Science*, 177:75–89, June 2007.

[15] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 34–44, Vancouver, Canada, 2009.

[16] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 528–550, Glasgow, Scotland, 2005.

[17] Nicholas DiGiuseppe and James A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, pages 210–220, Toronto, Canada, 2011.

[18] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[19] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, pages 947–954, 2009.

[20] K. Chidananda Gowda and G. Krishna. Agglomerative clustering using the concept of mutual nearest neighbourhood. *Pattern Recognition*, 10(2):105–112, 1978.

[21] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, Long Beach, USA, 2005.

[22] Marti A. Hearst, ST Dumais, E Osman, John Platt, and Bernhard Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, 1998.

[23] Susan Horwitz, Ben Liblit, and Marina Polishchuk. Better debugging via output tracing and callstack-sensitive slicing. *IEEE Transactions on Software Engineering*, 36:7–19, 2010.

[24] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[25] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[26] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3), 1999.

[27] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007.

[28] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 16–26, London, UK, 2007.

[29] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, Orlando, Florida, 2002.

[30] Christian Kern and Javier Esparza. Automatic error correction of Java programs. In *Formal Methods for Industrial Critical Systems*, pages 67–81. Springer, 2010.

[31] W. L. G. Koontz, P. M. Narendra, and K. Fukunaga. A branch and bound clustering algorithm. *IEEE Transactions of Computers*, 24(9):908–915, September 1975.

[32] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of 14th IEEE International Symposium on Software Reliability Engineering*, pages 442–456, Denver, Colorado, 2003.

[33] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, New York, NY, USA, 2003. ACM.

[34] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, Chicago, IL, USA, 2005.

[35] Chao Liu, Xiangyu Zhang, and Jiawei Han. A systematic study of failure proximity. *IEEE Transactions on Software Engineering*, 34:826–843, November 2008.

[36] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: a mutation system for Java. In *28th International Conference on Software Engineering*, pages 827–830, Shanghai, China, 2006.

[37] Wes Masri and Rawad Abou Assi. Cleansing test suites from coincidental correctness to enhance fault-localization. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation*, pages 165–174, Paris, France, 2010.

[38] Hsin Pan and Eugene H. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116P, Purdue University, 1992.

[39] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 245–254, Cape Town, South Africa, 2010.

[40] Pawan Prakash, Ramana Rao Kompella, Venugopalan Ramasubramanian, and Ranveer Chandra. dfault: Fault localization in large-scale peer-to-peer systems. In *Proceedings of the 11th ACM/IFIP/USENIX International Middleware Conference*, pages 252–272, Bangalore, India, 2010.

[41] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbour queries. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, 2003.

[42] Debra J. Richardson and Margaret C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Transactions on Software Engineering*, 19:533–553, 1993.

[43] Raul Santelices, James A. Jones, Yu Yanbing, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 56–66, Vancouver, Canada, 2009.

[44] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30:263–272, September 2005.

[45] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[46] Josep Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42:976–991, November 2011.

[47] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[48] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55, Vancouver, Canada, 2009.

[49] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[50] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

[51] D. Randall Wilson and Tony R. Martinez. Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, 6(1):1–34, 1997.

[52] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 42–51, Washington, DC, USA, 2008.

[53] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 201–212, Chicago, IL, 2009.

[54] Yanbing Yu, James A. Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering*, pages 201–210, Leipzig, Germany, 2008.

[55] C. T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions of Computers*, 20(1):68–86, January 1971.

[56] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.