STATISTICAL FAULT LOCALIZATION
AND CAUSAL INTERACTIONS

by

SHIH-FENG SUN

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2017
CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

SHIH-FENG SUN

candidate for the degree of Doctor of Philosophy*.

Committee Chair
Dr. Andy Podgurski

Committee Member
Dr. Gultekin Ozsoyoglu

Committee Member
Dr. Soumya Ray

Committee Member
Dr. Francis Merat

Date of Defense
November 4, 2016

*We also certify that written approval has been obtained for any proprietary material contained therein.
# Table of Contents

1 Introduction .......................................................................................................................... 1

1.1 Statistical Fault Localization Techniques ..................................................................... 1

1.2 Contributions .................................................................................................................. 3

1.3 Organization of the Dissertation .................................................................................... 4

2 Background .......................................................................................................................... 5

2.1 Basic CBSFL Notation ................................................................................................. 5

2.2 Causal Inference Concepts .......................................................................................... 6

2.3 Potential Outcome Framework ...................................................................................... 8

2.4 Control Flow Graph and Program Dependence Graph ............................................. 9

2.5 Baah et al.’s Causal SFL Method and Related Research ......................................... 11

3 Properties of Effective Metrics for Coverage-Based Statistical Fault Localization .. 13

3.1 Introduction .................................................................................................................... 13

3.2 Background .................................................................................................................... 14

3.2.1 Relative Measures and Related Measures ............................................................... 14

3.2.2 Liblit et al.’s “Importance” and “Increase” Measures .............................................. 17

3.2.3 Causal Effect and Associational Risk Difference .................................................... 18

3.3 Analysis of CBSFL Metrics That Have Performed Well in Many-Metric Comparisons ............................................................... 20

3.3.1 The F1-measure and Related Metrics .................................................................. 20
3.3.2 Metrics Related to the Ochiai Metric .......................................................... 22
3.3.3 Optimized Pattern-Similarity (Pattern-Similarity2) .................................... 24
3.3.4 Summarizing the Analysis ......................................................................... 26
3.4 New Metrics Derived from Associational Risk Difference ............................... 28
  3.4.1 Associational Risk Difference and Relative Precision ............................... 28
  3.4.2 New Metrics Involving Associational Risk Difference .............................. 30
3.5 Empirical Study ............................................................................................ 31
  3.5.1 Study Setup ............................................................................................ 32
  3.5.2 Results for RQ1: Metrics $w \times RP$ or $w \times max(RP, RR)$ over $RP$ ......... 35
  3.5.3 Results for RQ2: Metrics $w \times precision$ over Precision ....................... 39
  3.5.4 Results for RQ3: Metrics $w \times RP$ over Metrics $w \times precision$ .......... 41
  3.5.5 Results for RQ4: $ARD$ over Precision ................................................ 46
  3.5.6 Results for RQ5: Metrics $w \times ARD$ over $ARD$ ..................................... 47
  3.5.7 Results for RQ6: Metrics $w \times ARD$ over Metrics $w \times precision$ ...... 51
  3.5.8 Discussion ............................................................................................... 54
  3.5.9 Study Limitations ................................................................................... 57
3.6 Related Work .................................................................................................. 59
3.7 Conclusion ....................................................................................................... 60
4 Causal Interactions for Program Statement Interactions .................................. 62
  4.1 Introduction .................................................................................................. 62
List of Tables

Table 3-1. Details of Subject Programs ................................................................. 34

Table 3-2. Percentage Reductions for Single-fault Programs ($w \times RP$ or $w \times \max(RP, RR)$ vs. $RP$) ................................................................................................................ 36

Table 3-3. Percentage Reductions for Multiple-fault Programs ($w \times RP$ or $w \times \max(RP, RR)$ vs. $RP$). ................................................................................................................ 37

Table 3-4. Percentage Reductions for Single-fault Programs ($w \times prec$ vs. $prec$)....... 40

Table 3-5. Percentage Reductions for Multiple-fault Programs ($w \times prec$ vs. $prec$). .... 40

Table 3-6. Percentage Reductions for Single-fault Programs ($w \times RP$ vs. $w \times prec$)... 42

Table 3-7. Percentage Reductions for Multiple-fault Programs ($w \times RP$ vs. $w \times prec$). 43

Table 3-8. Percentage Reductions for Both Single-fault and Multiple-fault Programs ($ARD$ vs. $ prec$). ................................................................................................................ 47

Table 3-9. Percentage Reductions for Single-fault Programs ($w \times ARD$ vs. $ARD$)........ 48

Table 3-10. Percentage Reductions for Multiple-fault Programs ($w \times ARD$ vs. $ARD$). .. 49

Table 3-11. Percentage Reductions for Single-fault Programs ($w \times ARD$ vs. $w \times prec$).
........................................................................................................................................... 51

Table 3-12. Percentage Reductions for Multiple-fault Programs ($w \times ARD$ vs. $w \times prec$). ................................................................. 52

Table 3-13. An Example for Two Elements Having the Same Precision Scores .......... 55

Table 4-1. Types of Causal Interaction........................................................................ 70

Table 4-2. Summary of Subject Programs.................................................................... 100

Table 4-3. Average Percentage Cost for Locating FCIs ($|AIM|$)................................. 102
Table 4-4. Average Percentage Cost for Locating FRIs (|AIM|) ................................. 103
Table 4-5. Average Percentage Cost for Locating FCIs (|t-score|) ............................... 103
Table 4-6. Average Percentage Cost for Locating FRIs (|t-score|) ............................... 104
List of Figures

Figure 2-1. Causal DAG with backdoor path ................................................................. 8

Figure 3-1. An example program for comparing precision to ARD ......................... 29

Figure 3-2. Percentage reductions for single-fault programs (w × RP or w ×
mmax(RP, RR) vs. RP). .............................................................................................. 37

Figure 3-3. Percentage reductions for multiple-fault programs (w × RP or w ×
mmax(RP, RR) vs. RP). .............................................................................................. 38

Figure 3-4. Percentage reductions for single-fault programs (w × RP vs. w × prec)..... 42

Figure 3-5. Percentage reductions for multiple-fault programs (w × RP vs. w × prec). 44

Figure 3-6. Percentage reductions with number of faults in programs....................... 45

Figure 3-7. Percentage reductions for single-fault programs (w × ARD vs. ARD)....... 49

Figure 3-8. Percentage reductions for multiple-fault programs (w × ARD vs. ARD)..... 50

Figure 3-9. Percentage reductions for single-fault programs (w × ARD vs. w × prec) .. 52

Figure 3-10. Percentage reductions for multiple-fault programs (w × ARD vs. w × prec).
................................................................................................................................. 53

Figure 4-1. A piece of Mozilla fix code........................................................................... 65

Figure 4-2. An example of FEP and FCI ................................................................. 81

Figure 4-3. A FRI vs. a FI1 interaction ......................................................................... 83

Figure 4-4. A FCI vs. a FI2 interaction ......................................................................... 84

Figure 4-5. A FCI and a FRI vs. a FI4 interaction ....................................................... 85

Figure 4-6. Algorithm to locate interaction pairs ..................................................... 89

Figure 4-7. Interaction pair information ..................................................................... 91
Figure 4-8. A causal graph with two exposures............................................................... 92
Acknowledgements

First, I would like to thank my advisor, Dr. Andy Podgurski for his patience, guidance and support during my Ph.D. He led me into software testing and causal inference fields. Many ideas in my research are raised and inspired through our numerous discussions and meetings. I would like to thank my committee members, Dr. Gultekin Ozsoyoglu, Dr. Soumya Ray, and Dr. Francis Merat for their comments and suggestions about my research.

I would like to thank the Government of Republic of China (Taiwan) for the support which helps me to concentrate on my study at Case Western Reserve University. I would like to thank all members in my group, Dr. Ray-young Chang, Dr. Gang Shu, Dr. Zhuofu Bai, and Tim Henderson. Each discussion provided me with different perspectives to solve problems. I would also like to thank all my friends in Cleveland, OH. Hanging out and chatting with them helped me to release the pressure during my study.

Finally, I would like to thank my parents and every family member who encourages me to keep moving forward. Especially, I would like to thank my wife Pei-Jiun Huang. She always stands by me.
Statistical Fault Localization and Causal Interactions

Abstract

by

SHIH-FENG SUN

Coverage-based statistical fault-localization (CBSFL) techniques have been proposed for characterizing the “suspiciousness” of program statements based on code-coverage profiles and PASS/FAIL labels for a set of test or operational executions. The resulting suspiciousness scores are typically intended to be used to rank statements for inspection by developers, on the assumption that statements which cause labeled failures will tend to receive high ranks. Many CBSFL metrics have been proposed.

This dissertation examines two issues related to proposed CBSFL metrics. First, it examines the structure of some of the most effective of the proposed metrics, when they are expressed as expressions involving relatively simple probabilities. It is shown that these metrics have a common structure that makes it easier to understand their strengths and weaknesses. The second issue examined here is the impact, on the occurrence of observable program failures, of dynamic interactions between the execution of one program statement and the execution of another. To clarify this issue, event-based definitions of fault-revealing and fault-concealing interactions (FRIs and FCIs) in programs are presented and they are then related to both the theory of causal interactions developed by causal inference researchers and to characterizations of fault-interactions and failed error propagation developed by software researchers. It is shown that statistical tests developed to detect causal interactions are applicable to detecting FRIs and FCIs. A
preliminary approach, based on causal interaction tests, is proposed for locating statements involved in both FRIs and FCIs.
1 INTRODUCTION

1.1 Statistical Fault Localization Techniques

Many coverage-based statistical fault-localization (CBSFL) metrics have been proposed for characterizing the “suspiciousness” of program statements based on code-coverage profiles and PASS/FAIL labels for a set of test or operational executions [1][2][3][4][5][6][7][8]. Essentially, these metrics measure the strength of statistical associations between the coverage of individual statements and the occurrence of program failures [9]. The resulting suspiciousness scores are typically intended to be used to rank statements for inspection by developers, on the assumption that statements which cause labeled failures will tend to receive high ranks. CBSFL metrics have the advantages of employing easily collected execution profiles and of being applicable to profiles of different granularity levels (e.g., statement, basic block, function). Most of the metrics can be computed very quickly, and many CBSFL metrics are proposed.

This dissertation examines two issues related to CBSFL metrics. First, the common structure of the most effective metrics suggested in different studies is examined. These commonalities can be used to better understand the metrics’ strengths and weaknesses, which have been little explored in the literature. The second issue examined here is the impact, on the occurrence of observable program failures, of dynamic interactions between the execution of one program statement and the execution of another.

To explore the commonalities between effective metrics, we consider the results of recent studies which have empirically investigated the comparative effectiveness of a
substantial number of existing or potential CBSFL metrics [1][5][7][8]. Although these studies should not be considered definitive, because they involved limited varieties of subject programs and faults, the metrics that they suggest are most effective, notably symmetric Klosgen [1], Ochiai [1][8], Sorensen-Dice [5], and Jaccard [5], do share some interesting properties with each other, with Baah et al.’s CBSFL approach based on causal inference methodology [9][10], and with measures used in other fields such as data mining [11][12][13][14][15] and epidemiology [16]. We explore these commonalities, with the goal of clarifying the properties that make some CBSFL metrics more effective than others. In particular, we show that the most effective metrics incorporate distinct terms (subexpressions) that may be viewed as implicitly addressing two goals: (1) providing an estimate of the average causal effect of a program element $e$ on the occurrence of failures (its average failure-causing effect or AFCE), which may be subject to confounding bias due to other program elements, and (2) decreasing the AFCE estimate, if necessary, to reflect evidence that $e$ is not the cause of all observed failures. We also discuss the best ways to accomplish these goals and how further improvements in SFL metrics might be achieved.

For the issue raised by interactions between program statements, it is known that some faulty statements induce observable failures, only (or with greater frequency) when they are executed together with particular, independently-executable statements. Call such an event a fault-revealing (statement) interaction (FRI). It is also known that some faults are concealed, only (or with greater frequency) when they are executed together with particular, independently-executable statements. Call this event a fault-concealing (statement) interaction (FCI). Having computed conventional fault localization scores for individual statements (ideally using causal inference methodology), it is desirable to also
test for both FRIs and FCIs involving pairs of statements, in order to better understand the conditions under which possible faults are revealed or concealed.

The theory and methodology developed for studying causal interactions [17], e.g. in medical research, also provides a systematic way to analyze statement interactions in programs. In particular, we can use causal interaction tests, *additive interaction test* (AI test) and *compositional epistasis test* (CE test), to detect the occurrence of statement interactions in programs. Developers can use the AI test first to locate FCIs and FRIs and then use the CE test to decide a specific interaction type. The formula used with the CE test varies depending on different potential outcome combinations, thus helping developers identify a specific type of interaction.

### 1.2 Contributions

This dissertation makes the following contributions:

1. Algebraic and probabilistic analyses on the effective metrics are conducted to identify their key properties.

2. New refined metrics are introduced, which are in the form of a product of *weight* and *relative precision* and a product of *weight* and *associational risk difference*.

3. An empirical study is conducted to assess the relative importance of those important properties.

4. Original definitions of fault-revealing and fault-concealing interactions are presented in terms of generic runtime events that can be detected by execution profiling or monitoring.
5. The relationships of these definitions to previous characterizations of general causal interactions [18] and of software fault-interactions [19][20] and failed error propagation [21] are explained.

6. Established statistical tests for general causal interactions are shown to be relevant to the detection of fault-revealing and fault-concealing interactions.

7. A preliminary approach, based on causal interaction tests, is proposed for locating statements involved in both fault-revealing interactions and fault-concealing interactions.

1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows:

Chapter 2 defines basic CBSFL notation and presents related background about applying causal inference on statistical fault localization.

Chapter 3 presents the properties of effective metrics for coverage-based statistical fault localization.

Chapter 4 presents how the causal interaction theory is applied to illustrate statement interactions in programs. Also, a method is proposed to identify program statement interactions.

Chapter 5 concludes this dissertation.
2 BACKGROUND

2.1 Basic CBSFL Notation

The CBSFL metrics that we consider are composed of simpler expressions, involving various counts computed from execution profiles or labels, that in several cases are actually estimators for marginal and conditional probabilities (defined over a test suite or an input distribution) involving coverage or non-coverage of a program element, on one hand, and/or involving program failure or success, on the other hand. Where possible, we express CBSFL metrics in terms of meaningful probabilities to aid interpretation. We define them for generic “coverable” program elements $e$ such as statements, basic blocks, or subprograms. It is assumed that, for a given program, a metric is applied to elements of only one kind. (In the empirical study we report on, the elements were methods.)

We first define some basic counts and expressions that will be used in our analysis: $n$ is the total number of tests or field executions observed, labeled, and profiled (we shall refer to both simply as tests); $p$ is the total number of tests that pass (succeed); $f$ is the total number of tests that fail; $n_e$ is the total number of tests that cover program element $e$; $p_e$ is the number of tests that pass and also cover $e$; $f_e$ is the number of tests that fail and also cover $e$. In probability expressions, $P$ denotes the event that a test passes, $F$ denotes the event that a test fails, and $e$ denotes the event that a test covers element $e$. Thus, $Pr(F)$ is the probability of program failure, where $Pr(F) \approx \frac{f}{n}$; $Pr(e)$ is the probability that element $e$ is covered, where $Pr(e | F) \approx \frac{f_e}{n_e}$; $Pr(e | F)$ is the probability that $e$ is covered given that the program fails, also called recall [9][14] or sensitivity [14], where $Pr(e | F) \approx \frac{f_e}{f}$; and
Pr(F ∩ e) is the probability that e is covered and the program fails, also called support [11][15], where Pr(F ∩ e) ≈ \( \frac{f_e}{n} \). By Bayes’ Theorem, these three probabilities are closely related:

\[
Pr(F \mid e) = \frac{Pr(F \cap e)}{Pr(e)} = \frac{Pr(e \mid F) Pr(F)}{Pr(e)}
\]

The probability Pr(F | e) is the associational risk [16] of failure given that program element e is covered. It is not necessarily equal to the causal risk, because of the possibility of confounding and other forms of bias [22]. Note that if Pr(F) > 0 but Pr(e | F) < 1 then element e cannot be the only cause of failures. We shall see that this makes Pr(e | F) useful for CBSFL in combination with Pr(F | e). Observe that \( 1 - Pr(e \mid F) = Pr(e \text{ not covered} \mid F) = Pr(\neg e \mid F) \) is a lower bound on the probability that e is not the cause of a realized failure, because even if e is covered during a failing execution, it may not be the failure’s cause.

2.2 Causal Inference Concepts

Causal inference research seeks to provide methods for making unbiased estimates of the causal effects of certain study variables, such as those representing exposures or treatments in observational studies or randomized experiments, upon variables representing outcomes of interest. A causal DAG (causal graph, causal diagram) [22] is a directed acyclic graph (DAG) describing the known or assumed causal relationships among a set of study variables, where each vertex represents a random variable and an edge \( X \rightarrow Y \) indicates that variable X potentially or actually causes variable Y. A causal path between X and another variable Z is a directed path from X to Z. Figure 2-1 is a simple
example of a causal DAG. One of the primary concerns in causal inference is the elimination of confounding bias [22], which is estimation bias caused by the presence of one or more common causes of the exposure variable and the outcome variable. Confounding of the estimated average causal effect of an exposure variable $E$ (e.g., a coverage indicator for a given statement) upon an outcome variable $Y$ (e.g., indicating whether a failure occurred) is due to the presence of at least one common cause $C$ of $E$ and $Y$. Causal inference theory provides a number of graph-theoretic results that characterize, in terms of causal DAGs, the conditions for making unbiased causal inferences. For example, Pearl’s Back-Door Adjustment Theorem [22] shows that confounding bias can be eliminated by adjusting for (conditioning on) a set of covariates $C$ that blocks all “backdoor paths” between the exposure variable $E$ and the outcome variable $Y$ in a causal DAG. A backdoor path between $E$ and $Y$ is a non-causal path that begins with an edge into $T$. For example, the path $E \leftarrow C \rightarrow Y$ in Figure 2-1 is a backdoor path. Intuitively, such a path indicates a possible statistical association between the treatment variable and the outcome variable that could distort estimation of the causal effect of $E$ on $Y$. A set of covariates $C$ is said to block a path in a causal graph if the path (1) contains a chain $\rightarrow U \rightarrow$ or a fork $\leftarrow U \rightarrow$, where $C$ is in $U$, or (2) it contains a collider $\rightarrow U \leftarrow$, where $U$ and its descendants are not in $C$. In Figure 2-1, the backdoor path $E \leftarrow C \rightarrow Y$ is blocked by $C$. 

2.3 Potential Outcome Framework

The potential outcome framework [23] is a theoretical framework for estimating the causal effect of a treatment variable upon an outcome variable, using data from an observational study or randomized experiment. Let $E^i$ be a binary variable whose value indicates the exposure received by individual $i$, for $i = 1 \ldots n$. Let $Y^i$ be a variable whose values represent the possible outcomes for unit $i$. By dropping the index $i$, random variables $E$ and $Y$ are obtained whose values vary over the study population. Prior to administering or determining the actual exposure for a unit $i$, we can imagine that $i$ has two potential outcome random variables $Y_1^i$ and $Y_0^i$, corresponding to the two possible exposure values. Assuming that each unit receives exactly one of the possible exposure value, only one potential outcome is observed from each unit. The other is counterfactual — counter to the facts. However, the expected values $E[Y_1]$ and $E[Y_0]$ and the average causal effect $E[Y_1] - E[Y_0]$ for a study population can often be estimated without bias using sound causal inference techniques [22]. The potential outcome framework [23] is a theoretical framework for estimating the causal effect of a treatment variable upon an outcome variable, using data from an observational study or randomized experiment. Let $E^i$ be a binary random variable.
whose value indicates the exposure received by individual \(i\), for \(i = 1 \ldots n\). Let \(Y^i\) be a random variable whose values represent the possible outcomes for unit \(i\). By dropping the index \(i\), random variables \(E\) and \(Y\) are obtained whose values vary over the study population. Prior to administering or determining the actual exposure for a unit \(i\), we can imagine that \(i\) has two potential outcome random variables \(Y^i_1\) and \(Y^i_0\), corresponding to the two possible exposure values. Assuming that each unit receives exactly one of the possible exposure values, only one potential outcome is observed from each unit. The other is counterfactual — counter to the facts. However, the expected values \(E[Y_1]\) and \(E[Y_0]\) and the average causal effect (ACE) for a study population can often be estimated without bias using sound causal inference techniques [22]. The average causal effect in the population is used to represent individual’s causal effect because the causal effect of a randomly selected individual is equal to the average causal effect across individuals in the population [16], and we have

\[
ACE = E[Y_1] - E[Y_0]
\]

(2-1)

where \(E[\cdot]\) denotes the expectation or the population average. Note that if the potential outcomes \(Y^i_1\) and \(Y^i_0\) are binary, \(E[Y_1]\) and \(E[Y_0]\) are the counterfactual probabilities \(\Pr(Y_1 = 1)\) and \(\Pr(Y_0 = 1)\), respectively.

2.4 Control Flow Graph and Program Dependence Graph

The program dependence graph (PDG) [24] is a directed graph whose vertices represent program statements and whose edges represent data and control dependences between statements. A PDG therefore contains a data dependence subgraph and a control
dependence subgraph. A system dependence graph (SDG) [25] for a software system consists of PDGs for each subprogram of the system, connected by interprocedural dependence edges. The PDG of a program can be constructed by analyzing its control flow graph (CFG), annotated to indicate where variables are defined and used [24]. The CFG is a directed graph in which each vertex represents a statement, instruction, or basic block and each edge represents a possible flow of control. We assume that in a CFG: (1) the maximum outdegree of any vertex is two, (2) there are two distinguished vertices, the initial vertex \( v_{in} \) and the final vertex \( v_{fin} \), and (3) every vertex occurs on some walk from \( v_{in} \) to \( v_{fin} \). Let \( P \) be a program or program unit having a control-flow graph \( CFG(P) \). For a statement \( s_i \) in \( P \), we shall sometimes also denote the corresponding vertex in \( CFG(P) \) by \( s_i \) and refer to it as a statement.

We now define some terms necessary for formally defining different types of control dependence [26][27]. Vertex \( v \) dominates vertex \( u \), denoted as \( v \ dom \ u \), if every \( v_{in} \) to \( v_{fin} \) walk contains \( v \). Vertex \( u \) postdominates vertex \( v \) if every \( v \) to \( v_{fin} \) walk contains \( u \). The immediate postdominator of a vertex, denoted \( ipd(v) \), is the vertex that is the first proper postdominator of \( v \) to occur on every \( v \) to \( v_{fin} \) walk. Vertex \( u \) is control dependent on vertex \( v \) if there exists a \( v \) to \( u \) walk not containing \( ipd(v) \). For \( L \in \{true, false\} \), a vertex \( u \) is directly control dependent on vertex \( v \) with label \( L \), denoted \( u \ dcd_L \ v \), if \( v \) has successors \( v' \) and \( v'' \) such that \( u \) postdominates \( v' \), where the decision edge \( (v, v') \) has truth value \( L \), but \( u \) does not postdominate \( v'' \). If \( u \ dcd_T \ v \) or \( u \ dcd_F \ v \), we say that \( u \) is directly control dependent on \( v \), denoted \( u \ dcd \ v \). Vertex \( u \) is directly forward control dependent on vertex \( v \) with label \( L \), denoted \( u \ DFCDL \ v \), if \( u \ dcd_L \ v \) and \( \neg(u \ dom \ v) \). If \( u \ DFCDT \ v \) or \( u \ DFCDF \ v \), we say that \( u \) is directly forward control
dependent on \( v \), denoted \( u \) DFCD \( v \), and we say that \( v \) is a \textbf{DFCD predecessor} of \( u \) and write \( v = \text{pred}(u) \).

2.5 \textit{Baah et al.’s Causal SFL Method and Related Research}

Baah, Podgurski, and Harrold [9] pointed out that most \textit{statistical fault localization} (SFL) metrics are vulnerable to confounding bias, because they do not measure and adjust for confounding factors, and that such metrics are distorted indicators of failure causation. Baah \textit{et al.} [9] proposed a \textit{causal statistical fault localization} (CSFL) technique based on statistical regression. They used the PDG to derive a casual graph, applied the Back-Door Adjustment Theorem [22] to adjust for the confounding bias, and presented a linear regression model for AFCE of a statement \( s \). The model is

\[
Y = \beta_0 + \beta_1 T_s + \beta_2 C_s + \epsilon
\]

(2-2)

where the outcome variable \( Y \) indicates the outcome of the test (1: fails, 0: passes); the treatment variable \( T_s \) indicates the coverage of statement \( s \) (1: covered, 0: not covered); the covariate variable \( C_s \) indicates the coverage of the \textit{DFCD} predecessor \( \text{pred}(s) \) of statement \( s \) (1: covered, 0: not covered); \( \beta_0, \beta_1, \) and \( \beta_2 \) are coefficients, and \( \epsilon \) is a random error. The estimate \( \hat{\beta}_1 \) of the coefficient of \( \beta_1 \) is the AFCE estimator. Bai \textit{et al.} [28] demonstrated that the AFCE estimator \( \hat{\beta}_1 \) approximates the \textit{conditional association risk difference} [29] within the stratum of executions with \( C_s = 1 \),
This equals to the causal risk difference \[29\] within the stratum \(C_s = 1\) if the treatment effect of \(T_s\) on \(Y\) is unconfounded in that stratum.

**Positivity issues**

“Positivity” is an important condition for making valid causal inferences that was not discussed in Baah et al.’s paper \[9\]. The positivity condition holds with respect to a treatment \(T\) and a set \(X\) of covariates for confounding adjustment if the conditional probability \(\Pr(T = t|X = x)\) of receiving treatment value \(t\), given the covariate values \(X = x\), is greater than zero for all \(x\) with \(\Pr(X = x) > 0\). Bai et al. \[28\] showed that, with structured programs, Baah et al.’s technique \[9\] exhibits violations of positivity because it is not possible that a statement \(s\) is covered \((T_s = 1)\) when \(\text{pred}(s)\) is not covered \((C_s = 0)\). However, Bai et al. \[28\] proved that positivity usually holds for the stratum of executions with \(C_s = 1\) because \(\Pr(T_s = 1|C_s = 1) > 0\) and \(\Pr(T_s = 0|C_s = 1) > 0\).

Let us call this form of positivity **conditional positivity**. Conditional positivity is violated when, among other situations, a statement \(s\) is always covered \((T_s = 1)\) or is never covered \((T_s = 0)\) given that \(\text{pred}(s)\) is covered \((C_s = 1)\). Baah et al. also pointed out that if conditional positivity is violated, one of the two probability terms in \(2-3\) will be undefined, so the AFCE estimator \(\hat{\beta}_1\) will be undefined \[28\].
3 PROPERTIES OF EFFECTIVE METRICS FOR COVERAGE-BASED STATISTICAL FAULT LOCALIZATION

3.1 Introduction

A few studies have empirically investigated the comparative effectiveness of a substantial number of existing or potential CBSFL metrics [1][5][7][8]. Although these studies should not be considered definitive, because they involved limited varieties of subject programs and faults, the metrics that they suggest are most effective, notably symmetric Klosgen [1], Ochiai [1][8], Added Value [1], Sorensen-Dice [5], Jaccard [5], and Optimized Pattern-Similarity [7], do share some interesting properties with each other, with Baah et al.’s CBSFL approach based on causal inference methodology [9][10], and with measures used in other fields such as data mining [11][12][13][14][15] and epidemiology [16].

We analyze the structure of the aforementioned CBSFL metrics using basic algebra and probability theory in order to reveal their similarities and differences. Based on the results, we explore how some other metrics might be enhanced. We also report on an empirical study we conducted to better understand how the structure of the metrics affects their performance. It applied the metrics with data, provided by Steinmann et al. [4], from over 5,000 versions of single-fault programs and around 49,000 versions of multiple-fault programs created from 10 original subject programs and more than 5,000 injectable faults.
3.2 Background

3.2.1 Relative Measures and Related Measures

Lavrac et al. [15] pointed out that *relative measures* for evaluating learned rules give more information about the utility of a rule than do absolute measures. For instance, if in a prediction task the accuracy of a rule is lower than the relative frequency of the class it predicts, then the rule will perform badly, regardless of its absolute accuracy. There are two relative measures we will consider, *relative precision* and *relative recall*. We describe these measures using the CBSFL-specific notation introduced above. The *relative precision* \((RP)\) measure, also called *relative accuracy* [15] or *precision gain* [14], is

\[
RP = \Pr(F \mid e) - \Pr(F)
\]

(3-1)

Unlike (absolute) *precision*, \(RP\) varies from \(-1\) to \(1\). It can be interpreted as a precision gain relative to \(\Pr(F)\). Paraphrasing Yao et al. [13], one may consider \(\Pr(F)\) to be the prior probability of program failure and \(\Pr(F \mid e)\) to be the posterior probability of program failure given that \(e\) is covered. The difference of posterior and prior probabilities represents how much our estimate of the probability of failure should change after learning that \(e\) was covered. The rankings produced by sorting program elements according to \(\Pr(F \mid e)\) and \(RP\) are the same, however, because \(\Pr(F)\) is constant for a given program and sample of inputs [6]. The *relative recall* \((RR)\) measure, also called *relative sensitivity* [15], is

\[
RR = \Pr(e \mid F) - \Pr(e)
\]

(3-2)
Like $RP$, $RR$ also varies from $-1$ to $1$. It can be interpreted as a gain in sensitivity relative to $Pr(e)$.

We characterize the relationship between $RP$ and $RR$ by the following derivation:

$$RR = Pr(e | F) - Pr(e)$$

$$= \frac{Pr(F \cap e)}{Pr(F)} \times \frac{Pr(e)}{Pr(F)} - Pr(e) \times \frac{Pr(F)}{Pr(F)}$$

$$= \frac{Pr(e)}{Pr(F)} \times \left( \frac{Pr(F \cap e)}{Pr(e)} - Pr(F) \right)$$

$$= \frac{Pr(e)}{Pr(F)} \times (Pr(F | e) - Pr(F))$$

$$= \frac{Pr(e)}{Pr(F)} \times RP$$

(3-3)

Since $Pr(e)/Pr(F)$ is always positive, this implies that $RP$ and $RR$ are either both positive or both negative. Observe that if $Pr(e)/Pr(F) < 1$ then failures are more likely to occur than is coverage of program element $e$, hence there must be causes of failure other than $e$. In this case, $Pr(e)/Pr(F)$ serves to downweight the $RP$ component of $RR$. This property of $RR$ seems desirable for a CBSFL metric. If $Pr(e)/Pr(F) > 1$ then $RP$ is upweighted, which seems less desirable.

For a program element $e$, the quantity $PG = \frac{Pr(F | e) - Pr(F)}{n \times Pr(F)}$ may be viewed as the precision gain [14] provided by each test covering $e$. Assuming that the number of tests covering $e$ is $c_e$, the total gain in precision provided by these tests is $c_e \times PG$, which is approximately equal to the last expression for $RR$ in (3-3), since $Pr(e) \approx \frac{c_e}{n}$.

In this chapter, two CBSFL metrics $m_1$ and $m_2$ are considered to be score equivalent, denoted $m_1 \equiv m_2$, if for every program and test set they produce the same suspiciousness.
scores for each program element. The metrics are rank equivalent, denoted \( m_1 \equiv_R m_2 \), if for every program and test set they produce the same ranking of elements (with those having the same score receiving the same rank). Naturally, the scores of elements carry more information than their ranks. For example, a ranking does not reflect the presence of distinct clusters of program elements within which the elements have similar (but typically unequal) scores and between which the elements have dissimilar scores.

Naish et al. [6] defined a new metric called Ample2 (Amp2), which is a variation of the Ample metric [30] that omits the absolute value. We rewrite their definitions of the Ample2 metric in terms of our notation and then extend that equality with the following derivation:

\[
Amp2 \approx \frac{Pr(F \cap e)}{Pr(F)} - \frac{Pr(P \cap e)}{Pr(P)}
\]

\[
= \frac{Pr(F \cap e)}{Pr(F)} \times \frac{Pr(P)}{Pr(P)} - \frac{Pr(e) - Pr(F \cap e)}{Pr(P)} \times \frac{Pr(F)}{Pr(F)}
\]

\[
= \frac{Pr(F \cap e) \times Pr(P) - Pr(e) \times Pr(F) + Pr(F \cap e) \times Pr(F)}{Pr(P) \times Pr(F)}
\]

\[
= \frac{1}{Pr(P)} \times \frac{Pr(F \cap e) - Pr(e) \times Pr(F)}{Pr(F)}
\]

\[
= \frac{1}{Pr(P)} \times (Pr(e | F) - Pr(e))
\]

(3-4)

The second term in the last expression in derivation (3-4) is relative recall, so we may also write \( Amp2 = \frac{1}{Pr(P)} \times RR \). This proves that Ample2 and RR are rank equivalent, \( Amp2 \equiv_R RR \), because \( Pr(P) \) is a fixed value for a given program and test set.
3.2.2 Liblit et al.‘s “Importance” and “Increase” Measures

Liblit et al. [31] proposed an SFL metric called *Importance* that is based on the outcome of both existing program predicates and additional, inserted predicates. Although it is predicate-based rather than coverage based, it represents an early effort to address the issues we consider in this chapter. For a given predicate \( q \), *Importance* is the geometric mean of two terms:

\[
\text{Importance}(q) = \left( \frac{\log(f)}{\log(f_q)} \right)^2 \cdot \frac{1}{\text{Increase}(q)}
\]

where \( f \) is the total number of tests that fail, \( f_q \) is the number of failing tests in which \( q \) evaluated to \textit{true}, and where *Increase*(\( q \)) is an estimator for the difference of probabilities

\[
\Pr(F | q \text{ evaluates to } \text{true}) - \Pr(F | q \text{ is evaluated})
\]

The second term \( \Pr(F | q \text{ is evaluated}) \), denoted \textit{Context} by Liblit et al. [31], is a correction intended to ensure that \( q \) is scored “not by the chance that it implies failure, but by how much difference it makes that the predicate is observed to be true versus simply reaching the line where the predicate is checked” [31]. Gore and Reynolds [32] stated that the correction term in *Increase* “attempts to factor out predicates that are more susceptible to failure because of the program flow once the fault is triggered. While this heuristic can be effective it is not a proven solution.” The last statement refers to the fact that the correction in *Increase* is not based on causal inference theory (e.g., Pearl’s Backdoor Adjustment Theorem [22]). Gore and Reynolds called the bias that the correction is intended to remove “failure-flow confounding bias”.

Liblit et al.’s Increase metric can be viewed as a special, conditional case of the relative measure RP. Consider a program with a branch predicate $q$ that directly controls execution of a statement $s$. Given that $q$ is evaluated, Increase can be expressed as

$$\Pr(F \mid s \text{ covered}, q \text{ evaluated}) - \Pr(F \mid q \text{ evaluated})$$

Suppose that the branch from $q$ to $s$ is taken if and only if $q$ evaluates to true. Then the first term of the difference is equivalent to $\Pr(F \mid q \text{ evaluates to true})$, which is the first term of Increase. The second term of the difference, $\Pr(F \mid q \text{ is evaluated})$, is the second term of Increase.

Although Increase employs a heuristic correction for confounding bias and relative precision somewhat resembles Increase, RP does not correct for confounding bias, because its “adjustment” term $\Pr(F)$ does not vary for different program elements. Confounding bias in SFL depends on the program dependence relationships between statements [9][10].

### 3.2.3 Causal Effect and Associational Risk Difference

Landsberg et al. [7] evaluated 157 association measures for use in fault localization, which included two associational causal measures, $Suppes = \Pr(E \mid C) - \Pr(E \mid \lnot C)$ and $Eels = \Pr(E \mid C) - \Pr(E)$, originally measuring the degree that the execution of a program entity $C$ causes the error $E$. When we apply these two measures on programs, the entity $C$ represents the coverage status of program element $e$ and the error $E$ represents the outcome of tests. The measure $Suppes$ can be rewritten as $\Pr(F \mid e) - \Pr(F \mid \lnot e)$, which is associational risk difference, and the measure $Eels$ can be rewritten as $\Pr(F \mid e) - \Pr(F)$, which is relative precision. (Note that this is not the same as the causal risk difference.)
Section 2.3 presents the *potential outcome framework* and shows that $E[Y_t] = \Pr(Y_t = 1)$ when the outcomes are binary [16]. Thus, *average causal effect (ACE)* in (2-1) can be expressed as

$$ACE = \Pr(Y_1 = 1) - \Pr(Y_0 = 1)$$

*Exchangeability* means that the counterfactual risk under every treatment value $t$ is the same in the treated and in the untreated [16]. This means that the potential outcome and actual treatment are independent, denoted $Y_t \perp \perp T$. If the exchangeability holds, then $\Pr(Y_t = 1) = \Pr(Y = 1 \mid T = t)$. A randomized experiment is expected to produce exchangeability because it ensures that the missing values (counterfactual outcomes) occurred by chance. In ideal randomized experiments, association is causation [16], so $ACE$, which is a causal risk difference [9], can be estimated by using the associational risk difference (ARD),

$$ACE = \Pr(Y = 1 \mid T = 1) - \Pr(Y = 1 \mid T = 0)$$

which indicates the difference between the probability of death given that an individual is treated and the probability of death given an individual is untreated. In programs, a treatment represents the execution of a program element (1: covered, 0: uncovered), and an outcome represents the outcome of a test case (1: failed, 0: passed). The associational risk difference can be rewritten in terms of our notation:

$$\Pr(F \mid e) - \Pr(F \mid \neg e)$$
3.3 Analysis of CBSFL Metrics That Have Performed Well in Many-Metric Comparisons

In this subsection, the CBSFL metrics we analyze include: \textit{Jaccard} (Jac) and \textit{Sorensen-Dice} (SD) \cite{5}, \textit{symmetric Klosgen} (SK) \cite{11}\cite{1}, \textit{Ochiai} (Och) \cite{1}\cite{2}\cite{8}, and \textit{Pattern-Similarity2} (PS2) \cite{7}. In addition, we derive three “refined” metrics from existing ones, by incorporating terms from other metrics. Note that we analyze only metrics that are intended to be used with programs that may contain multiple faults. The analysis examines each metric’s composition, properties, and relationships to other metrics.

3.3.1 The F1-measure and Related Metrics

An empirical comparison of 42 SFL metrics by Hofer et al. \cite{5} indicated that the \textit{Jaccard} and \textit{Sorensen-Dice} metrics were among the three most effective metrics for fault localization in \textit{spreadsheet} applications (\textit{Ochiai} was the third). Rewriting their definitions of the metrics in terms of our notation, we have

\[
Jac = \frac{\Pr(F \cap e)}{\Pr(F) + \Pr(e) - \Pr(F \cap e)} \tag{3-5}
\]

and

\[
SD = \frac{2 \times \Pr(F \cap e)}{\Pr(F) + \Pr(e)} \tag{3-6}
\]

Baah et al. \cite{9} point out that the \textit{F1-measure} (F1) is the harmonic mean of \textit{recall} and \textit{precision}. We extend that equality with the following derivation:
This proves that $F1$ and Sorensen-Dice are score equivalent, because the third expression in derivation (3-7) is algebraically equivalent to the definition of Sorensen-Dice in (3-6). Hence, we may write $Jac \equiv R SD \equiv F1$. For simplicity, we shall consider $F1$ to be the representative for these three metrics. Observe that if $Pr(F) > Pr(e)$ then $(2 \times Pr(e))/(Pr(F) + Pr(e)) < 1$, hence $(2 \times Pr(e))/(Pr(F) + Pr(e))$ serves to downweight the precision component of $F1$ when coverage of $e$ cannot account for all failures. As with relative recall, such downweighting seems desirable for CBSFL. On the other hand, if $Pr(F) < Pr(e)$ then precision is upweighted.

We can derive a new metric by replacing the precision term $Pr(F \mid e)$ in the last expression of derivation (3-7) with $RP$. We call the new metric the relative $F1$-measure ($RF1$):

$$RF1 = \frac{2 \times Pr(e)}{Pr(F) + Pr(e)} \times (Pr(F \mid e) - Pr(F))$$

(3-8)
3.3.2 Metrics Related to the Ochiai Metric

In the empirical study by Lucia et al. [1], which evaluated the effectiveness of 40 association measures for use in fault localization, the Ochiai and symmetric Klosgen metrics performed the best overall. We shall examine the relationships among Ochiai [2], symmetric Klosgen [1], and asymmetric Klosgen [33].

Baah et al. [9] showed that the Ochiai metric, which is a symmetric, can be expressed in terms of recall and precision:

\[
Och \approx \sqrt{\Pr(e \mid F) \times \Pr(F \mid e)}
\]

We saw in Section 2.1 that if \(\Pr(F) > 0\) but \(\Pr(e \mid F) < 1\) then program element \(e\) is not the only cause of failures. Thus, in this case the recall component of \(Och\) serves to downweight the precision component by the maximum proportion of failures that could, based on coverage, be caused by \(e\). To clarify the relationship of \(Och\) to some other metrics, we rewrite it as follows:

\[
Och \approx \sqrt{\frac{\Pr(e \mid F) \times \Pr(F \mid e)}{\Pr(e)}}
\]

\[
= \sqrt{\frac{\Pr(F \cap e)}{\Pr(F)}} \times \Pr(F \mid e) \times \frac{\Pr(e)}{\Pr(F)}
\]

\[
= \sqrt{\Pr(F \mid e) \times \Pr(F \mid e)} \times \frac{\Pr(e)}{\Pr(F)}
\]

\[
= \frac{\Pr(e)}{\Pr(F)} \times \Pr(F \mid e)
\]

(3-10)

This characterization of \(Och\) embeds within the square root the same weighting factor \(\Pr(e)/\Pr(F)\) that is present in relative recall.
We define a new measure, called *relative Ochiai (RO)*, by replacing \( \Pr(F \mid e) \) with \( RP \) in the last line of derivation (3-10):

\[
RO = \sqrt{\frac{\Pr(e)}{\Pr(F)}} \times (\Pr(F \mid e) - \Pr(F))
\]

(3-11)

One might consider replacing both \( \Pr(F \mid e) \) and \( \Pr(e \mid F) \) in (3-9) with \( RP \) and *relative recall*, respectively. However, we have seen that both \( RP \) and \( RR \) may vary from \(-1\) to \(1\). If both \( RP \) and \( RR \) are negative for a program element \( e \), this suggests that \( e \) is unlikely to be faulty. Counterintuitively, the product of \( RP \) and \( RR \) is *positive* in this case, indicating that \( e \) is at least somewhat suspicious. (Note that since \( RP \) and \( RR \) are either both positive or both negative, their product is always positive.)

Note that a two-variable measure \( m(A, B) \) is *symmetric* if \( m(a, b) = m(b, a) \) for all values \( a, b \) [11]. The *asymmetric Klosgen (AK)* measure [33] is defined in terms of probabilities by

\[
AK = \sqrt{\Pr(e)} \times (\Pr(F \mid e) - \Pr(F))
\]

(3-12)

The second term on the right is *relative precision*, so we may also write \( AK = \sqrt{\Pr(e)} \times \) \( RP \). Comparison of (3-11) and (3-12) makes clear that \( RO \) and \( AK \) are rank equivalent, \( RO \equiv_R \) \( AK \), because \( \Pr(F) \) is a fixed value for a given program and test set. Thus, applying both \( RO \) and \( AK \) to the same program will produce the same ranking of program elements.

Lucia et al. [1] evaluated *symmetric Klosgen*, which was introduced by Tan et al. [11], as a fault localization metric. It is defined in terms of probabilities by
\[ SK = \sqrt{\Pr(F \cap e)} \times \max(\Pr(F | e) - \Pr(F), \Pr(e | F) - \Pr(e)) \]

\[ = \sqrt{\Pr(F \cap e)} \times \max(RP, RR) \]

(3-13)

Note that the factor \(\max(RP, RR)\) in \(SK\) is equal to the Added Value metric evaluated by Lucia et al. [1], so we do not consider Added Value further. For a given program and test set, the factor \(\sqrt{\Pr(F \cap e)}\) is greatest for program elements \(e\) whose coverage coincides most with the occurrence of failures (whether or not the elements are faulty). Thus, \(\sqrt{\Pr(F \cap e)}\) plays a role in \(SK\) similar to those of the weighting factors we have identified in relative recall, the \(F_1\)-measure, and Ochiai.

### 3.3.3 Optimized Pattern-Similarity (Pattern-Similarity2)

Landsberg et al. [7] evaluated 157 association measures for use in fault localization, including 95 that had not previously been used for that purpose. They also introduced two metrics \(Lex_{Ochiai}\) and Pattern-Similarity2, and these two metrics performed best overall in their evaluation. (\(Lex_{Ochiai}\) differs from \(Ochiai\) only when \(f_e \neq f\), in which case its value is \(p - p_e + 2\). We do not consider it further here.) Pattern-Similarity2 is based on the Pattern-Similarity (PS) metric [34], which we now define and then re-express in terms of probabilities:

\[ PS = -\frac{4 \times (f - f_e) \times p_e}{(f_e + p_e + (f - f_e) + (p - p_e))^2} \]

\[ = -\frac{4 \times (f - f_e) \times p_e}{n^2} \]

\[ \approx -4 \times (\Pr(F) - \Pr(F \cap e)) \times \Pr(P \cap e) \]

\[ = -4 \times (\Pr(F) - \Pr(F \cap e)) \times (\Pr(e) - \Pr(F \cap e)) \]
\[\begin{align*}
&= -4 \times \Pr(F) \times (1 - \Pr(e | F)) \times \Pr(e) \times (1 - \Pr(F | e)) \\
&= -4 \times \Pr(F) \times \Pr(\neg e | F) \times \Pr(e) \times \Pr(P | e)
\end{align*}\]

(3-14)

(To avoid division by zero, Landsberg et al. add a small constant, 0.5, to \(f_e, p_e, f - f_e,\) and \(p - p_e\) when calculating the value of \(PS\), which we have disregarded here.) It is evident that \(PS\) is based on recall and precision, which are in the third and fifth factors, respectively, in the next-to-last expression of derivation (3-14). However, in \(PS\) both measures are subtracted from 1, unlike their occurrences in other CBSFL metrics we examine. Consider the last two lines of derivation (3-14). We saw in Section 2.1 that the quantity \(1 - \Pr(e | F) = \Pr(e \text{ not covered} | F) = \Pr(\neg e | F)\) is a lower bound on the probability that \(e\) is not the cause of a realized failure. The probability \(1 - \Pr(F | e) = \Pr(P | e)\) is a well-known quantity that is called the false discovery rate (FDR) [35], assuming that in running a test that covers program element \(e\) we seek to discover failures caused by \(e\). Intuitively, if \(e\) is a major cause of failures, both of these probabilities should be small, in which case \(PS\), which is never positive, will be relatively large.

Landsberg et al. derive the PS2 metric from \(PS\) by adding 0.1 to \(f - f_e\) and adding 0.5 to \(p_e\). We now define PS2 in terms of counts, as Landsberg et al. do, and then re-express it in terms of probabilities:

\[
\begin{align*}
PS2 &= \frac{-4 \times (f - f_e + 0.1) \times (p_e + 0.5)}{(f_e + (p_e + 0.5) + (f - f_e + 0.1) + (p - p_e))^2} \\
&= \frac{-4 \times (f - f_e) \times p_e - 2 \times (f - f_e) - 0.4 \times p_e - 0.2}{(n + 0.6)^2} \\
&\approx -4 \times (\Pr(F) - \Pr(F \cap e)) \times \Pr(P \cap e) \\
&\quad - \frac{2}{n} \times (\Pr(F) - \Pr(F \cap e)) - \frac{0.4}{n} \times \Pr(P \cap e)
\end{align*}
\]
= -4 \times \Pr(F) \times (1 - \Pr(e \mid F)) \times \Pr(e) \times (1 - \Pr(F \mid e))

- \frac{2}{n} \times \Pr(F) \times (1 - \Pr(e \mid F)) - \frac{0.4}{n} \times \Pr(e) \times (1 - \Pr(F \mid e))

= -4 \times \Pr(F) \times \Pr(\neg e \mid F) \times \Pr(e) \times \Pr(P \mid e)

- \frac{2}{n} \times \Pr(F) \times \Pr(\neg e \mid F) - \frac{0.4}{n} \times \Pr(e) \times \Pr(P \mid e)

= PS - \frac{2}{n} \times \Pr(F) \times \Pr(\neg e \mid F) - \frac{0.4}{n} \times \Pr(e) \times \Pr(P \mid e)

(3-15)

In derivation (3-15), we have assumed $n$ is much larger than the small constants added to different quantities. Hence, in the second expression the denominator $(n + 0.6)^2$ is approximated by $n^2$.

Thus, $PS2$ subtracts two new terms, involving $\Pr(\neg e \mid F)$ and $\Pr(P \mid e)$ respectively, from $PS$. As with $PS$, if $e$ is a major cause of failures, we should expect both of these probabilities to be small, in which case $PS2$, which is never positive, will be relatively large. The magnitudes of the two terms subtracted from $PS$ diminish as the number of tests $n$ increases. Nevertheless, they evidently make a difference. Landsberg et al. [7] reported empirical results indicating that $PS2$ outperforms $PS$.

### 3.3.4 Summarizing the Analysis

We now summarize our algebraic and probabilistic analysis of CBSFL metrics. Most of the preexisting CBSFL metrics we have considered contain *precision* as a term, including *relative recall*, the *F1-measure* (which again is score equivalent to Sorensen-Dice and rank equivalent to Jaccard), *Ochiai*, *asymmetric Klosgen* (which is rank equivalent to relative Ochiai), *symmetric Klosgen*, *Pattern-Similarity*, and *Pattern-Similarity*2. Three metrics
(RR, AK, and SK) contain a relative precision term. Two metrics (F1 and Och) are of the form \( w \times \text{precision} \), two others (RR and AK) are of the form \( w \times RP \), and one (SK) is of the form \( w \times \max(RP, RR) \), where in each case \( w \) is a term that downweights precision or RP for a program element \( e \) when it cannot be the cause of all observed failures. The latter metrics include two of the three metrics that performed best in recent comparisons of many CBSFL metrics [1], namely Och and SK. The last of the three metrics is PS2, which has a different structure.

We have defined two new metrics, relative F1 and relative Ochiai by replacing the precision term in F1 and Ochiai by RP. We can similarly modify the well-known Tarantula metric of Jones et al. [3]. Baah et al. [9] showed that when \( \Pr(P) \approx \Pr(F) \), Tarantula approximates precision. Moreover, Tarantula and precision are rank equivalent [36][6]. To improve the effectiveness of Tarantula, we multiply precision by RR to form a new metric, called enhanced Tarantula (ET):

\[
ET = RR \times \Pr(F \mid e) \\
= \frac{\Pr(e)}{\Pr(F)} \times (\Pr(F \mid e) - \Pr(F)) \times \frac{\Pr(F \cap e)}{\Pr(e)} \\
= \frac{\Pr(F \cap e)}{\Pr(F)} \times (\Pr(F \mid e) - \Pr(F)) \\
= \Pr(e \mid F) \times (\Pr(F \mid e) - \Pr(F)) \\
= \Pr(e \mid F) \times RP
\]

(3-16)

Thus, \( ET \) is actually a product of recall and RP, so it shares the form \( w \times RP \) with RR and AK, where \( w \) downweights RP when coverage of \( s \) cannot account for all failures.
3.4 New Metrics Derived from Associational Risk Difference

In this subsection, we first analyze the comparison of associational risk difference and relative precision. Then, we define another four new metrics with associational risk difference, and these new metrics are in the form $w \times ARD$. The first three new metrics are defined based on the original metrics Ochiai, F1-measure and recall, respectively. The fourth new metric is derived from enhanced Tarantula.

3.4.1 Associational Risk Difference and Relative Precision

In Section 3.2.3, we have seen that two associational causal measures Suppes and Eels, which are also called associational risk difference (ARD) and relative precision (RP), respectively, are used for fault localization by Landsberg et al. [7]. To clarify the relationship of ARD to RP, we rewrite ARD as follows:

$$ARD = \Pr(F \mid e) - \Pr(F \mid \neg e)$$

$$= \frac{\Pr(F \cap e)}{\Pr(e)} - \frac{\Pr(F \cap \neg e)}{\Pr(\neg e)}$$

$$= \frac{\Pr(F \cap e)}{\Pr(e)} \times \frac{\Pr(\neg e)}{\Pr(\neg e)} - \frac{\Pr(F) - \Pr(F \cap e)}{\Pr(\neg e)} \times \frac{\Pr(e)}{\Pr(e)}$$

$$= \frac{\Pr(F \cap e) \times \Pr(\neg e) - \Pr(F) \times \Pr(e) + \Pr(F \cap e) \times \Pr(e)}{\Pr(e) \times \Pr(\neg e)}$$

$$= \frac{1}{\Pr(\neg e)} \times \frac{\Pr(F \cap e) \times \Pr(e) - \Pr(F) \times \Pr(e)}{\Pr(e)}$$

$$= \frac{1}{\Pr(\neg e)} \times (\Pr(F \mid e) - \Pr(F))$$

(3-17)
The second term in the last expression in derivation (3-17) is relative precision, so we may also write $ADR = \frac{1}{\Pr(\neg e)} \times RP$.

We have seen that precision is rank equivalent to $RP$ because $\Pr(F)$ in $RP$ is fixed for a given program and set of tests. Consider two program elements having the same precision values, where one program element is faulty and the other is nonfaulty. $ARD$, which is $\Pr(F \mid e) - \Pr(F \mid \neg e)$, provides more information than precision because it includes the additional term $-\Pr(F \mid \neg e)$. For example, Figure 3-1 shows an example program with three test cases, where “1” in a cell represents the statement is covered and “0” otherwise. The bottommost row shows the outcomes of tests (F: FAIL, P: PASS). The fault is at statement S4 and the correct statement should be “b=b+5”. Statements S2 and S4 have the same precision values, but have different $ARD$ values 0.50 and 1.00, respectively. Therefore, the term $-\Pr(F \mid \neg e)$ in $ARD$ lowers the score of S2.

<table>
<thead>
<tr>
<th>Example</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>prec</th>
<th>ARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Outcome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-1. An example program for comparing precision to $ARD$
3.4.2 New Metrics Involving Associational Risk Difference

We have seen three metrics \( \text{Ochiai} \), \( F1\text{-measure} \) and \( \text{recall} \), which are of the form \( w \times \text{precision} \). Based on \( \text{Ochiai} \), we also define a new measure, called \( \text{risk Ochiai} (RDO) \), by replacing \( \Pr(F \mid e) \) with \( ARD \) in the last line of derivation (3-10):

\[
RDO = \frac{\Pr(e)}{\Pr(F)} \times (\Pr(F \mid e) - \Pr(F \mid \neg e))
\]

(3-18)

Based on the \( F1\text{-measure} \), we define a new metric by replacing \( \Pr(F \mid e) \) in the last expression of derivation (3-7) with \( ARD \). We call this new metric \( \text{risk F1-measure} (RDF1) \):

\[
RDF1 = \frac{2 \times \Pr(e)}{\Pr(F) + \Pr(e)} \times (\Pr(F \mid e) - \Pr(F \mid \neg e))
\]

(3-19)

Based on \( \text{recall} \), which can be written as \( \frac{\Pr(e)}{\Pr(F)} \times \Pr(F \mid e) \), we also define a new measure, called \( \text{risk recall} (RDRec) \), by replacing \( \Pr(F \mid e) \) with \( ARD \):

\[
RDRec = \frac{\Pr(e)}{\Pr(F)} \times (\Pr(F \mid e) - \Pr(F \mid \neg e))
\]

(3-20)

We take \( \Pr(e \mid F) \) in \( \text{enhanced Tarantula} \) as a weight to define the last new metric and call this new metric the \( \text{enhanced Tarantula2} (ET2) \):

\[
ET2 = \Pr(e \mid F) \times (\Pr(F \mid e) - \Pr(F \mid \neg e))
\]

(3-21)

The first term in probability on the right is \( \text{recall} \), and the second term in the parentheses on the right is \( ARD \).
3.5 Empirical Study

We have seen that among the CBSFL metrics found to be most effective in recent empirical comparisons [1][5][7], primarily Ochiai, symmetric Klosgen, Pattern-Similarity2, and the F1-measure (recalling that $\text{Jac} \equiv R \text{SD} \equiv F1$), all but the PS2 metric follow one of three forms: $w \times \text{precision}$, $w \times \text{RP}$, or $w \times \text{max(RP, RR)}$, where $w$ is a term that downweights precision or RP for a program element $e$ when it cannot be the cause of all observed failures. This raises the question of how important the weights $w$ are to the effectiveness of the metrics having one of these three weighted forms, including those just listed as well as the three metrics we defined by “refining” existing metrics, namely relative $F1$, relative Ochiai (recalling that $RO \equiv R \text{AK}$), and enhanced Tarantula. To address this concern, we will consider three research questions:

- RQ1: Are the weights $w$ helpful in the metrics of the form $w \times \text{RP}$ or of the form $w \times \text{max(RP, RR)}$?
- RQ2: Are the weights $w$ helpful in the metrics of the form $w \times \text{precision}$?
- RQ3: Is the effectiveness of the metrics in the form $w \times \text{precision}$ enhanced by using the relative precision term in place of precision? Why or why not? Observe that the metrics of the form $w \times \text{RP} = w \times \text{precision} - w \times \text{Pr}(F)$ have additional terms $-w \times \text{Pr}(F)$ that are not present in their corresponding basic metrics, which are in the form $w \times \text{precision}$. Hence, this question also considers how important the additional terms $-w \times \text{Pr}(F)$ are in the “refined” metrics of the form $w \times \text{RP}$.
Baah et al. [9] showed the better performance of the average failure-causing effect measure (AFCE) over precision. Section 3.2.3 shows the relation of AFCE and the associational risk difference. This raises another research question:

- **RQ4:** Does associational risk difference outperform precision?

Section 3.4.2 introduces four refined metrics risk Ochiai, risk F1-measure, risk recall and enhanced Tarantula2, which are in the form \( w \times \text{ARD} \). The former three metrics, risk Ochiai, risk F1-measure, and risk recall are derived from Ochiai, F1-measure, and recall, respectively, which each have the form \( w \times \text{precision} \), by replacing the precision terms with the ARD terms. The latter metric, enhanced Tarantula2, is derived from enhanced Tarantula, which has the form \( w \times \text{RP} \), by replacing the RP term with the ARD term. This raises the following two research questions:

- **RQ5:** Are the weights \( w \) helpful in the metrics of the form \( w \times \text{ARD} \)?
- **RQ6:** Is the effectiveness of the metrics of the form \( w \times \text{precision} \) enhanced by using the associational risk difference term in place of the precision term? This question also considers how important the additional terms \(-w \times \Pr(\neg F | e)\) are in the “refined” metrics in the form \( \times \text{ARD} = w \times \Pr(F | e) - w \times \Pr(F | \neg e) \).

We use the results of empirical study to answer these six research questions. We also discuss and summarize all metrics involving the relative precision and associational risk difference terms. First, we describe the setup of our study.

### 3.5.1 Study Setup

The empirical study applied the metrics with data (provided by Steinmann et al. [4]) from over 5,000 versions of single-fault programs and around 49,000 versions of multiple-
fault programs created from 10 original subject programs and more than 5,000 injectable faults. These subject programs and faults are from Steimann et al. [4] and are publicly accessible. The test coverage information for the subject programs are recorded on the method level. Table 3-1 shows the details of these subject programs, including the program version, the number of methods, the number of methods under test, the number of test cases, the number of faults, and the number of multiple-fault versions. Here, the number of faults also indicates the number of single-fault versions. In each subject program, faults were injected with six mutation operators which involve negating conditions, replacing constants, deleting statements, “inversing” operators, assigning null values, and returning null values. One hundred faults were randomly generated with each mutation operator. Therefore, up to 600 faults were seeded (for some subject programs, a mutation operator generated less than 100 faults).

As shown in Table 3-1, each subject program has over 350 injectable faults. To create multiple-fault programs, 1,000 faulty versions were generated by randomly injecting \( l \) faults for each \( l \in \{2, 4, 8, 16, 32\} \), such that the faults were located in different methods. So, there were around 5,000 multiple-fault versions for each subject program. In our study, we discarded some versions for which the measure \( Cost \), which is defined below, could not be estimated.
Table 3-1. Details of Subject Programs

<table>
<thead>
<tr>
<th>Subject programs</th>
<th># Methods</th>
<th># Methods under tests</th>
<th># Test cases</th>
<th># Faults</th>
<th># Multiple-fault ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daikon 4.6.4</td>
<td>14387</td>
<td>1936</td>
<td>157</td>
<td>352</td>
<td>4992</td>
</tr>
<tr>
<td>Eventbus 1.4</td>
<td>859</td>
<td>338</td>
<td>91</td>
<td>577</td>
<td>5000</td>
</tr>
<tr>
<td>Jaxen 1.1.5</td>
<td>1689</td>
<td>961</td>
<td>695</td>
<td>600</td>
<td>5000</td>
</tr>
<tr>
<td>Jster 1.37b</td>
<td>378</td>
<td>152</td>
<td>64</td>
<td>411</td>
<td>5000</td>
</tr>
<tr>
<td>JExel 1.0.0b13</td>
<td>242</td>
<td>150</td>
<td>335</td>
<td>537</td>
<td>5000</td>
</tr>
<tr>
<td>Jpasec 2.0</td>
<td>1011</td>
<td>893</td>
<td>510</td>
<td>598</td>
<td>5000</td>
</tr>
<tr>
<td>AC codec 1.3</td>
<td>265</td>
<td>229</td>
<td>188</td>
<td>543</td>
<td>5000</td>
</tr>
<tr>
<td>AC Lang 3.0</td>
<td>5373</td>
<td>2075</td>
<td>1666</td>
<td>599</td>
<td>5000</td>
</tr>
<tr>
<td>EclipseDraw2d 3.4.2</td>
<td>3231</td>
<td>875</td>
<td>89</td>
<td>570</td>
<td>5000</td>
</tr>
<tr>
<td>HTML Parser 1.6</td>
<td>1925</td>
<td>785</td>
<td>600</td>
<td>599</td>
<td>5000</td>
</tr>
<tr>
<td>Total</td>
<td>29360</td>
<td>8397</td>
<td>4395</td>
<td>5386</td>
<td>49992</td>
</tr>
</tbody>
</table>

The evaluation was conducted using the statistical package R, v3.0.2, on a workstation with dual Intel Xeon 3.7 CPUs.

The measure Cost was used to evaluate the effectiveness of CBSFL metrics. It is the percentage of all methods that a developer needs to examine, in nonincreasing order of suspiciousness, to find the first faulty method. To deal with the issue that multiple methods may receive the same suspiciousness score, called ties [37], we used ideas of Xuan and Monperrus [38] and Steinmann et al. [4] to obtain our cost measure. Given a set M of methods, Cost is defined by

\[
Cost(m) = | \{ m | sc(m) > sc(m^*) \} | + \frac{|\{m | sc(m) = sc(m^*)\}| + 1}{k + 1}
\]

(3-22)

where m ∈ M is any method, m* is the first faulty method in nonincreasing order of suspiciousness scores, sc(m) is the suspiciousness score of m, |...| denotes the size of a set, and k is the number of faults in critical ties [37]. We assumed that if developers examined a faulty method, the fault would be found.
We average the Cost values for a chosen metric over all versions of each subject program. Following Bandyopadhyay and Ghosh [39], we then compute, for each subject program, the percentage reduction in average Cost achieved by the chosen metric over the baseline metric. Suppose that for a program version, the Cost for the baseline metric is \( C \) and the Cost for the chosen metric used is \( D \). The percentage reduction is given by the expression \( \frac{(C - D)}{C} \times 100\% \). This value will be negative if the baseline metric performs better than the chosen metric.

### 3.5.2 Results for RQ1: Metrics \( w \times RP \) or \( w \times \max(RP, RR) \) over RP

To evaluate the effect of weight term \( w \) in the metrics of the form \( w \times RP \) or \( w \times \max(RP, RR) \), namely \( SK, RO, RF1, RR \), and \( ET \), we first computed the percentage reduction in average Cost that each of them achieve over the baseline metric \( RP \), for both single-fault programs and multiple-fault programs. (Note that since \( AK \) is rank equivalent to \( RO \), Cost is the same for both metrics.) Table 3-2 shows the percentage reductions in average Cost for single-fault programs. Per-program and per-metric averages are shown in the rightmost column and the bottommost row, respectively.
Table 3-2. Percentage Reductions for Single-fault Programs \( (w \times RP \text{ or } w \times \max(RP, RR) \text{ vs. } RP) \)

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>SK</th>
<th>RO</th>
<th>RF1</th>
<th>RR</th>
<th>ET</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>12.62%</td>
<td>18.42%</td>
<td>18.25%</td>
<td>18.80%</td>
<td>12.22%</td>
<td>16.06%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>62.45%</td>
<td>57.14%</td>
<td>58.87%</td>
<td>65.66%</td>
<td>59.36%</td>
<td>60.69%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>61.92%</td>
<td>45.25%</td>
<td>41.55%</td>
<td>61.17%</td>
<td>49.33%</td>
<td>51.85%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>43.12%</td>
<td>35.51%</td>
<td>35.10%</td>
<td>43.49%</td>
<td>36.47%</td>
<td>38.74%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>53.66%</td>
<td>40.68%</td>
<td>37.59%</td>
<td>57.12%</td>
<td>38.30%</td>
<td>45.47%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>75.90%</td>
<td>71.07%</td>
<td>67.16%</td>
<td>70.21%</td>
<td>71.40%</td>
<td>71.15%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>48.43%</td>
<td>41.55%</td>
<td>38.04%</td>
<td>48.46%</td>
<td>42.05%</td>
<td>43.71%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>38.37%</td>
<td>51.26%</td>
<td>49.69%</td>
<td>52.88%</td>
<td>35.72%</td>
<td>45.59%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>32.62%</td>
<td>28.34%</td>
<td>26.62%</td>
<td>33.14%</td>
<td>28.43%</td>
<td>29.83%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>72.71%</td>
<td>61.81%</td>
<td>58.45%</td>
<td>72.13%</td>
<td>64.56%</td>
<td>65.93%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>50.18%</td>
<td>45.10%</td>
<td>43.13%</td>
<td>52.31%</td>
<td>43.78%</td>
<td>46.90%</td>
</tr>
</tbody>
</table>

Figure 3-2 graphically depicts the percentage reductions in average \( \text{Cost} \) for single-fault programs. All five weighted metrics were less costly than \( RP \) over the versions of each of the ten subject programs. Thus, the weight terms \( w \) were beneficial for the single-fault programs. The points on the solid line represent the average percentage reduction for each subject program over all five metrics. \( RR \) achieved the largest average reduction (52.31%) over all subject programs. The average reduction achieved by \( SK \) (50.18%) was competitive with that of \( RR \). \( RF1 \) had the lowest average reduction (43.13%). Also, the between-program variance (0.028) was greater than the between-metric variation (0.002).
Figure 3-2. Percentage reductions for single-fault programs \((w \times RP)\) or \(w \times \max(RP, RR)\) vs. \(RP\).

Table 3-3 shows the percentage reductions in average cost for the multiple-fault programs and it has the same fields as Table 3-2 has.

Table 3-3. Percentage Reductions for Multiple-fault Programs \((w \times RP)\) or \(w \times \max(RP, RR)\) vs. \(RP\).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>SK</th>
<th>RO</th>
<th>RF1</th>
<th>RR</th>
<th>ET</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>43.90%</td>
<td>44.08%</td>
<td>43.69%</td>
<td>39.24%</td>
<td>43.74%</td>
<td>42.93%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>66.30%</td>
<td>65.00%</td>
<td>65.04%</td>
<td>64.24%</td>
<td>64.81%</td>
<td>65.08%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>62.24%</td>
<td>59.43%</td>
<td>57.80%</td>
<td>56.93%</td>
<td>58.94%</td>
<td>59.07%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>40.39%</td>
<td>40.82%</td>
<td>42.30%</td>
<td>44.72%</td>
<td>45.87%</td>
<td>42.82%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>47.97%</td>
<td>42.19%</td>
<td>41.51%</td>
<td>42.17%</td>
<td>43.67%</td>
<td>43.50%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>72.91%</td>
<td>72.51%</td>
<td>67.25%</td>
<td>49.81%</td>
<td>69.39%</td>
<td>66.37%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>64.08%</td>
<td>64.81%</td>
<td>65.26%</td>
<td>66.14%</td>
<td>65.66%</td>
<td>65.19%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>53.31%</td>
<td>56.28%</td>
<td>55.96%</td>
<td>46.07%</td>
<td>56.62%</td>
<td>53.65%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>32.29%</td>
<td>33.14%</td>
<td>31.16%</td>
<td>24.76%</td>
<td>31.41%</td>
<td>30.55%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>18.92%</td>
<td>19.77%</td>
<td>17.03%</td>
<td>2.16%</td>
<td>13.94%</td>
<td>14.36%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>50.23%</td>
<td>49.80%</td>
<td>48.70%</td>
<td>43.63%</td>
<td>49.40%</td>
<td>48.35%</td>
</tr>
</tbody>
</table>
Figure 3-3 graphically depicts the percentage reductions in average Cost for multiple-fault programs. All five weighted metrics were less costly than RP over the versions of each of the ten subject program. Thus, the weight terms $w$ were beneficial for the multiple-fault programs also, which is not surprising since they penalize $RP$ when there is evidence of multiple faults. The points on the solid line represent the average percentage reduction for each subject program over all five metrics. $SK$ achieved the largest average reduction (50.23%) over all subject programs. $RO$, $ET$, and $RF1$ were also competitive, with average reductions of 49.80%, 49.40% and 48.70%, respectively. $RR$ had the lowest average reduction (43.63%).

The metrics $SK$, $RO$, $RF1$, $RR$ and $ET$ exhibited average positive reduction over the baseline metric $RP$. It means that the weights $\sqrt{Pr(F \cap e)}$, $\sqrt{Pr(e)/Pr(F)}$, $2 \times$
Pr(e)/(Pr(F) + Pr(e)), Pr(e)/Pr(F) and Pr(e | F), enhance SK, RO, RF1, RR and ET, respectively, on both single-fault and multiple-fault programs.

**Results for RQ1:** The results of empirical study show that the weights $w$ in the metrics of the form $w \times RP$ or $w \times \max(RP, RR)$ apparently help with both single-fault and multiple-fault programs.

### 3.5.3 Results for RQ2: Metrics $w \times \text{precision}$ over Precision

Before looking at RQ3, which asks whether the additional terms $-w \times Pr(F)$ enhance the metrics of the form $w \times RP = w \times \text{precision} - w \times Pr(F)$, namely RO, RF1, RR and ET, we first observe the effect of the weights $w$ in their corresponding basic metrics of the form $w \times \text{precision}$, which are $Och$, $F1$, recall, and $\Pr(e | F) \times \text{precision}$.

Table 3-4 shows the percentage reductions in average Cost achieved by each of the basic metrics over precision for single-fault programs. RO, RF1 and ET exhibited positive reductions over precision for all programs. Their average percentage reductions over all programs were 47.01%, 43.01% and 47.04%, respectively. The metric recall achieved negative reductions over precision for some programs, whose average was -4.06%.
Table 3-4. Percentage Reductions for Single-fault Programs ($w \times \text{prec}$ vs. $\text{prec}$).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>Och/prec</th>
<th>F1/prec</th>
<th>Recall/prec</th>
<th>recall*prec/prec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>12.23%</td>
<td>12.13%</td>
<td>-9.86%</td>
<td>12.23%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>65.14%</td>
<td>63.98%</td>
<td>36.29%</td>
<td>65.19%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>63.72%</td>
<td>48.04%</td>
<td>-6.32%</td>
<td>63.72%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>38.89%</td>
<td>36.27%</td>
<td>-16.55%</td>
<td>38.95%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>41.78%</td>
<td>36.52%</td>
<td>18.44%</td>
<td>41.78%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>71.21%</td>
<td>66.87%</td>
<td>19.22%</td>
<td>71.22%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>42.40%</td>
<td>39.20%</td>
<td>28.65%</td>
<td>42.40%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>35.83%</td>
<td>34.52%</td>
<td>9.05%</td>
<td>35.96%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>29.10%</td>
<td>28.10%</td>
<td>-31.93%</td>
<td>29.11%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>-11.32%</td>
<td>-11.90%</td>
<td>-192.52%</td>
<td>-11.32%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>47.01%</td>
<td>43.01%</td>
<td>-4.06%</td>
<td>47.04%</td>
</tr>
</tbody>
</table>

Table 3-5 shows the percentage reductions in average Cost achieved by each of the basic metrics over precision for multiple-fault programs. $Och$, $F1$ and $Pr(e \mid F) \times precision$ exhibited positive reductions over precision for most programs. Their average percentage reductions over all programs were 43.62%, 41.47% and 43.62%, respectively. The metric recall achieved negative reductions over precision for some programs, whose average was -18.59%.

Table 3-5. Percentage Reductions for Multiple-fault Programs ($w \times \text{prec}$ vs. $\text{prec}$).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>Och/prec</th>
<th>F1/prec</th>
<th>Recall/prec</th>
<th>recall*prec/prec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>43.36%</td>
<td>42.22%</td>
<td>18.73%</td>
<td>43.36%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>61.70%</td>
<td>60.26%</td>
<td>44.95%</td>
<td>61.71%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>56.53%</td>
<td>54.26%</td>
<td>-4.34%</td>
<td>56.53%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>48.55%</td>
<td>46.43%</td>
<td>20.32%</td>
<td>48.56%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>30.70%</td>
<td>29.18%</td>
<td>-55.96%</td>
<td>30.69%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>58.84%</td>
<td>50.90%</td>
<td>-57.47%</td>
<td>58.84%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>66.48%</td>
<td>65.47%</td>
<td>62.36%</td>
<td>66.47%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>56.55%</td>
<td>54.98%</td>
<td>-7.93%</td>
<td>56.55%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>24.76%</td>
<td>22.89%</td>
<td>-14.05%</td>
<td>24.77%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>-11.32%</td>
<td>-11.90%</td>
<td>-192.52%</td>
<td>-11.32%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>43.62%</td>
<td>41.47%</td>
<td>-18.59%</td>
<td>43.62%</td>
</tr>
</tbody>
</table>
Table 3-4 and Table 3-5 show that Och, F1 and Pr(e | F) \times precision perform very similarly and that recall is the least effective of the metrics we compared over single-fault and multiple-fault programs.

**Results for RQ2:** The results of the empirical study indicate that not all the weights \( w \) in the metrics of the form \( w \times precision \), namely Och, F1, recall, and Pr(e | F) \times precision, are helpful. The weights \( \sqrt{Pr(e)/Pr(F)} \), \( (2 \times Pr(e))/(Pr(F) + Pr(e)) \) and \( Pr(e | F) \) enhance the metrics Och, F1 and Pr(e | F) \times precision, respectively, for both single-fault and multiple-fault programs, but the weight \( Pr(e)/Pr(F) \) does not enhance recall for both single-fault and multiple-fault programs.

### 3.5.4 Results for RQ3: Metrics \( w \times RP \) over Metrics \( w \times precision \)

We next empirically compared the percentage reductions in average Cost achieved by each of the metrics of the form \( w \times RP \) (relative Ochiai, relative F1, relative recall, and enhanced Tarantula) over the corresponding metrics of the form \( w \times precision \) (Ochiai, F1, recall, and Pr(e | F) \times precision, respectively). (Recall that asymmetric Klosgen is rank equivalent to relative Ochiai.) Table 3-6 shows the percentage reductions for single-fault programs.
Table 3-6. Percentage Reductions for Single-fault Programs ($w \times RP$ vs. $w \times prec$).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RO/Och</th>
<th>RF1/F1</th>
<th>RR/recall</th>
<th>ET/wprec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>7.05%</td>
<td>6.97%</td>
<td>26.09%</td>
<td>-0.02%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>-22.94%</td>
<td>-14.20%</td>
<td>46.10%</td>
<td>-16.75%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>-50.90%</td>
<td>-12.48%</td>
<td>63.48%</td>
<td>-39.66%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>-5.53%</td>
<td>-1.83%</td>
<td>51.51%</td>
<td>-4.07%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>-1.89%</td>
<td>1.69%</td>
<td>47.42%</td>
<td>-5.98%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>-0.47%</td>
<td>0.88%</td>
<td>63.12%</td>
<td>0.62%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>-1.49%</td>
<td>-1.92%</td>
<td>27.77%</td>
<td>-0.60%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>24.05%</td>
<td>23.17%</td>
<td>48.20%</td>
<td>-0.36%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>-1.08%</td>
<td>-2.06%</td>
<td>49.32%</td>
<td>-0.96%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>-26.39%</td>
<td>-17.05%</td>
<td>85.14%</td>
<td>-17.30%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>-7.96%</td>
<td>-1.68%</td>
<td>50.81%</td>
<td>-8.51%</td>
</tr>
</tbody>
</table>

Figure 3-4 graphically depicts the same percentage reductions in average Cost as Table 3-6. RO, RF1, and ET exhibited negative reductions (increases) over Och, F1, and $Pr(e | F) \times precision$, respectively, for some programs. Their average percentage reductions over all programs were -7.96%, -1.68%, and -8.51%, respectively. RR achieved positive reductions over recall for all ten programs, whose average was 50.81%.

![Single-Fault Programs - w×RP vs. w×Prec (Baseline)](image_url)

Figure 3-4. Percentage reductions for single-fault programs ($w \times RP$ vs. $w \times prec$).
Table 3-7 shows the percentage reductions in average Cost achieved by each of the metrics of the form \( w \times RP \) over the corresponding metrics for multiple-fault programs.

Table 3-7. Percentage Reductions for Multiple-fault Programs (\( w \times RP \) vs. \( w \times prec \)).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RO/Och</th>
<th>RF1/F1</th>
<th>RR/recall</th>
<th>ET/wprec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>1.26%</td>
<td>2.55%</td>
<td>25.24%</td>
<td>0.67%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>8.63%</td>
<td>12.03%</td>
<td>35.05%</td>
<td>8.11%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>6.66%</td>
<td>7.75%</td>
<td>58.72%</td>
<td>5.53%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>-15.04%</td>
<td>-7.72%</td>
<td>30.62%</td>
<td>-5.22%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>16.59%</td>
<td>17.42%</td>
<td>62.92%</td>
<td>18.72%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>33.22%</td>
<td>33.30%</td>
<td>68.13%</td>
<td>25.62%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>-4.98%</td>
<td>-0.59%</td>
<td>10.05%</td>
<td>-2.42%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>-0.62%</td>
<td>2.18%</td>
<td>50.03%</td>
<td>0.14%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>11.14%</td>
<td>10.73%</td>
<td>34.03%</td>
<td>8.83%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>27.93%</td>
<td>25.85%</td>
<td>66.55%</td>
<td>22.69%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>8.48%</td>
<td>10.35%</td>
<td>44.13%</td>
<td>8.27%</td>
</tr>
</tbody>
</table>

Figure 3-5 graphically depicts the same percentage reductions in average Cost as Table 3-7. With the multiple-fault programs, \( RO, RF1, \) and \( ET \) achieved positive reductions over \( Och, F1, \) and \( Pr(e \mid F) \times precision \), respectively, which average 8.48%, 10.35%, and 8.27%, respectively, over all programs. \( RR \) achieved positive reductions over \( recall \) for all ten programs, which average 44.13%.
Comparing Figure 3-4 to Figure 3-5, we can see that the percentage reductions in average cost for $RO$, $RF_1$, and $ET$ shift mainly in the positive direction. Observe that the metrics $RO$, $RF_1$, $RR$, and $ET$ have the form $w \times RP = w \times \text{precision} - w \times \Pr(F)$. Hence they have a term $-w \times \Pr(F)$ that is not present in their corresponding baseline metrics $Och$, $F_1$, recall, and $Pr(e \mid F) \times \text{precision}$, which have the form $w \times \text{precision}$. $\Pr(F)$ is fixed for a given program and set of tests, so this additional term is mainly controlled by the weight $w$. We know that $w$ is relatively small for program elements whose coverage cannot explain most failures, so $w$ will “adjust” or “penalize” the additional term for such elements. The results of our study suggest that metrics with the additional term $-w \times \Pr(F)$ perform better than metrics without this term.

Figure 3-6 shows the relations between the percentage reductions in average Cost achieved by $RO$, $RF_1$, $ET$, and $RR$ over their corresponding basic metrics, which are $Och$, $F_1$, $Pr(e \mid F) \times \text{precision}$ and recall, and the number of executed faults in programs.
Figure 3-6. Percentage reductions with number of faults in programs.

RR was always less costly than recall over all versions. This is because the basic metric recall alone in RR exhibits less effective than precision on both single-fault and multiple-fault programs as shown in Table 3-4 and Table 3-5 and the additional term $-Pr(e)/Pr(F) \times Pr(F) = -Pr(e)$ enhances RR on all programs with different number of faults. RO, RF1 and ET were less costly than Och, F1 and $Pr(e | F) \times precision$, respectively, over the versions which contain more than four executed faults. Their percentage reductions mostly increase as the number of faults in programs increase. A reasonable explanation for this is that the Pr(F) for each versions could increase when the number of faults in programs increases and the increased Pr(F) magnifies the effect of the weights $w$ in the terms $-w \times Pr(F)$. 
Results for RQ3: No matter how the weights $w$ affect the effectiveness of the basic metrics of the form $w \times \text{precision}$ based on the results of RQ2, the results of our study suggest that metrics with the additional term $-w \times \Pr(F)$, which are in the form $w \times RP = w \times \Pr(F \mid e) - w \times \Pr(F)$, perform better than metrics without this term, which are in the form $w \times \Pr(F \mid e)$, especially when there are more than four faults in programs.

3.5.5 Results for RQ4: ARD over Precision

To evaluate the effectiveness of ARD, we computed the percentage reductions in average Cost that ARD achieved over precision for both single-fault programs and multiple-fault programs. Table 3-8 shows the percentage reductions in average Cost for both single-fault programs and multiple-fault programs. The averages are shown in the bottommost row.

For single-fault programs, ARD was less costly than precision over each of the ten subject programs, and the average percentage reduction achieved by ARD is 29.74%. For multiple-fault programs, except for the tenth program “HTML Parser”, ARD was less costly than precision over each of other nine subject programs, and the average percentage reduction achieved by ARD is 40.27%. Thus, the experiment results showed that the term $- \Pr(F \mid \neg e)$ in the first line of derivation (3-17) made ARD to outperform precision in both single-fault programs and multiple-fault programs.
Table 3-8. Percentage Reductions for Both Single-fault and Multiple-fault Programs (ARD vs. prec).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>Single-fault programs</th>
<th>Multiple-fault programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>15.90%</td>
<td>41.73%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>46.63%</td>
<td>58.42%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>46.42%</td>
<td>55.89%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>23.69%</td>
<td>41.05%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>19.84%</td>
<td>28.49%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>37.78%</td>
<td>67.86%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>9.02%</td>
<td>37.80%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>28.97%</td>
<td>38.70%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>20.19%</td>
<td>32.96%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>48.95%</td>
<td>-0.24%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>29.74%</td>
<td>40.27%</td>
</tr>
</tbody>
</table>

The term \( Pr(F \mid \neg e) \) in ARD can lower the value of precision for nonfaulty program elements. For example, in single-fault programs \( f_\pi \) for a faulty program statement should be zero [6] such that \( Pr(F \cap \neg e) = 0 \) and \( Pr(F \mid \neg e) = \frac{Pr(F \cap \neg e)}{Pr(\neg e)} = 0 \).

Thus, for a nonfaulty statement, its \( f_\pi \) could be zero or greater than zero such that \( Pr(F \cap \neg e) \geq 0 \) and \( Pr(F \mid \neg e) = Pr(F \cap \neg e)/Pr(\neg e) \geq 0 \). Therefore, for a faulty statement and a nonfaulty statement, if they have the same precisions, ARD for the nonfaulty statement, in which \( Pr(F \mid \neg e) \geq 0 \), could be lower than ARD for the faulty statement, in which \( Pr(F \mid \neg e) = 0 \).

Results for RQ4: The results indicate that the associational risk difference outperforms precision for both single-fault and multiple-fault programs.

3.5.6 Results for RQ5: Metrics \( w \times ARD \) over ARD

To evaluate the effect of weight terms \( w \) in the metrics of the form \( w \times ARD \), namely RDO, RDF1, RDRec and ET2, we compute the percentage reductions achieved over ARD, for both single-fault programs and multiple-fault programs. Table 3-9 shows the
percentage reductions in average Cost for single-fault programs. Per-program and per-metric averages are shown in the rightmost column and the bottommost row, respectively.

Table 3-9. Percentage Reductions for Single-fault Programs (w × ARD vs. ARD).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RDO</th>
<th>RDF1</th>
<th>RDRrec</th>
<th>ET2</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>3.01%</td>
<td>2.95%</td>
<td>-14.43%</td>
<td>-4.36%</td>
<td>1.23%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>32.06%</td>
<td>31.65%</td>
<td>-1.69%</td>
<td>31.70%</td>
<td>31.28%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>43.55%</td>
<td>26.87%</td>
<td>-77.26%</td>
<td>38.12%</td>
<td>43.10%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>22.83%</td>
<td>19.08%</td>
<td>-50.38%</td>
<td>22.77%</td>
<td>22.19%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>33.41%</td>
<td>27.43%</td>
<td>11.43%</td>
<td>27.86%</td>
<td>32.82%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>56.52%</td>
<td>50.71%</td>
<td>-17.03%</td>
<td>55.46%</td>
<td>56.23%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>36.96%</td>
<td>34.46%</td>
<td>22.28%</td>
<td>36.96%</td>
<td>37.03%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>32.40%</td>
<td>30.48%</td>
<td>-12.06%</td>
<td>10.12%</td>
<td>26.58%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>12.54%</td>
<td>11.81%</td>
<td>-65.68%</td>
<td>11.54%</td>
<td>12.57%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>42.92%</td>
<td>35.83%</td>
<td>-134.69%</td>
<td>41.75%</td>
<td>42.90%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>31.62%</td>
<td>27.13%</td>
<td>-33.95%</td>
<td>27.19%</td>
<td>30.59%</td>
</tr>
</tbody>
</table>

Figure 3-7 graphically depicts the percentage reductions in average Cost for single-fault programs. RDO, RDF1 and ET2 were less costly than ARD over most versions of each of the ten subject programs. Their average percentage reductions over all programs were 31.62%, 27.13% and 27.19%, respectively. RDRrec achieved negative reductions over ARD, whose average was -33.95%. The points on the dotted line represent the average percentage reduction for each subject program over all four metrics.
Figure 3-7. Percentage reductions for single-fault programs ($w \times ARD$ vs. $ARD$).

Table 3-10 shows the percentage reductions in average $Cost$ for the multiple-fault programs and it has the same fields as Table 3-9 has.

Table 3-10. Percentage Reductions for Multiple-fault Programs ($w \times ARD$ vs. $ARD$).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RDO</th>
<th>RDF1</th>
<th>RDRec</th>
<th>ET2</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>3.30%</td>
<td>2.34%</td>
<td>-9.03%</td>
<td>2.85%</td>
<td>3.03%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>9.33%</td>
<td>8.47%</td>
<td>-5.66%</td>
<td>9.23%</td>
<td>8.65%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>-3.16%</td>
<td>-4.40%</td>
<td>-96.99%</td>
<td>-0.44%</td>
<td>-4.15%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>12.88%</td>
<td>11.84%</td>
<td>-2.42%</td>
<td>13.24%</td>
<td>12.41%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>9.42%</td>
<td>5.92%</td>
<td>-39.21%</td>
<td>8.10%</td>
<td>8.69%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>-9.36%</td>
<td>-26.48%</td>
<td>-173.31%</td>
<td>-14.20%</td>
<td>-10.85%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>45.00%</td>
<td>45.13%</td>
<td>44.56%</td>
<td>45.61%</td>
<td>43.94%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>29.39%</td>
<td>27.96%</td>
<td>7.49%</td>
<td>29.13%</td>
<td>26.94%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>-8.10%</td>
<td>-9.86%</td>
<td>-29.30%</td>
<td>-9.23%</td>
<td>-8.21%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>-18.04%</td>
<td>-18.49%</td>
<td>-97.18%</td>
<td>-14.31%</td>
<td>-17.28%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>7.07%</td>
<td>4.24%</td>
<td>-40.11%</td>
<td>7.00%</td>
<td>6.32%</td>
</tr>
</tbody>
</table>

Figure 3-8 graphically depicts the percentage reductions in average $Cost$ for multiple-fault programs. $RDO$, $RDF1$ and $ET2$ exhibited negative reductions (increases) over $ARD$ for
some programs, but their average percentage reductions over all programs were 7.07%, 4.24% and 7.00%, respectively. *RDRec* achieved negative reductions over *ARD*, whose average was -40.11%. The points on the dotted line represent the average percentage reduction for each subject program over all four metrics.

![Figure 3-8](image.png)

Figure 3-8. Percentage reductions for multiple-fault programs (*w × ARD* vs. *ARD*).

The metrics *RDO*, *RDF1* and *ET2* exhibited average positive reductions over the baseline metric *ARD*, which means that the weights \( \sqrt{\Pr(e)/\Pr(F)} \), \( (2 \times \Pr(e))/(\Pr(F) + \Pr(e)) \) and \( \Pr(e | F) \) enhance *RDO*, *RDF1* and *ET2*, respectively, on both single-fault and multiple-fault programs. *RDRec* exhibited average negative reduction over the baseline metric *ARD*, which means that the weight \( \Pr(e)/\Pr(F) \) cannot enhance *RDRec*.

**Results for RQ5:** The weights \( \sqrt{\Pr(e)/\Pr(F)} \), \( (2 \times \Pr(e))/(\Pr(F) + \Pr(e)) \) and \( \Pr(e | F) \) in the metrics *RDO*, *RDF1* and *ET2*, respectively, help with both
single-fault and multiple-fault programs. But the weight \( \Pr(e)/\Pr(F) \) in \( RDR_{Rec} \) is not helpful on both single-fault and multiple-fault programs.

3.5.7 Results for RQ6: Metrics \( w \times ARD \) over Metrics \( w \times precision \)

We empirically compared the percentage reductions in average \( Cost \) achieved by each of the metrics of the form \( w \times ARD \) (risk Ochiai, risk \( F1 \)-measure, risk recall and \textit{enhanced Tarantula2}) over the corresponding basic metrics of the form \( w \times precision \) (Ochiai, \( F1 \)-measure, recall and \( \Pr(e \mid F) \times precision \)), respectively. Table 3-11 shows the percentage reductions for single-fault programs.

Table 3-11. Percentage Reductions for Single-fault Programs (\( w \times ARD \) vs. \( w \times prec \)).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RDO/Och</th>
<th>RDF1/F1</th>
<th>RDR_{Rec}/recall</th>
<th>ET2/wprec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>7.12%</td>
<td>7.07%</td>
<td>12.40%</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>-1.29%</td>
<td>-4.03%</td>
<td>14.81%</td>
<td>-4.72%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>24.60%</td>
<td>16.64%</td>
<td>10.68%</td>
<td>8.62%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>3.11%</td>
<td>3.63%</td>
<td>1.54%</td>
<td>3.46%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>8.36%</td>
<td>8.30%</td>
<td>12.95%</td>
<td>0.67%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>7.45%</td>
<td>6.05%</td>
<td>9.86%</td>
<td>3.73%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>1.92%</td>
<td>0.42%</td>
<td>0.90%</td>
<td>0.42%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>24.59%</td>
<td>25.18%</td>
<td>12.49%</td>
<td>0.32%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>2.10%</td>
<td>1.54%</td>
<td>-0.23%</td>
<td>0.40%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>7.73%</td>
<td>3.56%</td>
<td>36.11%</td>
<td>1.59%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>8.57%</td>
<td>6.84%</td>
<td>11.15%</td>
<td>1.45%</td>
</tr>
</tbody>
</table>

Figure 3-9 graphically depicts the same percentage reductions in average \( Cost \) as Table 3-11. \( RDO, RDF1, RDR_{Rec} \) and \( ET2 \) exhibited positive reductions (decreases) over \( Och, F1, recall \) and \( \Pr(e \mid F) \times precision \), respectively. Their average percentage reductions over all programs were 8.57%, 6.84%, 11.15% and 1.45%, respectively.
Figure 3-9. Percentage reductions for single-fault programs \((w \times ARD \ vs. \ w \times \text{prec})\).

Table 3-12 shows the percentage reductions in average \(Cost\) achieved by each of the metrics of the form \(w \times ARD\) over the corresponding basic metrics for multiple-fault programs.

Table 3-12. Percentage Reductions for Multiple-fault Programs \((w \times ARD \ vs. \ w \times \text{prec})\).

<table>
<thead>
<tr>
<th>#</th>
<th>Programs</th>
<th>RDO/Och</th>
<th>RDF1/F1</th>
<th>RDRec/recall</th>
<th>ET2/wprec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daikon</td>
<td>1.51%</td>
<td>0.51%</td>
<td>21.83%</td>
<td>0.05%</td>
</tr>
<tr>
<td>2</td>
<td>Eventbus</td>
<td>4.22%</td>
<td>1.57%</td>
<td>20.19%</td>
<td>1.45%</td>
</tr>
<tr>
<td>3</td>
<td>Jaxen</td>
<td>-0.68%</td>
<td>-4.69%</td>
<td>16.71%</td>
<td>-1.94%</td>
</tr>
<tr>
<td>4</td>
<td>Jster</td>
<td>2.98%</td>
<td>0.18%</td>
<td>24.23%</td>
<td>0.57%</td>
</tr>
<tr>
<td>5</td>
<td>JExel</td>
<td>5.01%</td>
<td>6.54%</td>
<td>36.17%</td>
<td>5.18%</td>
</tr>
<tr>
<td>6</td>
<td>Jparsec</td>
<td>17.22%</td>
<td>14.61%</td>
<td>44.22%</td>
<td>10.84%</td>
</tr>
<tr>
<td>7</td>
<td>AC codec</td>
<td>1.18%</td>
<td>-2.06%</td>
<td>8.39%</td>
<td>-0.89%</td>
</tr>
<tr>
<td>8</td>
<td>AC Lang</td>
<td>1.92%</td>
<td>0.39%</td>
<td>47.45%</td>
<td>0.01%</td>
</tr>
<tr>
<td>9</td>
<td>EclipseDraw2d</td>
<td>4.48%</td>
<td>3.68%</td>
<td>23.99%</td>
<td>2.67%</td>
</tr>
<tr>
<td>10</td>
<td>HTML Parser</td>
<td>-6.14%</td>
<td>-6.29%</td>
<td>32.43%</td>
<td>-2.93%</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>3.17%</td>
<td>1.44%</td>
<td>27.56%</td>
<td>1.50%</td>
</tr>
</tbody>
</table>
Figure 3-10 graphically depicts the same percentage reductions in average Cost as Table 3-12. With the multiple-fault programs, \( RDO \), \( RDF1 \), \( RDRec \) and \( ET2 \) achieved positive reductions over \( Och \), \( F1 \), recall and \( Pr(e \mid F) \times precision \), respectively, which average 3.17%, 1.44%, 27.56% and 1.50%, respectively, over all programs.

![Multiple-fault Programs - W*ARD vs. W*Prec (Baseline)](image)

Figure 3-10. Percentage reductions for multiple-fault programs (\( w \times ARD \) vs. \( w \times prec \)).

The metrics \( RDO, RDF1, RDRec \) and \( ET2 \), which are in the form \( w \times ARD = w \times Pr(F \mid e) - w \times Pr(F \mid \neg e) \), exhibited positive reductions (decreases) over their corresponding basic metrics, which are in the form \( w \times precision, Och, F1, recall \) and \( Pr(e \mid F) \times precision \), respectively, for both single-fault and multiple-fault programs. This indicates that the additional term \( -w \times Pr(F \mid \neg e) \) helps with both single-fault and multiple-fault programs.

**Results for RQ6:** The results of our study suggest that metrics with the additional term \( -w \times Pr(F \mid \neg e) \), which are in the form \( w \times ARD = w \times Pr(F \mid e) - w \times
Pr(F | ¬e), perform better than metrics without this term, which are in the form \( w \times \Pr(F | e) \).

### 3.5.8 Discussion

The results of our study indicate that effective CBSFL metrics incorporate a product term of the form \( w \times RP \), the form \( w \times \max(RP, RR) \) or the form \( w \times ARD \), where \( w \) is a term that downweights \( RP \) or \( ARD \) for a program element \( e \) when it cannot be the cause of all observed failures. Note that even if a program element \( e \) is structurally independent of one or more faults in a program (is not linked to them by program dependences), so that they do not confound its CBSFL score, evidence of their presence, such as low \( \Pr(e | F) \), arguably justifies lowering the score that \( e \) would receive without this information.

The weight term \( w \) varies between metrics: for \( AK \) in (3-12) it is \( \sqrt{\Pr(e)} \); for \( SK \) in (3-13) it is the root square of support, \( \sqrt{\Pr(F \cap e)} \); for \( RO \) in (3-11) and \( RDO \) in (3-18), it is \( \sqrt{\Pr(e)/\Pr(F)} \); for \( RF1 \) in (3-8) and \( RDF1 \) in (3-19), it is \( (2 \times \Pr(e))/\Pr(F) + \Pr(e)) \); for \( RR \) in (3-3) and \( RDRec \) in (3-20), it is \( \Pr(e)/\Pr(F) \); for \( ET \) in (3-16) and \( ET2 \) in (3-21), it is \( \Pr(e | F) \). Somewhat surprisingly, the results of RQ1, RQ2 and RQ5 show that the weights \( w \) in the metrics of the forms \( w \times RP \), \( w \times precision \) and \( w \times RP \), respectively apparently help with both single-fault and multiple-fault programs except for the weights \( \Pr(e)/\Pr(F) \) in recall and \( RDRec \).

For the single-fault programs, the weight \( w \) is helpful for \( RP \) and \( ARD \). Consider a faulty program element and a nonfaulty program element with the same scores estimated by \( RP \), where both values of \( \Pr(F \cap e) \) are different but the ratios of \( \Pr(F \cap e) \) to \( \Pr(e) \) are the same. For example, Table 3-13 shows the program element coverage information.
for both faulty and nonfaulty elements which have different values of \( \Pr(F \cap e) \) but the same ratios of \( \Pr(F \cap e) \) to \( \Pr(e) \), which is \( \Pr(F \mid e) \). \( RP \) cannot distinguish these two elements, but the weight \( \sqrt{\Pr(e)/\Pr(F)} \) can help \( RP \) to distinguish these two elements.

We can see that both elements have the same scores (1.7) estimated by \( RP \) but different scores estimated by \( RO \), and \( RO \) gives the faulty element a higher score (2.09) than the nonfaulty element’s score (1.32).

Table 3-13. An Example for Two Elements Having the Same Precision Scores

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of tests</th>
<th>Probabilities</th>
<th>Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( f_e ) ( f_\bar{e} ) ( p_e ) ( p_\bar{e} ) ( \Pr(F \cap e) ) ( \Pr(e) ) ( \Pr(F \mid e) ) ( \Pr(F) ) ( RP ) ( RO )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faulty</td>
<td>5 0 5 10 0.25 0.5 0.5 0.33 1.7 2.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonfaulty</td>
<td>2 3 2 13 0.1 0.2 0.5 0.33 1.7 1.32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the multiple-fault programs, the weight \( w \) is a term that downweights \( RP \) and \( ARD \) for a program element \( e \) when it cannot be the cause of all observed failures. The function of the weight \( w \) may also be partly explained by a remark of Baah et al. [9], suggesting that recall can serve to penalize SFL scores that are based on inadequate samples for individual statements:

“Consider a statement \( s \) for which \( \Pr(s \mid F) \) is very low. This statement will tend to be covered by few failing test cases and perhaps by few test cases overall. Thus, the sample of failing test cases available for estimating either \( \Pr(F \mid s) \) or the causal effect of \( s \) on failures is likely to be quite small.” [9]

The problem referred to is the imprecision inherent in estimating the frequency of rare events. Baah et al.’s remark applies to \( SK \) and \( ET (ET2) \), whose weight terms are \( \sqrt{\Pr(F \cap e)} \) and \( \Pr(e \mid F) = \Pr(F \cap e)/\Pr(F) \), respectively, because both metrics
of support, Pr(F ∩ e). The remark does not apply to RO (RDO), AK, RF1 (RDF1), and RR (RDRec), whose weights are \(\sqrt{\text{Pr}(e)/\text{Pr}(F)}, \sqrt{\text{Pr}(e)}, (2 \times \text{Pr}(e))/(\text{Pr}(F) + \text{Pr}(e)),\) and \(\text{Pr}(e)/\text{Pr}(F),\) respectively. Consider an element e for which \(\text{Pr}(F | e) = \text{Pr}(F \cap e)/\text{Pr}(e)\) is very low. This could be either because \(\text{Pr}(F \cap e)\) is very low or because \(\text{Pr}(e)\) is very high. If \(\text{Pr}(F \cap e)\) is very low, the scores produced by SK and ET will be penalized by their weights. If \(\text{Pr}(e)\) is very high, the scores produced by RO, AK, RF1, and RR might be boosted by their weights. However, their RP terms, which are very low because \(\text{Pr}(F | e)\) is very low, may counter this effect and keep the scores low.

The results of RQ3 show that the additional terms \(-w \times \text{Pr}(F)\) in the metrics of the form \(w \times RP\) enhance the metrics of the form \(w \times RP\) on multiple-fault programs. The results of RQ6 show that the additional terms \(-w \times \text{Pr}(F | \neg e)\) in the metrics of the form \(w \times ARD\) enhance the metrics of the form \(w \times ARD\) on both single-fault and multiple-fault programs.

Now, consider the comparison of precision, relative precision and associational risk difference. Again, relative precision and precision are rank equivalent, so they will take the same costs to locate faults in programs. Section 3.4.1 exhibits the relationship of relative precision and associational risk difference. The results of RQ4 show that associational risk difference outperforms precision (or relative precision) on both single-fault and multiple-fault programs.

The metrics SK, RO, RF1, RR, ET, RDO, RDF1, RDRec, and ET2 exhibited positive reductions over precision (or RP) for single-fault program, which average 50.18%, 45.10%, 43.13%, 52.31%, 43.78%, 50.83%, 48.00%, 9.36% and 47.66%, respectively, and for multi-fault programs, which average 50.23%, 49.80%, 48.70%, 43.63%, 49.40%, 45.10%, 43.13%, 52.31%, 43.78%, 50.83%, 48.00%, 9.36% and 47.66%, respectively.
44.09%, 42.84%, 18.23% and 44.27%, respectively. Thus, for both single-fault and multi-fault programs, RR and SK, respectively, are the most effective of the metrics we compared. However, for multiple-fault programs, SK, RO, RF1, and ET performed very similarly. It seems safe to assume that most large programs have multiple latent faults, but the results we obtained depend on failures from multiple faults actually occurring on the tests used in CBSFL. Thus, in choosing among these metrics, it is desirable to seek evidence for or against the occurrence of failures due to multiple faults.

Compared to the metrics SK, RO, RF1, RR, ET, RDO, RDF1, RDRec, and ET2, the PS2 metric exhibited negative reductions over RP for single-fault programs, which average -51.14%, and positive reductions over RP for multiple-fault programs, which average 9.26%. As mentioned in Section 3.3.3, PS2 is not of one of the forms \( w \times \text{precision} \), \( w \times \text{RP} \), \( w \times \max(\text{RP}, \text{RR}) \) or \( w \times \text{ARD} \).

### 3.5.9 Study Limitations

The results and conclusions of our empirical study are subject to the usual threats to validity for SFL studies [4]. Naturally, they should not be considered conclusive unless they are confirmed in a number of additional studies collectively involving many diverse and realistic subject programs, faults, and test sets. We believe that the SFL research community must pursue replicative studies and meta-analysis of their results. Our study does have the advantage, however, that its results are supported by the algebraic and probabilistic analysis presented in the chapter. Another issue is that simply ranking program elements and examining them until a fault is found is a simplistic approach to fault localization, because for example it ignores developers’ time constraints, discards
some of the information in continuous scores, and does not account for other relevant information available to developers. It does provide a fairly objective and informative means of evaluating and comparing metrics, however.

Steimann et al. [4] considered two modes for evaluating Cost in presence of multiple faults, which are (1) one-at-a-time mode, where one fault is identified and fixed, and then the fault localization process is repeated and (2) many-at-a-time mode, where all faults are identified and fixed in one run of the fault localization process. In our empirical study with multiple-fault programs, we did not employ knowledge of how many faults were in a program version, because we did not have access to the programs. Hence, many-at-a-time mode was not applicable. We employed a modified version of one-at-a-time mode, in which Cost was determined by identifying, but not fixing, the first fault, and in which the fault localization process was not repeated.

Importantly, the CBSFL metrics compared in our study were not compared empirically to techniques based on causal inference theory, such as those of Baah et al. [9][10], Gore and Reynolds [32], and Shu et al. [40]. The former metrics do not adjust for confounding bias, even heuristically, while the latter techniques do. Baah et al. [9] showed that plugging the causal effect estimate obtained with their statistical-regression based technique in place of the precision terms in Ochiai and the F1 measure improved their performance substantially. We expect this would also occur with other metrics considered in this chapter.
3.6 Related Work

Liblit et al. [31] pointed out, in the context of predicate-level debugging, that $\Pr(F \mid q \text{ is true})$ is a biased SFL metric, and they proposed the metric $\text{Increase} = \Pr(F \mid q \text{ is true}) - \Pr(F \mid q \text{ is evaluated})$ instead. Baah et al. [9] proposed using formal causal inference theory in statement-level SFL. For a given statement $s$, they control, using statistical regression, for coverage of its forward control dependence predecessor in order to reduce confounding bias when estimating the average causal effect that covering $s$ has upon the occurrence of failures. They showed that causal Ochiai, which is derived by replacing the precision term in Ochiai with their causal effect estimate, combines the advantages of Ochiai and their regression technique and substantially outperforms Ochiai. This method inspired us to embed relative measures into Ochiai and $F1$ to achieve a better performance. Baah et al. [10] also proposed using a matching method to control for coverage of both data and control dependences predecessors. Their empirical results indicated that their method performs better than causal Ochiai. Gore and Reynolds [32] distinguished between control flow and failure flow confounding biases in predicate-level debugging. They also explained that Liblit et al.’s Increase metric employs a heuristic to control for failure flow confounding bias, and proposed a causal-inference based solution to obtain an unbiased suspicious score for a given predicate. Bai et al. [28] analyzed the causal technique proposed by Baah et al. [9] algebraically and probabilistically, and they suggested improvements to the technique.

Hofer et al. [5] evaluated 42 similarity coefficients for debugging spreadsheets applications. Their results indicated that Ochiai, Jaccard, and Sorensen-Dice performed best. In this chapter, we found that Sorensen-Dice and $F1$ are score equivalent.
Landsberg et al. [7] classified 157 metrics into 5 categories by their properties and evaluate the effectiveness of these metrics. PS2 and the new metric $Lex_{Ochiai}$ performed best overall in their empirical study. PS2 is analyzed in Section 3.3.3. There are some effective metrics that we do not discuss in this chapter, because all of these metrics were evaluated or analyzed under the assumption that there is only a single fault in a program. For example, Naish et al. [6] proposed optimal ranking metrics, which exhibited better performance than 22 existing metrics in their empirical study. Lee et al. [41] compared 9 metrics to $Pr(F \mid s)$ and also proposed a way to measure the effectiveness of these metrics as the $Pr(F \mid s)$ value varies. They proved that $Pr(F \mid s)$ and Tarantula produce identical rankings. Wong3 performs the best in their empirical studies.

Steimann et al. [4] proposed a way of improving Tarantula, called $T^*$, that is similar to what we have done with ET. This involves multiplying Tarantula with the term $Confidence = \max(\% \text{ passed}(e), \% \text{ failed}(e))$ [3], where the terms $\% \text{ passed}(e)$ and $\% \text{ failed}(e)$ are the percentages of all passed and failed tests, respectively, that executed $e$. The $Confidence$ term can be expressed in terms of probabilities, as $\max(Pr(e \mid P), Pr(e \mid F))$. In an empirical study Steimann et al. observed that $T^*$ performed better than Tarantula and Ochiai on single-fault programs [4].

3.7 Conclusion

We have analyzed, using basic algebra and probability, the structure of several coverage-based statistical fault localization metrics that have performed well in recent comparisons of many metrics. This analysis indicated that effective CBSFL metrics incorporate a product term of the form $w \times RP$, the form $w \times \max(RP, RR)$ or the form
$w \times ARD$, where $RP$ is relative precision, $RR$ is relative recall, $ARD$ is associational risk difference and $w$ is a term that downweights $RP$ and $ARD$ for a program element $e$ when it cannot be the cause of all observed failures. Based on our analysis, we defined seven “refined” metrics, relative Ochiai, risk Ochiai, relative $F_1$, risk $F_1$, risk recall, enhanced Tarantula and enhanced Tarantula2. We have reported the results of an empirical study, which indicate that the weight $w$ apparently helps with both single-fault and multiple-fault programs. Considering both kinds of programs, our empirical results also indicate that relative recall and symmetric Klosgen, respectively, are the most effective of the metrics we compared. However, for multiple-fault programs, symmetric Klosgen, relative Ochiai, relative $F_1$, and enhanced Tarantula all performed similarly well.
4 CAUSAL INTERACTIONS FOR PROGRAM STATEMENT

INTERACTIONS

4.1 Introduction

The fact that certain software faults are revealed (result in observable failures) only when certain unusual interactions occur between program elements, which trigger failures or enable them to propagate to output, has long been recognized as an important one in the software engineering literature. Software engineering research on such topics as combinatorial interaction testing [42][43] and the feature interaction problem [44][45] explicitly focuses on such unusual interactions. They are also addressed, at least implicitly, by research on program slicing [46]; data flow testing [47][48][49] and, more generally, on program-dependence-based testing [27]. Interactions, or their non-occurrence, also lie behind the well-known phenomena coincidental correctness and failed-error propagation, which are obstacles to both software testing and debugging. Coincidental correctness (CC) [50] arises when a program produces correct output even though a fault is executed. Failed-error propagation (FEP) [21] occurs when an erroneous program state is induced but does not “propagate”\(^1\) to output (does not induce a sequence of erroneous states that results in erroneous output). If an erroneous state can propagate to output, along some program path, but does not, this is due to an interaction between the faulty code that caused the state and other code that prevents it from propagating under certain conditions.

\(^1\) This use of the term “propagate” is somewhat misleading. It is not assumed that erroneous data caused by a fault will necessarily reach the output unchanged. Rather, it is assumed that an initial erroneous state will directly or indirectly cause incorrect output.
The greatest challenge to adequately addressing interactions in software development is their sheer number [44], which may be exponential in the number of program elements that could possibly interact. This issue has been addressed in different ways. In combinatorial interaction testing [42][43], in which an interaction of interest corresponds to a combination of input-parameter values, one typically seeks to find an approximately minimal set of test cases that exercises all $t$-way interactions for a chosen, small value of $t$ (e.g., $t = 2$ or $t = 3$). Similarly, in feature-interaction testing of software product lines [51] one seeks to find a set of tests that cover all $t$-way combinations of product features, for a small value of $t$. Androutsopoulos et al. [21] present evidence of a correlation between failed error propagation and the estimated “squeeziness” of the program code executed after a faulty program element is executed. Squeeziness is a measure, based on conditional entropy, of the expected loss of information (carried in program state) caused by executing a portion of program code. Androutsopoulos et al. suggest comparing estimates of the squeeziness of alternative paths leaving a program element $elmt$ to facilitate the creation of a test case that covers $elmt$ but minimizes the potential for FEP.

In this section, we seek ways to detect two main types of interactions between generic runtime events, which we call fault-revealing interactions and fault concealing interactions. These interactions are causally related to the occurrence of software failures. Our goal is to facilitate the localization of certain software faults that are not well-localized by typical coverage-based statistical fault localization metrics [3][2], which do not take interactions into account. To this end, we adapt and exploit recent advances in causal inference theory and methodology that address the detection of causal interactions [17][18] by integrating statistical methods with analysis of causal graphs. In particular, we adapt certain statistical
measures and tests, developed for studying causal interactions between different exposures (e.g., genetic and environmental factors), for use in the domain of **statistical fault localization** (SFL). These techniques are based on both the *potential outcome (counterfactual) model* of causality [23] and Pearl’s *Structural Causal Model* (SCM) [22]. Their application to SFL raises some challenging special issues, such as the large number of potential interactions in software. We propose and evaluate a preliminary approach to addressing these issues.

In Section 4.2 a motivating example of a fault-concealing interaction is presented. In Section 4.3, necessary background on several topics is presented. In Section 4.4, our definitions of FRIs and FCIs are presented. In Section 4.5, these are related to other notions of interaction. In Section 4.6, our proposed approach to detecting FRIs and FCIs is presented. In Section 4.7, the empirical evaluation of our approach is reported on. Section 4.8 and Section 4.9 show the related work and the conclusion.

### 4.2 Motivating Example

We present an example of real-world program code that can give rise to a fault-concealing interaction. Figure 4-1 shows both a piece of code exhibiting a missing-statement fault and a description of its fix (lines 563–568), from the Mozilla web browser bug tracking system.² If a user has an Adobe Acrobat PDF file open in a browser tab and they attempt to open another new tab with quick-keys (Ctrl+T), the browser incorrectly presents an Acrobat dialog rather than a new browser tab. Given a test case that realizes this scenario, one can determine with a debugger that object “style” initialized at line 531

---

² Mozilla bug tracking system (id: 273456): http://hg.mozilla.org/mozilla-central/file/b4193c7aa44c/widget/src/windows/NSWindow.cpp
of in Figure 4-1 carries erroneous information, which could be produced by the call to function “WindowStyle” at line 531 or by execution of lines 542~555. The erroneous information eventually propagates to object “mWnd” at line 569 causing the erroneous dialog to appear. The code of the fix (line 563~568) corrects the erroneous information in object “style” by computing its bitwise OR with “WS_DISABLED” before “style” is used by “mWnd” at line 569. The value of “WS_DISABLED” is 0x08000000L.\(^3\) Therefore, the statement at line 567 can be seen as a blocking the erroneous state from propagating to the output.

```
Line    Line
531     DWORD style = WindowStyle();
534     ... if (mWindowType == eWindowType_popup) {
542     ... } else if (mWindowType == eWindowType_invisible) {
544     style &= ~0x40000000; // WS_CHILDWINDOW
545     } else {
547     if (aInitData->clipChildren) {
548     style |= WS_CLIPCHILDREN;
549     } else {
550     style &= ~WS_CLIPCHILDREN;
551     }
552     if (aInitData->clipSiblings) {
553     style |= WS_CLIPSIBLINGS;
554     }
555     }
563     // Plugins are created in the disabled state so that they can't
564     // steal focus away from our main window. This is especially
565     // important if the plugin has loaded in a background tab.
566     + if (aInitData->mWindowType == eWindowType_plugin) {
567     + style |= WS_DISABLED;
568     + }
569     mWnd = ::CreateWindowExW(extendedStyle,
572     ... style,
572     ...);
```

Figure 4-1. A piece of Mozilla fix code

\(^3\) Microsoft Dev Center website -- https://msdn.microsoft.com/en-us/library/windows/desktop/ms632600(v=vs.85).aspx
4.3 Causal Interaction Concepts

Rothman’s *sufficient-cause framework* [52] conceptualizes causation in terms of collection of different *sufficient causes*, each of which consists of a minimal set of conditions that inevitably produce an outcome of interest. VanderWeele and Robins [53] related Rothman’s sufficient-cause framework to the potential outcomes framework [23] and formalized the notion of a *sufficient cause interaction* (SCI) to describe a form of interaction (called *synergism*) in the sufficient-cause framework. For example, smoking is a cause of lung cancer, but it is not a sufficient cause. A sufficient cause may consist of smoking together with certain genetic and environmental risk factors. With Boolean causal variables, a sufficient cause can be represented as a conjunction of variables and/or their negations. Two variables are considered to interact if they occur in the same sufficient cause.

VanderWeele et al. [17][18] discussed causal interactions between two exposures. Let $G$ and $E$ denote two binary exposure variables of interest (e.g., indicating the presence or absence of a genetic risk factor and an environmental risk factor, respectively) and let $D$ denote a binary outcome variable (e.g., indicating the presence or absence of a particular disease). In the counterfactual framework, $D_{ge}$ represents the potential outcome for an individual (or unit) when $G = g$ and $E = e$ for fixed values $g$ and $e$. There are four possible potential outcomes, $D_{11}, D_{10}, D_{01}$ and $D_{00}$. In general, we can only observe one of the four potential outcomes as an actual outcome for an individual. To apply causal interaction concepts in programs, we can use the two binary exposures to represent the coverage status of two program elements (1: COVERED, 0: NOT COVERED) and the binary outcome to represent the test outcome (1: FAIL, 0: PASS).
A set of covariates $C$ suffices to control for confounding of the joint causal effect of exposures $G$ and $E$ on $D$ if the potential outcome $D_{ge}$ is conditionally independent of $G$ and $E$ given $C$ [53], which is denoted by

$$D_{ge} \perp \!(G,E) \mid C$$

(4-1)

Since $D$ is a binary outcome, the mean $E[D_{ge}]$ is the risk $\Pr(D_{ge} = 1)$ of developing the outcome. Let $p_{ge}$ denote the probability $\Pr(D_{ge} = 1 \mid C = c)$ of the counterfactual outcome $D_{ge} = 1$ for those with $C = c$. There are four such probabilities: $p_{11}, p_{10}, p_{01},$ and $p_{00}$. If (4-1) holds then [53]:

$$p_{ge} = \Pr(D = 1 \mid G = g, E = e, C = c)$$

(4-2)

In this case, the counterfactual probability $p_{ge}$, which is the causal risk for those with $C = c$, can be estimated without confounding bias by estimating the probability on the right, which is the associational risk for those with $C = c$ and which involves only observable variables. Note that the latter can be estimated directly because it involves only observed variables.

A natural way to assess an interaction between two exposures is to measure the difference between the joint effect of the exposures together and the summed effects of each considered individually [17][18]. This is measured by

$$(p_{11} - p_{00}) - [(p_{10} - p_{00}) + (p_{01} - p_{00})]$$

(4-3)
This is said to be a measure of interaction on the “additive scale” [17][18]. We shall refer to it as the additive interaction measure (AI measure or AIM), and it can be rewritten as

\[ p_{11} - p_{10} - p_{01} + p_{00} \]

(4-4)

We refer to an inequality comparison of AIM and zero as an AI test (AIT). The condition \( p_{11} - p_{10} - p_{01} + p_{00} \neq 0 \) indicates the occurrence of an additive interaction between the two exposures. If \( p_{11} - p_{10} - p_{01} + p_{00} > 0 \), the interaction is said to be positive or synergistic [54]. If \( p_{11} - p_{10} - p_{01} + p_{00} < 0 \), the interaction is said to be negative or antagonistic.

VanderWeele et al. [17][18] pointed out that the value of the AI measure can be estimated by using a linear regression model

\[
\Pr(D = 1|G = g, E = e, C = c) = \alpha_0 + \alpha_1 g + \alpha_2 e + \alpha_3 ge + \alpha_4 c
\]

(4-5)

where \( \alpha_0 = p_{00}, \alpha_1 = p_{10} - p_{00}, \alpha_2 = p_{01} - p_{00}, \alpha_3 = p_{11} - p_{10} - p_{01} + p_{00}, C \) is a vector of covariates, and \( \alpha_4 \) is the unconstrained coefficient of \( c \). Thus, the value of the AI measure can be estimated by the fitted value \( \hat{\alpha}_3 \) of the coefficient \( \alpha_3 \).

Suzuki and VanderWeele [55] discussed the concept of compositional epistasis, or “epistasis in the sense of masking”, which involves one causal factor (e.g., a genetic factor) masking the effect of another. They summarized sufficient conditions for eight epistatic response patterns and described how these eight patterns can be detected by an empirical test, which we call the CE test (CET). The CE test involves a stronger condition than the AI test [56]. The formula used with the CE test varies depending on different potential
outcome combinations. Suzuki and VanderWeele [55] also pointed out that different CE tests can be expressed by different combinations of the coefficients in the model (4-5).

Given four binary potential outcome variables, there are $16 (= 2^4)$ different response types (response patterns), and VanderWeele et al. [54] mentioned that 10 of them are implied by nonadditivity, i.e. these 10 response types correspond to 10 types of causal interactions. Table 4-1 describes these 10 types of causal interaction and shows type numbers, potential outcome response patterns ($D_{11}, D_{10}, D_{01}$ and $D_{00}$), AI test outcomes, and corresponding CE test outcomes. Since both positive and negative values of the AI-measure could indicate an occurrence of an interaction between two exposures, the absolute value of the AI measure can be used to detect such an interaction. The higher this value is for a pair of exposures, the more likely it is that there is an interaction between them. Unfortunately, the AI measure does not allow us to determine the type of an interaction. For example, the AI-measure values cannot be used to distinguish Type 2 and 3 interactions, because both of their AIM values are -1. However, each interaction of type 1, 2, 3, 4, 7, 8, 9, or 10 has a corresponding CET formula, which can be expressed by using the combination of the coefficients in model (4-5) as shown in Table 4-1. These CE tests can be used to distinguish the interaction type. Hence, we use both the AI and CE measures in our approach to help locate pairs of statements involved in fault-revealing and fault-concealing interactions, which we call interaction pairs.
### 4.4 Fault-Revealing and Fault-Concealing Interactions

The causal-interaction concepts and methodology described in the causal inference literature, which were introduced in Section 4.3, provide a principled basis for addressing causal interactions. We seek to explore their applicability to the problem of locating interaction pairs involved in what we have called fault-revealing and fault-concealing interactions. We conceive of these as interactions between runtime events, rather than as interactions between faulty statements as in some other approaches to characterizing fault-related interactions in programs [20][19]. Although the abstract concept of a sufficient-cause interaction (see Section 4.3) encompasses interactions between events, it is not expressed in terms of program events or other software concepts. Therefore, we propose original definitions of fault-revealing and fault-concealing interactions that are expressed in terms of such concepts. Before doing so, we first define generic runtime events, of which such interactions are comprised.

A **generic (runtime) event** of type EvtType is an event involving execution, possibly with specified constraints, of a set Elms of one or more program elements of type

<table>
<thead>
<tr>
<th>Type</th>
<th>D_{11}</th>
<th>D_{10}</th>
<th>D_{01}</th>
<th>D_{00}</th>
<th>AIT</th>
<th>CET formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>AIM &gt; 0</td>
<td>$-\alpha_3 - 2(\alpha_0 + \alpha_1 + \alpha_2) &gt; 0$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>AIM &lt; 0</td>
<td>$-\alpha_3 - 2(\alpha_0 + \alpha_1) &gt; 0$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AIM &lt; 0</td>
<td>$-\alpha_3 - 2(\alpha_0 + \alpha_2) &gt; 0$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>AIM &gt; 0</td>
<td>$\alpha_3 - 2\alpha_0 &gt; 0$</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>AIM &lt; 0</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>AIM &gt; 0</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>AIM &lt; 0</td>
<td>$\alpha_3 - 2(1 - \alpha_0 - \alpha_1 - \alpha_2) &gt; 0$</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AIM &gt; 0</td>
<td>$\alpha_3 - 2(1 - \alpha_0 - \alpha_1) &gt; 0$</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>AIM &gt; 0</td>
<td>$\alpha_3 - 2(1 - \alpha_0 - \alpha_2) &gt; 0$</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>AIM &lt; 0</td>
<td>$2(\alpha_2 - 1) - \alpha_3 &gt; 0$</td>
</tr>
</tbody>
</table>
where the occurrence or non-occurrence of any event of EvtType can be detected automatically and in the same way for any program, e.g., with instrumentation or by runtime monitoring. A basic example of a generic event is coverage of a particular program statement. Note that an event that can be detected only in ways that depend on a particular program’s requirements, such as by using a test oracle, is not a generic event. A compound generic event is a generic event comprised of multiple generic events occurring during the same execution. One example is coverage of a particular pair of statements. Another example is coverage of a particular data dependence. A generic event that is not a compound generic event is said to be a simple generic event. Two generic events \( \text{evt}_1 \) and \( \text{evt}_2 \) are independently inducible if there are four complete program inputs \( w, x, y, \) and \( z \) such that: \( w \) induces neither \( \text{evt}_1 \) nor \( \text{evt}_2 \); \( x \) induces \( \text{evt}_1 \) but not \( \text{evt}_2 \); \( y \) induces \( \text{evt}_2 \) but not \( \text{evt}_1 \); and \( z \) induces both \( \text{evt}_1 \) and \( \text{evt}_2 \). (Note that the occurrences of two independently inducible events are not necessarily statistically independent.)

The following Theorem presents some of the implications for a program’s control structure of the fact that two statements can be executed independently.

**Theorem 1:** For a program or program unit \( P \) having a control-flow graph \( \text{CFG}(P) \), if \( s_1 \) and \( s_2 \) are two statements in \( P \) such that an execution of \( s_1 \) and an execution of \( s_2 \) are independently inducible, then the following conditions hold: (a) neither \( s_1 \) nor \( s_2 \) dominates or postdominates the other; (b) \( s_1 \) is forward control dependent on some vertex and \( s_2 \) is also forward control dependent on some vertex; and (c) if \( P \) is a structured program then there is no vertex \( v \) in \( \text{CFG}(P) \) such that both \( s_1 \) and \( s_2 \) are forward control dependent on \( v \) with respect to the same conditional branch label \( L \).
Proof: Suppose that an execution of statement $s_1$ and an execution of statement $s_2$ are independently inducible. Then there are walks $W_1, W_2, W_3,$ and $W_4$ in $CFG(P)$ such that:

1. $W_1$ does not contain either $s_1$ or $s_2$.
2. $W_2$ contains $s_1$ but not $s_2$.
3. $W_3$ contains $s_2$ but not $s_1$.
4. $W_4$ contains both $s_1$ and $s_2$.

It follows from conditions (2) and (3) that neither $s_1$ nor $s_2$ dominates or postdominates the other. It follows from condition (1) that neither $s_1$ nor $s_2$ dominates the final vertex $v_f$ or postdominates the initial vertex $v_i$. By Lemma 3 of [26], which states that a vertex $u$ is forward control dependent on a vertex $v$ if and only if $u$ does not postdominate $v_f$, $s_1$ is forward control dependent on some vertex and $s_2$ is forward control dependent on some vertex. Therefore, both conditions (a) and (b) of the Theorem hold.

Suppose that $P$ is a structured program, so that each vertex in $CFG(P)$ has at most one forward control dependence predecessor. Assume, for a contradiction, that both $s_1$ and $s_2$ are forward control dependent on the same vertex $v$ with respect to the same edge label $L$. Then $s_1$ and $s_2$ would belong to the same forward control dependence region, and hence by Theorem 4 of [26], $s_1$ and $s_2$ would be chain equivalent, so either (i) $s_1$ would dominate $s_2$ and $s_2$ would postdominate $s_1$ or (ii) $s_2$ would dominate $s_1$ and $s_1$ would postdominate $s_2$.

However, both (i) and (ii) contradict our finding that condition (a) of the Theorem holds, and hence condition (c) of the Theorem must hold. ■

In what follows, when we refer to a runtime event “causing” another event, technically we mean actual causation as defined by [57]. However, the reader’s intuitive notion of event causation in programs should suffice for understanding. We assume that when a fault
is triggered, it causes an erroneous program state to occur, which may involve erroneous
data elements and/or an erroneous execution location (program counter). However, an
erroneous state may or may not cause an observable failure. An erroneous program state $S$
is said to be **fatal** if its occurrence during any execution causes an observable failure to
occur eventually. A program state $S$ is said to be **safe** if its occurrence during any execution
causes it to pass (to not fail). A generic event $evt$ involving execution of a faulty program
element $elm$ is **fault-revealing** if, during some program execution, some occurrence of $evt$
causes a fatal program state $S$. If $f$ is the fault in $elm$ that is responsible for $S$ then $evt$ is
said to **reveal $f$** or to be $f$-**revealing**.

Note that although generic runtime events are automatically detectable, the initial
occurrence of an erroneous program state is **not** automatically detectable in general, because
it depends on the program’s requirements. Also note that there is no requirement that the
set $Elms$ of program elements involved in a fault-revealing generic event $evt$ be identical
to the set $FE_f$ of elements of which the revealed fault $f$ is comprised. The two sets may
differ in the number and granularity of their elements. For example, $Elms$ may contain a
single function and $FE_f$ may be a subset of its statements. However, it is assumed that an
occurrence of $evt$ that reveals $f$ involves the execution of the elements of $FE_f$.

We are now ready to define the two types of interactions that are the focus of this
chapter:

**Definition 1.** A (two-way) **fault-revealing interaction (FRI)** $I$ during an execution
$exe$ is a generic compound event comprised of two independently-inducible events $evt_1$
and $evt_2$ that involve execution of distinct nonempty sets $Elms_1$ and $Elms_2$ of program
elements, respectively, and such that: (1) some co-occurrence of $evt_1$ and $evt_2$ in $exe$
causes a fatal state \( S_F \) and (2) no occurrence of \( evt_1 \) or \( evt_2 \) alone, during any execution, causes a fatal state. The FRI \( l \) is said to be an interaction between \( evt_1 \) and \( evt_2 \). If \( f \) is the fault in some member(s) of \( Elms_1 \) or \( Elms_2 \) that causes \( S_F \) then \( l \) is said to reveal \( f \) or to be \( f \)-revealing.

**Definition 2.** A (two-way) **fault-concealing interaction** (FCI) \( l \) during an execution \( exe \) is a generic compound event comprised of two independently-inducible events \( evt_1 \) and \( evt_2 \) that involve execution of distinct nonempty sets \( Elms_1 \) and \( Elms_2 \) of program elements, respectively, such that there is a fault \( f \) in some member(s) of \( Elms_1 \) for which the following two conditions hold:

1. During some occurrence of \( evt_1 \) in \( exe \), either
   
   a. execution of \( f \) causes an initial erroneous state \( S_E \) to occur, but some occurrence of \( evt_2 \) causes either \( S_E \) or an erroneous state \( S_E^+ \) caused by \( S_E \) to transition to a safe state \( S_S \); or
   
   b. some property \( P \) of the program state, which was caused to hold by some prior occurrence of \( evt_2 \), prevents \( f \) from causing an initial fatal state, although \( f \) is executed.

2. During some occurrence of \( evt_1 \) in a different execution \( exe' \), \( f \) causes an initial fatal state \( S_F \) to occur, but \( evt_2 \) does not occur during \( exe' \).

The FCI \( l \) is said to be an interaction between \( evt_1 \) and \( evt_2 \), and it is said to conceal \( f \) or to be \( f \)-concealing.

Observe that a fault-revealing interaction does not necessarily involve more than one fault. Thus, it is different from the notion of a **fault interaction** studied by Debroy and Wong [19] and DiGiuseppe and Jones [20]. Observe also that if execution of one program element \( elmt_1 \) prevents another, faulty element \( elmt_2 \) from being executed and so prevents a
failure, this is not considered to be a fault-concealing interaction involving \( \text{elm}t_1 \) and \( \text{elm}t_2 \), because only one of the two elements was executed. Finally, note that the conditional probability \( p = \Pr(f \text{ revealed} | e) \) (\( q = \Pr(f \text{ concealed} | e) \)) that a fault \( f \) is actually revealed (actually concealed) given the occurrence of an \( f \)-revealing (\( f \)-concealing) event \( e \) may lie anywhere in the half-open interval \((0, 1]\). We call this the \textit{conditional fault-revelation probability} (\textit{conditional fault-concealment probability}) of \( f \) given \( e \). Naturally, if \( e \) is the only \( f \)-revealing (\( f \)-concealing) event, the difficulty of discovering \( f \) tends to increase (decrease) as \( \Pr(e) \) and \( p \) both decrease (\( \Pr(e) \) and \( q \) both increase).

The foregoing definitions of fault-revealing and fault-concealing interactions are not perfectly precise, because the concepts of “fault” and “erroneous state”, as commonly used in the software engineering literature, are not perfectly precise, involving as they do a programmer’s intentions and the form of repairs. We hope that the definitions of FRI and FCI are nevertheless precise enough to be useful for reasoning about fault localization.

4.5 \textit{Relationships of FRIs and FCIs to Other Notions of Interaction}

Section 4.4 provided general definitions for fault-revealing and fault-concealing interactions. In this section, we focus on interactions involving two program statements, wherein the two generic program events comprising an FRI or FCI each represent an execution of a single statement, and a faulty element is a statement. We first relate FRIs and FCIs in detail to the ten interaction types in Table 4-1. We then informally relate FCIs to failed error propagation. Finally, we relate FRIs and FCIs to Debroy and Wong’s and DiGiuseppe and Jones’ notions of a “fault interaction” [19][20].
4.5.1 Relationships of FRIs and FCIs to the Counterfactual Interaction

Types

Let \( s_1 \) and \( s_2 \) be two statements in a program \( P \) that may be involved in a FRI or FCI. We use \( D_{ge} \) to denote the potential outcome (1: FAIL, 0: PASS) of an execution in which the coverage status of \( s_1 \) and \( s_2 \) is represented by \( g \) and \( e \) respectively (1: COVERED, 0: NOT COVERED).

Refer to the ten causal interaction types described by VanderWeele et al. [54], which are shown in Table 4-1. We first explain why some of them are not applicable to detecting failure-revealing or failure-concealing interactions involving coverage of a pair of program statements. Observe that interactions of types 1, 6, 8, 9 and 10 include the potential outcome \( D_{00} = 1 \), which represents the occurrence of an observable failure when, during an execution, neither of statements \( s_1 \) and \( s_2 \) are covered. Such a failure must be caused by a third statement, which is faulty. In this chapter, we do not investigate interactions involving three or more statements. Therefore, we will not consider interaction types 1, 6, 8, 9 and 10 further. Observe that interactions of type 2, 3, 4, 5 and 7 include the potential outcome \( D_{00} = 0 \), which represents the occurrence of a passing (successful) execution when neither of statements \( s_1 \) and \( s_2 \) is covered. Such a success could occur either because no erroneous program state occurs or because an erroneous state does occur but it is prevented from propagating to the output. We consider type 2, 3, 4, 5, and 7 interactions further in the next few subsections.

4.5.1.1 Type 2 and type 3 interactions
Observe that a type 2 interaction \((D_{11} = 0, D_{10} = 0, D_{01} = 1 \text{ and } D_{00} = 0)\) becomes a type 3 interaction \((D_{11} = 0, D_{10} = 1, D_{01} = 0 \text{ and } D_{00} = 0)\) if we simply swap the roles of statements \(s_1\) and \(s_2\). Therefore, it suffices to consider only one of the two types. In a type 3 interaction, the potential outcome \(D_{11} = 0\) represents the occurrence of a successful execution when both \(s_1\) and \(s_2\) are covered. \(D_{10} = 1\) represents an observable failure occurring when \(s_1\) is covered but \(s_2\) is not covered. \(D_{01} = 0\) represents a success occurring when \(s_1\) is not covered but \(s_2\) is covered. Finally, \(D_{00} = 0\) represents a success occurring when neither \(s_1\) nor \(s_2\) is covered. Together, these four potential outcomes indicate that a type 3 interaction is consistent with the presence of a fault at \(s_1\) that is concealed when \(s_2\) is covered. That is, a type 3 interaction is consistent with a fault-concealing interaction (and with failed error propagation [21]). (Similar reasoning shows that a type 2 interaction is also consistent with FCI and FEP.) A type 3 interaction is also consistent with some other scenarios such as the presence of a fault in any statement \(s_3\) in the same control dependence region [58] as \(s_1\) (the set of statements executed under the same control conditions as \(s_1\)) such that the fault is also concealed when \(s_2\) is covered, because such a statement \(s_3\) is covered if and only if \(s_1\) is covered. This is an inherent limitation of characterizing interactions based on statement coverage alone. As shown in Table 4-1, a type 2 or a type 3 interaction has a negative AI measure and has a corresponding CET formula. We use the CET formula to identify type 2 or type 3 interactions in our proposed approach, which is described in Section 4.6.1.

4.5.1.2 A type 4 interaction
Consider a type 4 interaction \((D_{11} = 1, D_{10} = 0, D_{01} = 0, \text{ and } D_{00} = 0)\). The potential outcome \(D_{11} = 1\) represents the occurrence, during an execution, of an observable failure when both statement \(s_1\) and statement \(s_2\) are covered. \(D_{10} = 0\) represents the occurrence of success when \(s_1\) is covered but \(s_2\) is not covered. \(D_{01} = 0\) represents the occurrence of success when \(s_1\) is not covered but \(s_2\) is covered. \(D_{00} = 0\) represents the occurrence of success when neither statement is covered. Together, these four potential outcomes are consistent with the presence of a fault at \(s_1\) or \(s_2\) (or both) that is (are) revealed only when both statements are covered. That is, a type 4 interaction is consistent with a synergistic fault-revealing interaction. A type 4 interaction is also consistent with some other scenarios such as the presence of a FRI involving any pair of statements \(s_3\) and \(s_4\) in the same control dependence regions as \(s_1\) and \(s_2\), respectively. Again, this is an inherent limitation of characterizing interactions based on statement coverage alone. As shown in Table 4-1, a type 4 interaction has a positive AI measure and has a corresponding CET formula.

**4.5.1.3 A type 5 interaction**

Consider a type 5 interaction \((D_{11} = 0, D_{10} = 1, D_{01} = 1 \text{ and } D_{00} = 0)\). We have discussed the meaning of each of its constituent potential outcomes in the two previous subsections. Taken together, they are consistent with the presence of faults at both statement \(s_1\) and statement \(s_2\) that are revealed by executing each statement individually but that are both concealed when both statements are covered. Thus, a type 5 interaction is consistent with a fault-concealing interaction (and with failed error propagation). A type 5 interaction is also consistent with other scenarios, such as the presence of a FCI involving
any pair of statements $s_3$ and $s_4$ in the same control dependence regions as $s_1$ and $s_2$, respectively. As shown in Table 4-1, a type 5 interaction has a negative AI measure but has no corresponding CET formulas.

4.5.1.4 A type 7 interaction (competing interaction)

Consider a type 7 interaction ($D_{11} = 1$, $D_{10} = 1$, $D_{01} = 1$ and $D_{00} = 0$). Again, we have discussed the meaning of each of its constituent potential outcomes above. Together, these four potential outcomes are consistent with the presence of faults at both statement $s_1$ and statement $s_2$ that are each revealed by covering the two statements individually, where one or both faults may be revealed by covering both statements together. (As with other interaction types, a type 7 interaction is also compatible with other scenarios.) This interaction type is described by VanderWeele et al. as competing antagonism [54]: “it is a competing antagonism because if both exposures are present, they will effectively compete to cause the outcome.” However, competing antagonism does not appear to correspond definitely to a meaningful type of interaction between program statements, because $D_{11} = 1$ could represent either a synergistic fault-revealing interaction or a simple failure that can be triggered by covering either $s_1$ or $s_2$ individually. In the latter case, no FRI or FCI occurs. For this reason, we do not consider employing a statistical test for a type 7 interaction.

To summarize, among the ten abstract interaction types described by VanderWeele et al. [54] and shown in Table 4-1, only five of them are, when applied to program coverage and failure data, consistent with fault-revealing or fault-concealing interactions involving pairs of program statements: a type 2, 3, or 5 interaction is consistent with a fault-
concealing interaction (and with failed error propagation); and a type 4 interaction is consistent with a fault-revealing interaction. Therefore, we shall focus on exploring the use of causal interaction tests to detect these kinds of interactions in programs.

4.5.2 Relationship of FCIs to Failed Error Propagation

We now informally relate fault-concealing interactions to failed error propagation [21], which is one of the causes of coincidental correctness. FEP occurs when a fault causes an erroneous program state but no observable failure occurs, that is, the program’s output is correct. Androutsopoulos et al. [21] pointed out that one kind of FEP occurs when a program removes traces of an erroneous state before it propagates to a point where can be observed. In this case, a state update function along some execution path “loses” erroneous information. (For example, an erroneous variable value may be overwritten with a constant.) If some input triggers the fault but the resulting erroneous state does propagate to the output because no “lossy” state update occurs, this kind of FEP qualifies as an FCI, by conditions 1(a) and 2 of Definition 2.

Here is an example illustrating how FEP occurs in programs. Figure 4-2 shows a simple function along with statement coverage information (1: COVERED, 0: NOT COVERED) and test outcomes (P: PASS, F: FAIL) for six test cases. Statement s2 is faulty, and statement s4 effectively assigns 0 to the variable “y”. On tests with $a > 0$ and $b \leq 0$, the erroneous state caused by covering the faulty statement s2 propagates to the output. However, on tests with $a > 0$ and $b > 0$, the erroneous state caused by covering s2 is prevented from propagating to the output because the value of “y” is overwritten by zero at statement s4. This instance of FEP corresponds to a FCI involving s2 and s4.
### Statements

```plaintext
Func (a, b, c, y){
  s1:    if (a > 0)
  s2:      y = y + 10; (correct: y = y + 1)
  s3:    if (b > 0)
  s4:      y = y * 0; (overwrite the state)
  s5:    if (c > 0)
  s6:      y = y - 10;
  s7:    return y;
}
```

### Test cases

<table>
<thead>
<tr>
<th>Statements</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Outcome (P:pass/F:fail) | P | P | P | F | F | P

---

**Figure 4-2. An example of FEP and FCI**

### 4.5.3 Relationships of FRIs and FCIs to Fault Interactions

Whereas the fault-revealing and fault concealing interactions that we consider may involve either one or two faulty statements, Debroy and Wong [19] and DiGiuseppe and Jones [20] focused specifically on interactions involving multiple faulty statements, which they called fault interactions. (Debroy and Wong called such interactions “fault interference”.) They presented classifications of fault interactions based on comparing the results produced by a program with \( n \) faults when it is run on a set of test inputs to the results produced on the same inputs by each of the \( n \) single-fault programs that can be obtained by removing all but one of the \( n \) original faults. DiGiuseppe et al. also conducted an empirical study of fault interactions in six subject programs with more than 65,000 multiple-fault versions.

Debroy and Wong [19] described, and they and DiGiuseppe and Jones [20] studied, four classes of fault interactions (we use DiGiuseppe’s and Jones’ class names): (1) **fault synergy** (FI1) – failures occur in the multi-fault version that do not occur in any single-fault version; (2) **fault obfuscation** (FI2) – some test cases that fail on some single-fault version do not fail on the multi-fault version; (3) **pass/fail independence** (FI3) – every test
case that fails on any single-fault version also fails on the multi-fault version and any test case that fails on the multi-fault version also fails on one of the constituent single-fault versions; and (4) \textit{multi-type interaction} (FI4) – there exists at least one test case that fails on some single-fault version but does not fail on the multi-fault version, and there also exists at least one test case that fails on the multi-fault version but does not fail on any single-fault version. DiGiuseppe \textit{et al.}'s empirical results indicated that class FI2 was the most prevalent class.

Intuitively, fault synergy and fault obfuscation appear closely related to fault-revealing and fault-concealing interactions, respectively. However, the definitions of the fault interaction classes FI1-FI4 differ significantly from our definitions of FRIs and FCIs, even considering only interactions that involve a pair of faulty statements. Most importantly, the former definitions refer to differences in the input-output behavior of distinct programs (multi-fault vs. single-fault), whereas the latter definitions refer to differences in the internal behavior of distinct executions of the same faulty program. Since the faults in a program are not known prior to fault localization, the definitions of classes FI1-FI4, although they are conceptually useful, cannot be used directly to aid fault localization, whereas we shall show that the definitions of FRIs and FCIs can be.

Figure 4-3 illustrates the relationship between FRI and fault synergy. It shows four versions of a function and four tests T1~T4 with the corresponding outputs. Version v0 is the correct version, version v1 is a two-fault version containing faults at lines Ln3 and Ln5, and versions v2 and v3 are single-fault versions containing faults at lines Ln3 and Ln5, respectively. (Faulty lines are indicated by “*”.) Consider executing the four tests T1~T4 on version v1. Tests T2, T3 and T4 pass, and test T1 is the only test to fail. In this case, a
type 4 interaction, which is a FRI, occurs that involves lines Ln3 and Ln5. The fault is revealed when both faulty statements are covered together. Now consider executing test T1 on versions v1, v2 and v3. Test T1 fails on the two-fault version v1 but does not fail on either of the single-fault versions v2 or v3. Therefore, a fault-synergy interaction exists between the faulty lines Ln3 and Ln5. Thus, we have demonstrated that these two lines are involved in both a FRI and a fault-synergy interaction.

<table>
<thead>
<tr>
<th>Input</th>
<th>Correct(v0)</th>
<th>Fault 1&amp;2 (v1)</th>
<th>Fault 1 (v2)</th>
<th>Fault 2 (v3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ln1</td>
<td>c=0, d=0</td>
<td>c=0, d=0</td>
<td>c=0, d=0</td>
<td>c=0, d=0</td>
</tr>
<tr>
<td>Ln2</td>
<td>if(a &gt; 0){</td>
<td>if(a &gt; 0){</td>
<td>if(a &gt; 0){</td>
<td>if(a &gt; 0){</td>
</tr>
<tr>
<td></td>
<td>c = 0 }</td>
<td>c = 1 }*</td>
<td>c = 1 }*</td>
<td>c = 0</td>
</tr>
<tr>
<td>Ln3</td>
<td>if(b &gt; 0){</td>
<td>if(b &gt; 0){</td>
<td>if(b &gt; 0){</td>
<td>if(b &gt; 0){</td>
</tr>
<tr>
<td></td>
<td>d = 0 }</td>
<td>d = 1 }*</td>
<td>d = 0 }</td>
<td>d = 1 }*</td>
</tr>
<tr>
<td>Ln4</td>
<td>print c &amp; d</td>
<td>print c &amp; d</td>
<td>print c &amp; d</td>
<td>print c &amp; d</td>
</tr>
<tr>
<td>Ln5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ln6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>T1 (1,1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>T2 (1,0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>T3 (0,1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>T4 (0,0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-3. A FRI vs. a FI1 interaction

Figure 4-4 illustrates the relationship between FCI and fault obfuscation. It shows four versions of a function and four tests T1~T4 with the corresponding outputs. Version v0 is the correct version, version v1 is a two-fault version containing faults at lines Ln2 and Ln4, and versions v2 and v3 are single-fault versions containing faults at lines Ln2 and Ln4, respectively. Consider executing the four tests T1~T4 on version v1. Tests T2 and T3 fail, and tests T1 and T4 pass. In this case, a type 5 interaction, which is a FCI, occurs with version v1, in which the erroneous state triggered by covering line Ln2 transitions to a safe state when line Ln4 is covered. Now consider executing test T1 on versions v1, v2, and v3. Test T1 fails on the single-fault version v2 but does not fail on the two-fault version.
v1. Therefore, a fault-obfuscation interaction exists between two faulty lines Ln2 and Ln4.

Thus, we have demonstrated that these two lines are involved in both a FCI and a fault-obfuscation interaction.

<table>
<thead>
<tr>
<th>Input</th>
<th>Correct(v0)</th>
<th>Fault 1&amp;2(v1)</th>
<th>Fault1(v2)</th>
<th>Fault2 (v3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ln1</td>
<td>if(a &gt; 0)</td>
<td>if(a &gt; 0)</td>
<td>if(a &gt; 0)</td>
<td>if(a &gt; 0)</td>
</tr>
<tr>
<td>Ln2</td>
<td>y=y+1</td>
<td>y=y+1</td>
<td>y=y+1</td>
<td>y=y+1</td>
</tr>
<tr>
<td>Ln3</td>
<td>if (b &gt; 0){</td>
<td>if (b &gt; 0)</td>
<td>if (b &gt; 0)</td>
<td>if (b &gt; 0)</td>
</tr>
<tr>
<td>Ln4</td>
<td>y=y-2</td>
<td>y=y mod 2}</td>
<td>y=y-2</td>
<td>y=y mod 2}</td>
</tr>
<tr>
<td>Ln5</td>
<td>print y</td>
<td>print y</td>
<td>print y</td>
<td>print y</td>
</tr>
</tbody>
</table>

Figure 4-4. A FCI vs. a FI2 interaction

Note that the same pair of statements may be involved in both a FRI and a FCI that are induced by different inputs. This scenario seems closely related to a multi-type interaction. Figure 4-5 illustrates this. It shows four versions of a function and seven test cases T1~T7 with corresponding outputs. Version v0 is the correct version, version v1 is a two-fault version containing faults at lines Ln2 and Ln4, and versions v2 and v3 are single-fault versions containing faults at lines Ln2 and Ln4, respectively. First, consider executing the four tests T1, T2, T3 and T4 on version v1. Tests T2 and T3 fail because they cover the faulty lines Ln2 and Ln4, respectively. However, test T1, which covers both lines Ln2 and Ln4, does not fail. In this case, a type 5 interaction, which is a FCI, occurs that involves lines Ln2 and Ln4 in version v1. The erroneous state triggered by covering line Ln2 transitions to a safe state when line Ln4 is covered. Now consider executing the four tests T4~T7 on version v1. Tests T4, T6 and T7 pass, and test T5 is the only test to
fail when both faulty lines are covered together. In this case, a type 4 interaction, which is a FRI, occurs that involves lines Ln2 and Ln4. Thus, different sets of tests demonstrate a FCI and a FRI with version v1.

Consider executing test T1 on versions v1, v2, and v3. Test T1 fails on single-fault versions v2 and v3, but it does not fail on the two-fault version v1. Hence, a fault-obfuscation interaction exists between the faulty lines Ln2 and Ln4. Now consider executing test T5 on versions v1, v2, and v3. Test T5 fails on the two-fault version v1 but does not fail on the single-fault versions v2 or v3. Therefore, a fault-synergy interaction also exists between the faulty lines Ln2 and Ln4. This implies that a multi-type interaction exists between these lines. Thus, we have demonstrated that these two lines are involved in a FCI, a FRI, and a multi-type fault interaction.

<table>
<thead>
<tr>
<th>Input</th>
<th>Correct(v0)</th>
<th>Fault 1&amp;2(v1)</th>
<th>Fault 1 (v2)</th>
<th>Fault 2 (v3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ln1</td>
<td>(a, b, y)</td>
<td>(a, b, y)</td>
<td>(a, b, y)</td>
<td>(a, b, y)</td>
</tr>
<tr>
<td>Ln2</td>
<td>if(a &gt; 0) {</td>
<td>if(a &gt; 0) {</td>
<td>if(a &gt; 0) {</td>
<td>if(a &gt; 0) {</td>
</tr>
<tr>
<td></td>
<td>y = y × 3  }</td>
<td>y = y × 4 }*</td>
<td>y = y × 4 }*</td>
<td>y = y × 3 }*</td>
</tr>
<tr>
<td>Ln3</td>
<td>if (b &gt; 0) {</td>
<td>if (b &gt; 0) {</td>
<td>if (b &gt; 0) {</td>
<td>if (b &gt; 0) {</td>
</tr>
<tr>
<td></td>
<td>y = y mod 3}</td>
<td>y = y mod 3 }*</td>
<td>y = y mod 2 }</td>
<td>y = y mod 3 }*</td>
</tr>
<tr>
<td>Ln5</td>
<td>print y</td>
<td>print y</td>
<td>print y</td>
<td>print y</td>
</tr>
<tr>
<td>Ln6</td>
<td></td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
</tr>
<tr>
<td>T1</td>
<td>(1, 1, 1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>(1, 0, 3)</td>
<td>9</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>T3</td>
<td>(0, 1, 8)</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>T4</td>
<td>(0, 0, 2)</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T5</td>
<td>(1, 1, 2)</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>T6</td>
<td>(1, 0, 0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T7</td>
<td>(0, 1, 1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4-5. A FCI and a FRI vs. a FII4 interaction

Finally, pass/fail independence suggests a lack of interaction on the additive scale [17].
4.6 Proposed Approach to Locating Statements Involved in FRIs and FCIs

In this section, we propose an approach, based on the causal interaction tests mentioned in the section 4.3, for locating statements involved in fault-revealing and fault-concealing interactions. Then, we discuss the advantages of our approach compared to Masri’s fault localization technique [59], which employs information flow coverage information.

4.6.1 Proposed Approach

While Baah et al. [9] and Bai et al. [28] focus on situations involving one exposure (which they called the “treatment”), our study focuses on situations involving two exposures. We adapt Baah et al.’s CSFL technique and employ part of Bai et al.’s analysis of it in order to use the causal interaction tests described in Section 4.4 to identify fault-revealing and fault-concealing interactions between two program statements $s_1$ and $s_2$. To adjust for confounding enabled by control dependences (only), we adjust for coverage of the forward control dependence predecessors $\text{pred}(s_1)$ and $\text{pred}(s_2)$ of $s_1$ and $s_2$, respectively, if they exist. In well-structured programs, each statement should have at most one forward control dependence predecessor [28].

The binary exposure variables $X_1$ and $X_2$ indicate the coverage status of statements $s_1$ and $s_2$, respectively (1: COVERED, 0: NOT COVERED). Similarly, the binary covariates $C_1$ and $C_2$ indicate the coverage status of $\text{pred}(s_1)$ and $\text{pred}(s_2)$, respectively. The outcome variable $D$ represents the test outcome (1: FAIL, 0: PASS).

Based on the discussion of model (4-5) in Section 4.3, we could use the absolute value of the estimated AI measure $\hat{\alpha}_3$ as a suspiciousness score to locate pairs of interacting
statements, because the occurrence of an interaction causes either a positive or a negative value of the AI measure. Note that in our study we locate type 2, 3, 4 and 5 interactions, in which their corresponding AI measures are -1, -1, 1 and -2, respectively (in ideal situations). If we use the absolute values of AI measure as a suspiciousness score, a type 5 interaction pair is likely located before one of type 2, 3, and 4 interaction pairs is located. For example, if two types of interactions coexist in a program, let say types 4 and 5 interaction pairs, the type 5 interaction pair is more likely located before the type 4 interaction pair is located because the absolute value of AI measure for the type 5 interaction pair is higher than the absolute value of AI measure for the type 4 interaction pair. However, because estimates of the AI measure with large standard errors are unreliable, we “penalize” such estimates in the manner described by Bai et al. [60], who used the absolute value of the $t$-statistic for an estimated regression coefficient, rather than the coefficient estimate itself, as a suspiciousness score. We use both the absolute values of AI measure, which is the fitted coefficient $\hat{\alpha}_3$ in model (4-5), and the absolute value of the $t$-score for the fitted coefficient $\hat{\alpha}_3$ in model (4-5) as suspiciousness scores. The $t$-score is $t = \hat{\alpha}_3 / std(\hat{\alpha}_3)$, where $std(\hat{\alpha}_3)$ is the standard error of $\hat{\alpha}_3$.

Figure 4-6 shows our algorithm $FindInteractions$ for locating pairs of statements involved in fault-revealing and fault-concealing interactions. There are two stages in the algorithm. In the first stage (lines L1~L12), pairs of statements satisfying a property called dual-exposure positivity, which is defined below, are selected and the AI measure and $t$-score are computed for each pair. These selected pairs of statements are called candidate interaction pairs. However, we still need to determine the interaction type of the candidate interaction pair by testing for all possible interaction types, which includes interactions of
types 2, 3, 4 and 5, because we have no knowledge about the interaction type we are looking for in advance. To reduce the cost of determining the interaction type for each candidate interaction pair, in the second stage (lines L13~L23) we use the AI measure and the CET formulas to decide the order in which interaction types are checked, as explained below.

The inputs to algorithm \textit{FindInteractions} include: (1) a set \textit{StmtSet} of executed statements, which can be determined from execution trace files or statement coverage profiles; (2) the system dependence graph (SDG) of the subject program; and the number \textit{Num} of possible interaction pairs that the user would like to identify, which can be obtained using tools such as JavaPDG [61]. \textit{FindInteractions} invokes the following functions: \textit{DEpos()}, which checks whether a pair of statements satisfies the dual-exposure positivity property; \textit{AIM()} and \textit{tscore()}, which compute the AI measure and the t-score for a pair of statements, respectively; \textit{Sort()}, which sorts a set of statement pairs in nonincreasing order of the absolute values of their AI measures or the absolute values of their t-scores; \textit{infoflow()}, which checks whether there is a SDG path that is realized by at least one test case from the first statement to the second statement; \textit{CET2()}, which returns true if a pair of statements is most likely to exhibit a type 2 interaction based on its test formula in Table 4-1; and \textit{CET3()}, which returns true if a pair of statements is most likely to exhibit a type 3 interaction based on its test formula in Table 4-1. Note that the SDG is used in computing the functions \textit{DEpos()}, \textit{AIM()}, \textit{tscore()}, and \textit{infoflow()}. 
Algorithm *FindInteractions*

**Input:**
- StmtSet: a set of executed statements from coverage profiles
- SDG: system dependence graph
- Num: number of possible interaction pairs users want to identify.

**Output:** Num interaction pairs and corresponding types

```
L1  pair (su, sv, val, ts) p   // val: pair p’s AIM, ts: p’s t-score
L2  Set<pair p> P            // set of candidate interaction pairs
L3  Set<pair p> IP           // set of interaction pairs
// The 1st stage
L4  for each stmt si in StmtSet {
L5      for each stmt sj in StmtSet {
L6          if(DEpos(si, sj)){
L7              p.su = si;
L8              p.sv = sj;
L9              p.ts = tscore(si, sj);    // Compute t-score for each pair
L10             p.val = AIM(si, sj);    // Compute AIM for each pair
L11             P.add(p);                // Add pair p to set P
L12         }
L13      }
L14  P = Sort(P);             // sort P by |AIM| or |t-score|
// The 2nd stage
L15  for each pair p in P {
L16       if( !infoflow(p) ) continue
L17       if(p.val > 0){      // FRI
L18           Check type 4 first and then other types in random order. If pair p is an interaction pair, add it to IP.
L19        }else{             // FCI
L20           if(CET2(p)){
L21               tp = 2
L22           }else if (CET3(p)){
L23               tp = 3
L24           }else{
L25               tp = 5
L26        }
L27       Check type tp first, then other types, and type 4 last. If pair p is an interaction pair, add it to IP.
L28        }
L29      if (Num interaction pairs found) return IP
```

Figure 4-6. Algorithm to locate interaction pairs
1) The first stage

In algorithm FindInteractions, line L1 defines a candidate interaction pair that consists of two statements $s_u$ and $s_v$, the estimated AI measure for the pair, and the t-score. Lines L2 and L3 define a set P to store all candidate interaction pairs and a set IP to store all interaction pairs, respectively. Lines L4 and L5 iterate over all executed statements to form all possible candidate interaction pairs. In the example in Figure 4-2, there are 7 statements that are executed, so there are 49 ($7 \times 7$) candidate interaction pairs.

Line L6 of FindInteractions checks that a candidate interaction pair of statements satisfies a property that we call “dual-exposure positivity”. Hernan et al. [29] pointed out that positivity needs to be satisfied for joint exposures ($J, K$) with the four possible values (0, 0), (1, 0), (0, 1), and (1, 1) when identifying interactions between $J$ and $K$. Accordingly, we adapt the definition of conditional positivity in Section 2.5 to define a suitable new form of positivity. Given the condition that both $\text{pred}(s_1)$ and $\text{pred}(s_2)$ are covered ($C_1 = 1, C_2 = 1$), if there exist four different test cases covering both $s_1$ and $s_2$ ($X_1 = 1, X_2 = 1$), covering $s_1$ but not $s_2$ ($X_1 = 1, X_2 = 0$), covering $s_1$ but not $s_2$ ($X_1 = 0, X_2 = 1$), and covering neither $s_1$ and $s_2$ ($X_1 = 0, X_2 = 0$), we say that dual-exposure (DE) positivity holds for statements $s_1$ and $s_2$. That is, DE-positivity holds if both $\text{pred}(s_1)$ and $\text{pred}(s_2)$ are covered ($C_1 = 1, C_2 = 1$) and executions of $s_1$ and $s_2$ are independently inducible. Our experience suggests that if DE-positivity is violated, the linear regression model defined in Equation (4.5) cannot be fitted, because the least-squares regression algorithm will not converge.

Lines L7~L10 of FindInteractions construct the current candidate interaction pair, estimate its AI measure using the linear regression model defined in Equation (4.5) in
Section 4.3, and compute its $t$-score. Line L11 adds the pair to the set $P$. For the example in Figure 4-2, there are 6 pairs of statements satisfying DE-positivity, which are $<\text{Ln2, Ln4}>$, $<\text{Ln4, Ln6}>$, $<\text{Ln2, Ln6}>$, $<\text{Ln4, Ln2}>$, $<\text{Ln6, Ln4}>$ and $<\text{Ln6, Ln2}>$, and each of these 6 candidate interaction pairs has its corresponding AI measure and $t$-score. Figure 4-7 shows the former three candidate interaction pairs.

| Interaction Pair | $t$-score | AIM Value | $|\text{AIM}|$ | CET2 | CET3 | Masri’s |
|------------------|-----------|-----------|-------------|------|------|---------|
| $<\text{Ln2, Ln4}>$ | >1000 | -1 | 1 | false | true | 0 |
| $<\text{Ln4, Ln6}>$ | 0.577 | 0.5 | 0.5 | - | - | 0 |
| $<\text{Ln2, Ln6}>$ | $\approx 0$ | $\approx 0$ | - | - | 1 |

Figure 4-7. Interaction pair information

As with Baah et al.’s technique [9], we form a causal graph from the control dependence subgraph of the PDG. Figure 4-8 shows a causal graph with two exposure variables $X_1$ and $X_2$, corresponding confounders $C_1$ and $C_2$, and outcome variable $D$. Assume that $X_1$ and $X_2$ are coverage indicators for statements $s_1$ and $s_2$, respectively, and that $C_1$ and $C_2$ are coverage indicators for $\text{pred}(s_1)$ and $\text{pred}(s_2)$, respectively. Observe that there are no directed paths between $X_1$ and $X_2$, so neither variable is a cause of the other. As explained in Section 2.2, the back door paths $X_1 \leftarrow C_1 \rightarrow D$ and $X_2 \leftarrow C_2 \rightarrow D$ are blocked by $C_1$ and $C_2$, respectively. Thus, $C_1$ and $C_2$ satisfy the Back-Door Criterion relative to $X_1$, $X_2$, and $D$. It follows that by adjusting for $C_1$ and $C_2$ we can estimate the AI measure and $t$-score for the statement pair $<s_1, s_2>$ without confounding bias. The causal graph in Figure 4-8 can be used to illustrate the interaction pair $<\text{Ln2, Ln4}>$ in Figure 4-2. The variables $X_1$ and $X_2$ are coverage indicators for Ln2 and Ln4, respectively; $C_1$ and $C_2$

---

4 Note that in some additional cases other criteria for identifying causal effects [75] can be used to determine the variables to condition on.
are coverage indicators for $Ln1 = \text{pred}(Ln2)$ and $Ln3 = \text{pred}(Ln4)$, respectively; and $D$ indicates the test outcome.

![Causal Graph with Two Exposures](image)

Figure 4-8. A causal graph with two exposures.

In algorithm \textit{FindInteractions}, line L12 sorts the candidate interaction pairs in nonincreasing order of their suspiciousness scores, which again are the absolute values of the pairs’ AI measures or the absolute values of the pairs’ \textit{t}-scores. We can examine the candidate interaction pairs in the sorted order, but we still need to verify the interaction type of the examined pair in the second stage.

\textbf{2) The second stage}

In the second stage of \textit{FindInteractions}, each candidate interaction pair is processed in the sorted order to determine its interaction type, which may be 2, 3, 4, or 5. Again, our study focuses on interactions involving two statements, and only type 2, 3, 4, and 5 interactions can be used to characterize statement interactions based on the analysis in Section 4.5.1. VanderWeele \textit{et al.} [17][18] pointed out that the AI test and the CE test are
sufficient but not necessary for the causal interactions, i.e. if these conditions are satisfied then a causal interaction must present, but if the conditions are not satisfied, then there may or may not be a causal interaction. So, four types of interactions are checked for each suspicious candidate interaction pair. We use the combinations of AI measures and CET to distinguish each of these four types: (1) a type 2 interaction has a negative AI measure and satisfies its corresponding CET; (2) a type 3 interaction has a negative AI measure and satisfies its corresponding CET; (3) a type 4 interaction is the only interaction type that has positive AI measure among the four types; and (4) a type 5 interaction has a negative AI measure but no corresponding CET.

Line L13 starts to check each candidate interaction pairs in the set P in the sorted order. Line L14 checks if there is a SDG path that is realized by at least one test case from the first statement to the second statement in a candidate interaction pair before the pair is processed. If there is no such path, the pair is not processed because no information could possibly propagate from the first statement to the second one with the given test suite. (This check is done on the suspicious candidate interaction pairs in the second stage instead of the first stage because it is relatively expensive.) The three pairs <Ln2, Ln4>, <Ln4, Ln6> and <Ln2, Ln6> listed in Figure 4-7 have such paths from the first statement to the second statement, respectively. The rest other pairs <Ln4, Ln2>, <Ln6, Ln4> and <Ln6, Ln2>, which are not listed in Figure 4-7, have no such paths and will not be examined.

Line L15 of FindInteractions tests whether the AI measure value of the current pair is positive, which indicates that this pair is more likely to be a FRI pair than a FCI pair. If so, line L16 checks for a type 4 interaction first and then, if necessary, checks the three FCI types (types 2, 3 and 5) in random order. The type 2 and 3 CETs are not satisfied for the
candidate interaction pair that has a positive AI measure because the CE test involves a
stronger condition than the AI test. If the current pair is confirmed as an interaction pair,
the current pair is added to the set IP. If the AI measure of the current pair is negative,
indicating that the pair is more likely to be a FCI pair than a FRI pair, the FCI types 2, 3
and 5 are checked before the FRI type 4. The CET formulas for type 2 and 3 interactions
listed in Table 4-1 are used in lines L17~L21 to determine the most likely interaction type
of the pair. If CET2 returns true at line L17, the pair is most likely to be a type 2 interaction
pair, so line L18 recorded that a type 2 interaction should be checked for first. Otherwise,
if CET3 returns true at line L19 then the pair is most likely to be a type 3 interaction pair,
so line L20 records that a type 3 interaction should be checked for first. Otherwise, a type
5 interaction is most likely and so line L21 records that a type 5 interaction should be
checked for first. Line L22 checks the type of the interaction based on the results of lines
L17~L21, with type 4 being the last type that is checked. If the current pair is confirmed
as an interaction pair, it is added to the set IP. The variable Num indicates how many
interaction pairs the user wants to locate, which determines when the algorithm ends and
returns the set IP, which then contains all interaction pairs, at line L23. For the example in
Figure 4-2, pair <Ln2, Ln4> is examined first because of the high absolute of value of its
t-score, shown in the second column of Figure 4-7, when we choose the absolute values of
the t-scores as suspiciousness scores. Based on its AI value, CET2 outcome and CET3
outcome, the pair <Ln2, Ln4> should be checked first for a type 3 interaction, which is
confirmed. Observe that if n statements are executed by the given test set, the time
complexity of the algorithm FindInteractions is $O(n^2)$. 

94
4.6.2 Comparing Masri’s Approach to Our Approach

Masri [59] proposed a fault localization technique based on information flow coverage [62] to detect failures which are hard to identify because of interactions among program elements. For a pair of statements \(<s_1, s_2>\) such that there is an information flow from \(s_1\) to \(s_2\), a suspiciousness score is estimated by the metric

\[
\frac{\%fail(flow)}{\%fail(flow) + \%pass(flow)} = \frac{\%fail(flow)}{\%fail(flow) + \%pass(flow) + \%fail(flow)}
\]

\[\text{(4-6)}\]

where \(\%fail(flow)\) is the ratio of the number of failing tests that induce information flows from \(s_1\) to \(s_2\) to the total number of failing tests, and \(\%pass(flow)\) is the ratio of the number of passing tests that induce information flows from \(s_1\) to \(s_2\) to the total number of passing tests. The first term in the numerator of (4-6) is adapted from Tarantula [3] by replacing statement coverage with information flow coverage [59]. Assume there are dynamic information flows from \(s_1\) to \(s_2\) and a SDG path from \(s_1\) to \(s_2\) that is realized by at least one test case. We can express the metric (4-6) in probabilities with the outcome variable \(D\), which is a binary indicator of test outcome (1: fails, 0: passes) and two exposure variables \(X_1\) and \(X_2\), which are the coverage indicator of statements \(s_1\) and \(s_2\), respectively (1: covered, 0: not covered). Hence, metric (4-6) can be rewritten as

\[
\frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D = 1)}{\Pr(D = 1)} \cdot \frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D = 0)}{\Pr(D = 0)} + \frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D = 1)}{\Pr(D = 1)} \cdot \frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D = 0)}{\Pr(D = 0)}
\]

\[\text{(4-7)}\]
As mentioned by Baah et al. [9], if the total number of passing tests and the total number of failing tests are close \( \Pr(D = 0) \approx \Pr(D = 1) \), the Tarantula metric approximates precision, which is the probability that a program fails given that two exposures are covered. In the numerator of the second line of derivation (4-7), the first term satisfies

\[
\frac{\Pr(X_1 = 1, X_2 = 1|D=1)}{\Pr(X_1 = 1, X_2 = 1|D=1) + \Pr(X_1 = 1, X_2 = 1|D=0)}
\approx \frac{\Pr(X_1 = 1, X_2 = 1|D=1)}{\Pr(X_1 = 1, X_2 = 1|D=1) + \Pr(D=0)\Pr(X_1 = 1, X_2 = 1|D=0)}
\]

\[
= \frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D=1)}{\Pr(X_1 = 1 \cap X_2 = 1 \cap D=1) + \Pr(X_1 = 1 \cap X_2 = 1 \cap D=0)}
\]

\[
= \frac{\Pr(X_1 = 1 \cap X_2 = 1 \cap D=1)}{\Pr(X_1 = 1 \cap X_2 = 1)}
\]

\[
= \Pr(D = 1 | X_1 = 1, X_2 = 1)
\quad (4-8)
\]

Since the tests covering statements \( s_1 \) and \( s_2 \) must also cover both \( \text{pred}(s_1) \) and \( \text{pred}(s_2) \) in structured programs, the first term \( \Pr(D = 1 | X_1 = 1, X_2 = 1) \) in the numerator of the last line of derivation (4-8) is equal to \( \Pr(D = 1 | X_1 = 1, X_2 = 1, C_1 = 1, C_2 = 1) \), which is \( p_{11} \) according to Equation (4-2). Therefore, \( p_{11} \) is a term included in Masri’s metric.

While the \( t \)-score, which involves \( p_{11}, p_{10}, p_{01} \) and \( p_{00} \), considers four different kinds of tests that cover both \( s_1 \) and \( s_2 \), cover \( s_1 \) but not \( s_2 \), cover \( s_1 \) but not \( s_2 \), and cover neither \( s_1 \) nor \( s_2 \), Masri’s metric, which is shown in the last line of derivation (4-8), only considers the tests that cover both \( s_1 \) and \( s_2 \). Thus, Masri’s approach cannot handle FCIs well, which include type 2, 3, and 5 interactions. For example, given a set of tests, in which the tests
that covers \( s_1 \) and \( s_2 \) are passed \((D = 0)\) because FCIs involving \( s_1 \) and \( s_2 \) occur. The probability \( \Pr(D = 1|X_1 = 1, X_2 = 1) \), which approximates to \( p_{11} \) according to Equation (4-2) and the previous analysis, would be low because of such tests. The low probability \( p_{11} \) makes Masri’s scores for an interaction pair rather low. Let us take the program in Figure 4-2 as an example. The right-most column of Figure 4-7 shows Masri’s scores computed with metric (4-7). Masri’s approach misleadingly indicates that pair <Ln2, Ln6> should be examined prior to pair <Ln2, Ln4> which is an interaction pair.

Masri’s approach employs dynamic information flow (DIFA), which is useful for fault localization, however, its computation cost can be quite high [21][59]. Our approach checks static information flow only for more suspicious candidate interaction pairs, and can identify FCIs and FRIs while Masri’s approach cannot identify FCIs well.

### 4.7 Empirical Study

In this section, we report on an empirical study to assess our approach to localizing fault-revealing and fault-concealing interactions. It addresses one main research question (RQ): *How well does the approach identify pairs of statements involved in true FRIs and FCIs?*

The kind of FCI pair we detect consists of a faulty statement and another statement, which need not be faulty, that prevents an erroneous state from propagating to the outcome. We call the latter statement a “blocker.” The kind of FRI pair we detect consists of a faulty statement and another statement, which need not be faulty, such that a fatal state is only triggered only if both statements are covered by the same test. We call the latter statement an “enabler.”
4.7.1 Study Setup

We attempted to find examples of actual interaction pairs in the bug tracking systems of open-source projects that are suitable for use in evaluating our approach. However, this proved quite difficult. One reason is that bug reports typically focus on faults and their fixes but do not discuss interactions that either reveal or conceal the faults. Unsurprisingly, such interactions are simply not a concern of the developers making the reports. When we do find an interaction pair, we still need to obtain a set of test cases that induces the relevant interaction and is suitable for calculating meaningful values of the AI measure, etc. This task often requires detailed knowledge of the project.

We evaluate our approach to identifying fault-revealing and fault concealing interactions on some toy programs and three real world programs, NanoXML, Rome and Xerces2. We presently cannot find interaction pairs in the real world subject programs with the test cases available to us. So, to form an interaction pair we use a known faulty statement and randomly select another statement to “simulate” a blocker or enabler. To exclude pairs of statements that could not possibly be involved in an interaction, we require that there be a SDG path connecting the statements. Flag variables are used to simulate erroneous program states, and instructions are injected to manipulate them. We have different designs for FCI and FRI experiments:

- For a FCI experiment, a flag variable $flag$ is set after the fault and the blocker. For each test case, the program state becomes erroneous ($flag = 1$) whenever the fault is covered. The erroneous state is corrected ($flag = 0$) with different probabilities whenever the blocker is covered. A test fails if $flag = 1$ is observed at the output.
For a FRI experiment, two flag variables $flag1$ and $flag2$ are initialized to 0 and injected after the fault and the enabler, respectively. For each test case, $flag1$ is set to 1 if the fault is covered, and $flag2$ is set to 1 with different probabilities whenever the enabler is covered. A test fails is considered to fail only if we observe that $flag1 = 1$ and $flag2 = 1$ at the output.

Our study platform is based on the ASM Java bytecode manipulation framework [63], the JavaPDG dependence analysis tool [61] and the statistical package R [64] (version 3.0.2). We first instrument a Java subject program’s class files using ASM, to record statement coverage information at runtime. Then, JavaPDG is used to generate the PDG of the subject program. Each PDG node is mapped back to a source code statement. For each statement $s$, the PDG records the statement ID of $pred(s)$ if the latter exists. Once the statement coverage information is collected, R is used to calculate the $t$-score for each interaction pair.

We selected three real world subject programs written in Java: NanoXML, Rome, and Xerces2. NanoXML [65] (version 1) is an XML parser from Software-artifact Infrastructure Repository (SIR) [66]. Rome (revision 840) is an open source library for parsing, generating, and publishing RSS and Atom feeds [67]. Apache Xerces2 (v.2.9.1) is a processor for parsing, validating, serializing, and manipulating XML [68]. We created some toy programs to illustrate and understand the operation of different types of interactions on statement level. For the toy programs, we produce an interaction between statements for each program by using (1) logical operators (an example is shown in Figure 4-3), (2) modulo operators (an example is shown in Figure 4-5), and (3) multiplication by 0 (an example is shown in Figure 4-2).
NanoXML comes with test suites. To test Xerces2, XML files were collected from the system directories of an Ubuntu Linux 7.04 machine. For Rome, test cases were obtained by downloading Atom and RSS files with a custom web crawler. The test cases for toy programs were randomly generated.

A sample of faults was selected from the repository for each subject program: 1 for NanoXML, 4 for Rome, and 3 for Xerces2. Each fault generates a buggy program. For each buggy program, we randomly selected five statements as blockers and five statements as enablers and generated ten versions of the buggy program. A summary of our subject programs and tests is shown in Table 4-2.

<table>
<thead>
<tr>
<th></th>
<th>Avg. LOC</th>
<th># of Tests</th>
<th>Total versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>FCI</td>
</tr>
<tr>
<td>Toy progs.</td>
<td>0.14K</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>NanoXML</td>
<td>4.4K</td>
<td>1000</td>
<td>5</td>
</tr>
<tr>
<td>Rome</td>
<td>24K</td>
<td>2000</td>
<td>20</td>
</tr>
<tr>
<td>Xerces2</td>
<td>167K</td>
<td>1536</td>
<td>15</td>
</tr>
</tbody>
</table>

The measure $Cost()$ was used to evaluate the effectiveness of locating an interaction pair in RQ. It is the percentage of all candidate interaction pairs that a developer needs to examine to identify the first interaction pair. To deal with the issue that multiple candidate interaction pairs and the interaction pair may receive the same $t$-scores, we adapt cost measure (3-22) [69]. Given a set $P$ of candidate interaction pairs, $Cost()$ is defined by

$$Cost(p) = | \{ p | sc(p) > sc(p^*) \} | + \frac{| \{ p | sc(p) = sc(p^*) \} | + 1}{2}$$

where $p \in P$ is any pair, $p^*$ is the first interaction pair, $sc(p)$ is the absolute of $t$-score of $p$, and $|...|$ denotes the size of a set. We assumed that if developers examined an interaction
pair, the interaction pair would be found. Here, the cost we estimate is focused on how many candidate interaction pairs we need to examine.

4.7.2 Results

In our experiment, the input Num to FindInteractions is set to 1. To evaluate how efficient our approach is for locating the interaction pairs in programs, we computed the percentage costs for locating fault-concealing interactions or fault-revealing interactions on toy programs and real-world programs with different types of suspiciousness scores. We use the absolute values of the AI measures as suspiciousness scores for Approach1, and use the absolute values of the $t$-scores as suspiciousness scores for Approach2. In our simulation of blocking of erroneous states, the erroneous state triggered by covering the fault is blocked, in different experimental replications, with different probabilities (1.0, 0.8, 0.5, and 0.1) when a blocker is executed. We call these probabilities blocking probabilities. For example, we know that Ln4 in Figure 4-2 overwrites all erroneous states with zero when Ln4 is covered, so the blocking probability of statement Ln4 is 1.0. In our simulation of FRIs, the erroneous state triggered by covering the fault is made fatal with different probabilities (1.0, 0.8, 0.5, and 0.1) when an enabler is executed. We call these probabilities enabling probabilities.

First, we evaluate Approach1, which uses the absolute values of AI measures as suspiciousness scores, on toy and real world programs. Table 4-3 shows the average percentage costs, for different blocking probabilities, of locating FCIs, as a percentage of the total number of candidate interaction pairs, in 44 total versions of the four subject programs. Generally, the average costs increase as the blocking probabilities decrease. For
the toy programs, the average costs were each 6.03% when blocking probabilities were 1.0, 0.8, and 0.5, and the average cost was 9.48% when the blocking probability was 0.1. For the real-world programs, the corresponding average costs were 2.09%, 2.44%, 4.68%, and 14.77%, respectively. The rightmost column shows the average number of candidate interaction pairs per program that satisfy \( DE \)-positivity.

Table 4-3. Average Percentage Cost for Locating FCIs (|AIM|)

<table>
<thead>
<tr>
<th>Blocking prob.</th>
<th>Toy (4 ver.) Avg.</th>
<th>1.0</th>
<th>0.8</th>
<th>0.5</th>
<th>0.1</th>
<th>pairs #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy (4 ver.) Avg.</td>
<td>6.03%</td>
<td>6.03%</td>
<td>6.03%</td>
<td>9.48%</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>Real world NanoXML (5 ver.)</td>
<td>1.80%</td>
<td>2.39%</td>
<td>5.25%</td>
<td>14.00%</td>
<td>5.7K</td>
<td></td>
</tr>
<tr>
<td>Rome (20 ver.)</td>
<td>3.46%</td>
<td>3.63%</td>
<td>5.24%</td>
<td>11.25%</td>
<td>118K</td>
<td></td>
</tr>
<tr>
<td>Xerces2 (15 ver.)</td>
<td>0.35%</td>
<td>0.87%</td>
<td>3.74%</td>
<td>19.72%</td>
<td>204K</td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>2.09%</td>
<td>2.44%</td>
<td>4.68%</td>
<td>14.77%</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>

Then, we computed the average percentage costs for locating fault-revealing interactions in toy programs and real world programs. Table 4-4 shows average percentage costs, for different enabling probabilities, of locating FRIs, as a percentage of the total number of candidate interaction pairs, in 42 total versions of the toy programs and the three real-world programs with different enabling probabilities. Generally, the average costs increase as the enabling probabilities decrease. For the toy programs, the average costs were each 6.03% when blocking probabilities were 1.0, 0.8, and 0.5, and the average cost was 16.38% when the blocking probability was 0.1. For the real-world programs, the corresponding average costs were 1.25%, 1.30%, 2.26%, and 5.31%, respectively. The right-most column shows the average number of candidate interaction pairs that satisfy \( DE \) positivity in each program.
Table 4-4. Average Percentage Cost for Locating FRIs (|AIM|)

<table>
<thead>
<tr>
<th>Enabling prob.</th>
<th>Toy (2 ver.) Avg.</th>
<th>1.0</th>
<th>0.8</th>
<th>0.5</th>
<th>0.1</th>
<th>pairs #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real world</td>
<td>NanoXML (5 ver.)</td>
<td>1.26%</td>
<td>1.34%</td>
<td>2.38%</td>
<td>4.15%</td>
<td>5.7K</td>
</tr>
<tr>
<td></td>
<td>Rome (20 ver.)</td>
<td>1.83%</td>
<td>1.87%</td>
<td>1.92%</td>
<td>3.49%</td>
<td>118K</td>
</tr>
<tr>
<td></td>
<td>Xerces2 (15 ver.)</td>
<td>0.48%</td>
<td>0.53%</td>
<td>2.67%</td>
<td>8.14%</td>
<td>204K</td>
</tr>
<tr>
<td></td>
<td>Avg.</td>
<td>1.25%</td>
<td>1.30%</td>
<td>2.26%</td>
<td>5.31%</td>
<td>--</td>
</tr>
</tbody>
</table>

Next we evaluate Approach2, which uses the absolute values of t-scores as suspiciousness scores, on toy and real world programs. Table 4-5 shows the average percentage costs, for different blocking probabilities, of locating FCIs, as a percentage of the total number of candidate interaction pairs, in 44 total versions of the four subject programs. Generally, the average costs increase as the blocking probabilities decrease. For the toy programs, the average costs were each 6.03% when blocking probabilities were 1.0, 0.8, and 0.5, and the average cost was 9.48% when the blocking probability was 0.1. For the real-world programs, the corresponding average costs were 1.88%, 2.04%, 3.52%, and 9.80%, respectively. The rightmost column shows the average number of candidate interaction pairs per program that satisfy *DE-positivity*.

Table 4-5. Average Percentage Cost for Locating FCIs (|t-score|)

<table>
<thead>
<tr>
<th>Blocking prob.</th>
<th>Toy (4 ver.) Avg.</th>
<th>1.0</th>
<th>0.8</th>
<th>0.5</th>
<th>0.1</th>
<th>pairs #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real world</td>
<td>NanoXML (5 ver.)</td>
<td>0.26%</td>
<td>0.47%</td>
<td>2.55%</td>
<td>11.10%</td>
<td>5.7K</td>
</tr>
<tr>
<td></td>
<td>Rome (20 ver.)</td>
<td>3.42%</td>
<td>3.54%</td>
<td>5.28%</td>
<td>9.08%</td>
<td>118K</td>
</tr>
<tr>
<td></td>
<td>Xerces2 (15 ver.)</td>
<td>0.36%</td>
<td>0.56%</td>
<td>1.49%</td>
<td>10.34%</td>
<td>204K</td>
</tr>
<tr>
<td></td>
<td>Avg.</td>
<td>1.88%</td>
<td>2.04%</td>
<td>3.52%</td>
<td>9.80%</td>
<td>--</td>
</tr>
</tbody>
</table>

Then, we computed the average percentage costs for locating fault-revealing interactions in toy programs and real world programs. Table 4-6 shows average percentage costs, for different enabling probabilities, of locating FRIs, as a percentage of the total
number of candidate interaction pairs, in 42 total versions of the toy programs and the three real-world programs with different enabling probabilities. Generally, the average costs increase as the enabling probabilities decrease. For the toy programs, the average costs were each 6.03% when blocking probabilities were 1.0, 0.8, and 0.5, and the average cost was 16.38% when the blocking probability was 0.1. For the real-world programs, the corresponding average costs were 0.47%, 0.77%, 1.28%, and 2.67%, respectively. The right-most column shows the average number of candidate interaction pairs that satisfy \textit{DE positivity} in each program.

<table>
<thead>
<tr>
<th>Enabling prob.</th>
<th>1.0</th>
<th>0.8</th>
<th>0.5</th>
<th>0.1</th>
<th>pairs #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy (2 ver.)  Avg.</td>
<td>6.03%</td>
<td>6.03%</td>
<td>6.03%</td>
<td>16.38%</td>
<td>58</td>
</tr>
<tr>
<td>nanoXML (5 ver.)</td>
<td>0.52%</td>
<td>0.58%</td>
<td>2.70%</td>
<td>10.74%</td>
<td>5.7K</td>
</tr>
<tr>
<td>Rome (20 ver.)</td>
<td>0.50%</td>
<td>0.82%</td>
<td>1.01%</td>
<td>1.73%</td>
<td>118K</td>
</tr>
<tr>
<td>Xerces2 (15 ver.)</td>
<td>0.41%</td>
<td>0.76%</td>
<td>1.15%</td>
<td>1.23%</td>
<td>204K</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.47%</td>
<td>0.77%</td>
<td>1.28%</td>
<td>2.67%</td>
<td>--</td>
</tr>
</tbody>
</table>

The results of our study indicate that statements involved in fault-revealing and fault-concealing interactions can be located by using our proposed approaches, Approach1 and Approach2. However, Approach2 was less costly than Approach1 on locating both fault-revealing and fault-concealing interactions in programs, which indicates that the candidate interaction pairs with larger standard error should be less suspicious than the candidate interaction pairs with smaller standard error.

When the probability of such interactions occurring is low, however, the efficiency of our approach is impaired, because the suspiciousness scores for the corresponding interaction pairs are then also low. In fact, our study indicates that this is true with fairly large programs, for which the cost increases when the probability that statement interactions
occur decreases. However, for some small programs, like our toy programs, although the interaction pairs have lower suspiciousness scores when the probability that statement interactions occur decreases, the cost to locate an interaction pair does not increase. This is because these values are still relatively high among suspiciousness scores for all candidate interaction pairs.

4.7.3 Study Limitations

Our proposed approach uses the coverages of statements and test outcomes to generate AI measures and t-scores for each candidate interaction pair, and which are examined by developers in the nonincreasing order of the absolute values of t-scores. These steps are similar to those of typical CBSFL techniques. Consequently, the results and conclusions of our empirical study are subject to the usual threats to validity for CBSFL studies [4], such as possible peculiarities of the test suites and fault masking if there are multiple faults in programs. Our results should not be considered conclusive unless they are confirmed in a number of additional studies collectively involving many diverse and realistic subject programs, faults, and test sets. While our approach limits confounding bias due to backdoor paths comprised of control dependences, it is desirable to also block backdoor paths involving data dependences and causal variables that represent the state of program variables [60].

Another issue is that, due to the difficulty of collecting, for use in empirical evaluation, both sufficient examples of interaction pairs in real-world programs and suitable tests for detecting them, we had to simulate FRIs and FCIs in our larger subject programs. We attempted to make this simulation realistic, by forming interaction pairs for our larger
subject programs that each involve an actual faulty statement paired with a statements connected to it by a SDG path that is realized by at least one test case.

4.7.4 Discussion

Although the empirical evaluation of our proposed approach suggests that it is useful for locating fault-revealing and fault-concealing interactions, there is reason to believe that the power of using coverage information alone for locating statements involved in FRIs and FCIs is fairly limited. For one thing, it cannot detect rare, conditional interactions involving statements that are commonly executed together.

4.8 Related Work

Voas [70] described the Propagation-Infection-Execution (PIE) technique, which is intended to identify locations in a program where faults, if they exist, are more likely to remain undetected during testing. PIE is based on the assumption that for a software failure to be observed, the infected erroneous program state caused by executing a fault must propagate to the output. Richardson and Thompson [71] also proposed a model, called RELAY, to describe how a fault causes a failure. The RELAY model, which includes origination, computational transfer, and data and control dependence transfer, is used to evaluate the fault detection capabilities of test data selection criteria.

Debroy and Wong [19] discussed constructive and destructive interference among faults and then classified these interferences into six kinds of interference. Their empirical studies on the Siemens suite of (small) programs indicated that the interference did occur and that the simultaneous occurrence of constructive and destructive interference is the
most common kind of interference. Hence, the assumption of fault-independence assumption should not hold in multiple-fault programs. DiGiuseppe and Jones [20] called the aforementioned kinds of interference fault interactions and they classified them into four types, which are described in Section 4.5.3. They conducted a more comprehensive empirical study, involving larger software systems, that addressed these four types of interaction. Their results indicated that fault obfuscation (FI2) was the most prevalent type and that multi-type interaction (FI4) was rare in the larger programs. This observation contradicts Debroy et al.’s observation that the multi-type interactions are the most common in programs. Groce et al. [72] used large-scale random testing to statistically characterize the significance of fault suppression in multiple-fault programs. They showed that suppression is common in C compiler, a flash file system, and JavaScript engines. Xue and Namin [73] statistically measured the influence and significance of fault interactions and showed that fault interaction degrades the effectiveness of coverage-based statistical fault-localization in multiple-fault programs.

Masri [59] proposed a statistical fault localization technique based on analyzing dynamic information flows (dynamic slice paths) [62] between statements. In this technique, interactions between statements are considered and the faulty statement is located by examining suspicious pairs of statements connected by dynamic information flows. Our approach locates interaction pairs by examining suspicious pairs of statements connected by an SDG path that is realized by at least one test case from the first statement to the second statement. Masri’s technique employs a non-causal (confounded) suspiciousness for flows that is adapted from Tarantula [3]. Our approach employs a metric
that is based on causal interactions tests, and it requires independent inducibility of coverage events, which avoids computation of meaningless scores.

Baah et al. [9] proposed statistical regression adjustment for coverage of the DFCF predecessor of a statement \( s \) to limit confounding bias when estimating the failure-causing effect of covering \( s \), and they presented empirical evidence of its effectiveness. Later, Baah et al. [10] used a matching method to control for coverage of both data and control dependences predecessors in order to limit confounding bias in SFL. Shu et al. [40] proposed a regression-based technique for CSFL at the method (subprogram) level, and they addressed an important precondition for causal inference called positivity in this context. Bai et al. [28] analyzed the regression based CSFL technique proposed by Baah et al. [9] algebraically and probabilistically and proposed improvements to it. They also examined the issue of positivity in detail. Bai et al. [60] proposed a value-based CSFL technique for numerical programs, called NUMFL, to control for confounding bias passed through variable values and to estimate the causal effects of individual numerical expressions on failures.

Cohen et al. [43] mentioned that system faults are often caused by unexpected interactions among components and proposed ways to use combinatorial objects to facilitate testing interactions among software components. Ghandehari et al. [74] pointed out that \( t \)-way combinatorial testing is used to detect failures that are triggered by combinations involving no more than \( t \) parameters. They proposed an approach to identify the combinations of multiple parameters that cause some tests to fail. First, a set of suspicious combinations are identified and ranked. Then, new tests are generated, and these new tests are executed to refine the ranking of suspicious combinations in the next
iteration. The process is repeated until certain conditions are satisfied. While combinatorial interaction testing, used as a black-box technique, focuses on detecting faults which are caused by parameter interaction, our approach, used as a white-box technique, provides a new way to localize faulty statements that cause fault-revealing and fault-concealing interactions.

4.9 Conclusion

This chapter has formally characterized fault-revealing and fault-concealing interactions that occur during program execution, and it has shown how these definitions are related to characterizations of causal interactions appearing in the causal inference literature. It has also related the definitions of FRIs and FCIs to characterizations of fault-interactions [19][20] and failed error propagation [21] by software researchers. A preliminary approach, based on causal interaction tests, for locating statements involved in both FRIs and FCIs was proposed. Empirical results were presented that suggest the approach may be useful.
5 Conclusion and Future Work

This dissertation focuses mainly on coverage-based statistical fault localization and examines two issues. First, the common structure of the most effective metrics suggested in different studies is examined, as well as how this commonality helps developers to better understand the strengths and weaknesses of some CBSFL metrics. Second, we examine the impact of program statement interactions which can cause faults to be concealed or revealed from observation at the outcome.

For the first issue, we algebraically and probabilistically analyze several CBSFL metrics that have performed well in recent comparisons of many metrics in order to identify their common properties. We also introduce several new metrics relative Ochiai, relative F1, enhanced Tarantula, risk Ochiai, risk F1, risk recall, and enhanced Tarantula2. The former three metrics are in the form $w \times RP$, and the latter four metrics are in the form $w \times ARD$. All of these metrics exhibit good performance over precision in our empirical study. The results of our empirical studies suggest that the most effective metrics contain a product of two terms: one that estimates the failure-causing effect of a program element (possibly with confounding bias) and one that weights the first term based on the evidence for the existence of faults in other program elements. Baah et al. [9] showed that plugging in the causal effect estimate obtained with their statistical-regression based technique in place of the precision terms in Ochiai and the F1 measure improved their performance substantially. We expect this would also occur with the other metrics considered in Chapter 3. We hope to confirm this in a future study.
To clarify the second issue, we present event-based definitions of fault-revealing and fault-concealing interactions in programs. Then, we relate them to the theory of causal interactions and to previous characterizations of fault-interactions and failed error propagation. A preliminary approach for locating statements involved in fault-revealing and fault-concealing interactions is proposed, based on the statistical tests developed to detect causal interactions. We evaluate our proposed approach on both toy programs and real world programs. The results show that statements involved in fault-revealing and fault-concealing interactions can be identified in the programs. As mentioned in Section 4.7.3, our proposed approach cannot detect rare, conditional interactions involving statements that are commonly executed together. To address this limitation, in future work we intend to explore the application of causal interaction tests to data collected about the values of program variables to detect fault-revealing and fault-concealing interactions.
REFERENCE


10.


