STATISTICAL CAUSAL ANALYSIS FOR FAULT LOCALIZATION

A Dissertation
Presented to
The Academic Faculty

by

George Kofi Baah

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2012

Copyright © 2012 by George Kofi Baah

STATISTICAL CAUSAL ANALYSIS FOR FAULT LOCALIZATION

Approved by:

Dr. Mary Jean Harrold, Advisor
College of Computing
Georgia Institute of Technology

Dr. Andy Podgurski
Electrical Engineering & Computer Science Department
Case Western Reserve University

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Dr. Mayur Naik
College of Computing
Georgia Institute of Technology

Dr. Alexander Gray
College of Computing
Georgia Institute of Technology

Date Approved: 29th June 2012

To God be the glory.

ACKNOWLEDGEMENTS

First, I would like to thank God for giving me the strength to endure this journey and also for putting people in my life to advise me during the journey.

I would like to thank my advisor, Dr. Mary Jean Harrold for her support during my Ph.D. She gave me the opportunity to work on problems that I found interesting. Her hands-on or hands-off approach management style helped me immensely in developing into a competent researcher. I would like to thank my committee members: Dr. Alessandro Orso, Dr. Alexander Gray, Dr. Mayur Naik, and Dr. Andy Podgurski. Thank you very much for your insightful comments and advise on my research topic. Additionally, I would like to thank Andy Podgurski for the numerous discussions we had on causal analysis, which is as significant part of the foundation upon which my current research is built.

I would like to thank my family (Mom, Dad, Eric, Samuel, and Peter) without whose prayers and encouragement this dissertation would not have been possible. I would like to thank my wife, Christina, for her prayers, incredible patience, encouragement, and unrelenting support. Her sacrifices that allowed me to achieve this goal were great, and I am very grateful. Thank you family, for all your support.

I would like to thank my undergraduate professors at Pace University: Dr. Carol E. Wolf and Dr. Joseph Bergin. Dr. Wolf was my first computer programming professor and through her exceptional teaching I was inspired to study Computer Science. I would like to thank Dr. Bergin who was instrumental in my admission to Georgia Tech. and also for giving me the confidence necessary to pursue a doctorate degree. Thank you Dr. Wolf and Dr. Bergin for encouraging me to go to graduate school.

I would like to thank members of the Aristotle Research Lab and all my friends at Georgia Tech. both past and present for their support. I would also like to thank Saswat Anand and his family for the intellectual discussions we had and the delicious food we shared together. I would like to thank Dr. Eli Tilevich (Virginia Tech.) for his constant encouragement, insightful comments, and New-York jokes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The ubiquitous nature of software demands that software is released without faults. However, software developers inadvertently introduce faults into software during development. To remove the faults in software, one of the tasks developers perform is debugging. However, debugging is a difficult, tedious, and time-consuming process. Several semi-automated techniques have been developed to reduce the burden on the developer during debugging. These techniques consist of experimental, statistical, and program-structure based techniques. Most of the debugging techniques address the part of the debugging process that relates to finding the location of the fault, which is referred to as fault localization. The current fault-localization techniques have several limitations. Some of the limitations of the techniques include (1) problems with program semantics, (2) the requirement for automated oracles, which in practice are difficult if not impossible to develop, and (3) the lack of theoretical basis for addressing the fault-localization problem.

The thesis of this dissertation is that statistical causal analysis combined with program analysis is a feasible and effective approach to finding the causes of software failures. The overall goal of this research is to significantly extend the state of the art in fault localization. To extend the state-of-the-art, a novel probabilistic model that combines program-analysis information with statistical information in a principled manner is developed. The model known as the probabilistic program dependence graph (PPDG) is applied to the fault-localization problem. The insights gained from applying the PPDG to fault localization fuels the development of a novel theoretical framework for fault localization based on established causal inference methodology.

The development of the framework enables current statistical fault-localization metrics to be analyzed from a causal perspective. The analysis of the metrics show that the metrics are related to each other thereby allowing the unification of the metrics. Also, the analysis of metrics from a causal perspective reveal that the current statistical techniques do not find the causes of program failures instead the techniques find the program elements most associated with failures. However, the fault-localization problem is a causal problem and statistical association does not imply causation. Several empirical studies are conducted on several software subjects and the results (1) confirm our analytical results, (2) demonstrate the efficacy of our causal technique for fault localization. The results demonstrate the research in this dissertation significantly improves on the state-of-the-art in fault localization.

CHAPTER 1

INTRODUCTION

1.1 Motivation

The pervasiveness and increase in complexity of software in our lives requires that software developers engineer high-quality software. However, software development is a human process and developers inadvertently introduce faults into software. According to the National Institute of Standards and Technology (NIST), software faults costs the U.S. economy an estimated 59.5 billion dollars a year [66]. The process by which faults are located and fixed when they cause failures is called *debugging*, and it is one of the ways of improving the quality of software. However, debugging is a difficult and time consuming process that is often performed manually. One of the important activities of debugging is *fault localization*, which is the task of finding the locations of causes of the software failures. Because fault localization is a laborious and time-consuming task, automated techniques have been developed with the goal of reducing the burden on developers during debugging. Such automatic fault-localization techniques will lead to quicker bug fixes and higher software quality.

1.2 Existing Fault-Localization Techniques

In recent years, a considerable number of fault-localization techniques have been developed. These techniques include statistical and machine-learning techniques (e.g., [1, 14, 15, 37, 38, 42, 43, 70]), slicing techniques (e.g., [2, 18, 28, 69, 74]), and state-altering techniques (e.g., [16, 35, 72, 73]). Figure 1 shows the time line of the development of the techniques. The horizontal axis shows approximately the year the technique was developed and the vertical axis shows the level of improvement of each

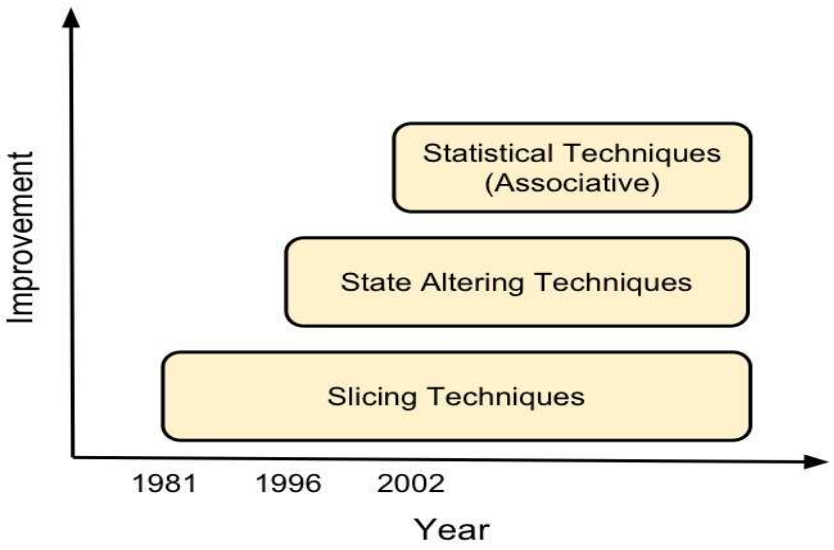


Figure 1: Time line of major fault localization approaches.

approach. Research in several published results indicate that statistical techniques in many cases perform better than slicing and state-altering techniques in terms of the amount of code the developer must examine to find the fault. However, research is still ongoing in each of the areas.

1.2.1 Statistical and Machine Learning Techniques

To locate the fault, statistical fault-localization techniques require execution data usually consisting of information about both passing and failing executions. The execution data typically contain the coverage of program entities and the outcome (success or failure) of the program. To determine which program entity is responsible for a given failure, the techniques assign a suspiciousness score to each program entity; this score is a quantity that measures the likelihood that a particular entity is responsible for the failure. The techniques rank the program entities based on the suspiciousness scores. The program entities are ordered in decreasing order of suspiciousness scores for the developer to examine because the assumption is that faulty program entities will have higher scores than non-faulty program entities.

Jones, Stasko, and Harrold developed a fault-localization technique called Tarantula, and Jones and Harrold [37] compared it to other fault-localization techniques that existed at the time. Tarantula consists of both a statistical component and a visualization component. The statistical component consists of a metric that Tarantula uses to rank statements in a program. The intuition is that statements executed primarily by failing test cases are more likely to be faulty than statements executed primarily by passing test cases. Tarantula also provides a visualization component that presents the results of the statistical component graphically to the developer using different color schemes. Abreu, Zoeteweij, and van Gemund [1] applied the Ochiai metric to the fault-localization problem; they also conducted a number of studies that compare different statistical fault-localization metrics. Their results show that the Ochiai metric performs better than the Tarantula metric in certain cases. Lucia and colleagues [44] perform an extensive set of empirical studies that compare many statistical fault-localization metrics. Lucia and colleagues use the Tarantula metric and the Ochiai metric as the baselines in their studies. They found that the Tarantula and Ochiai metrics perform almost the same. The also found that the other statistical metrics did not perform as well as the Tarantula or Ochiai metrics. Liblit and colleagues [42] introduced the Cooperative Bug Isolation (CBI) approach to debugging. The goal of the technique is to gather dynamic information from programs executing in the field for debugging. The metric used by the CBI approach places an emphasis on the association between a predicate being true and program failure. Liu and colleagues [43] also introduced a technique called SOBER, which addresses a limitation of CBI by analyzing whether a predicate is true or false. The intuition behind Liu and colleagues' technique is that there is an underlying statistical distribution governing the correct and incorrect executions and that the greater the divergence between the correct and incorrect statistical distributions with respect to a predicate the more relevant the predicate is to the fault. Wong and colleagues [70] present a statistical

technique that is based on the assumption that different test cases contribute differently if computing the suspiciousness of a program entity. Wong and colleagues assign different weights to different sets of test cases and then estimate the suspiciousness of statements in the program.

Jiang and Su [36] contend that the statistical techniques that use predicates as program entities may be ineffective when the fault is not associated with a predicate. Jiang and Su present a machine-learning technique that uses feature selection, clustering, and branch prediction to construct faulty control-flow paths that contain enough contextual information to help the developer understand and locate the faulty program entity. Cheng and colleagues [15] present a fault-localization technique based on using discriminative graph mining to extract the top subgraphs that may contain the fault. The intuition behind their work is that fault-localization techniques that provide the developer with a ranked list of program entities do not provide the developer with enough contextual information. Burnell and Horvitz [13] use belief networks (Bayesian networks [49]) for finding the location of faults in mainframe assembler programs. Their technique constructs belief networks using information obtained from experts in their targeted domain. A limitation specific to their technique is that the information used to construct the belief networks can be expensive to obtain and fraught with inaccuracies. Naish, Lee, and Ramamohanarao [48] present a model-based analysis of many statistical metrics and find that many of the metrics are similar in performance. They also provide optimal ranking metrics that can be used for fault localization regardless of the number of test cases.

In general, all these statistical fault-localization techniques have several limitations. First, the techniques ignore the semantic relationships between program entities that are potentially induced by syntactic dependences [56]. For example, most of the techniques ignore variable definitions and where they are used in programs. Second, although the goal of statistical techniques is to find the cause of program failures the

techniques are actually estimating the program entity most correlated with failure. However, statistical correlation does not imply causation [31, 68]. Third, the statistical techniques do not provide any theoretical basis for fault localization. Therefore it is difficult to evaluate the efficacy of the techniques and which technique to use for fault localization.

1.2.2 State-Altering Techniques

State-altering techniques alter the states of an executing program to attempt to isolate the location of the faulty program entity. Zeller [72] developed the delta debugging technique, which requires one passing and one failing execution that are identical in terms of the program paths. The intuition behind this technique is that any difference between the two identical executions is probably the cause of the failure. The goal of delta debugging is to isolate the differences between passing and failing executions by swapping states between the passing and failing execution until the smallest state difference that causes the program to fail is found. This small state difference (δ) is the cause of the failure.

Zhang, Gupta, and Gupta [73] developed a variant of delta debugging called predicate switching that also alters the states of predicates during program execution. Given a failing execution, the goal of predicate switching is to find the predicate that if switched (i.e., from false to true or true to false) causes the program to execute successfully. Zhang, Gupta, and Gupta call such predicates *critical predicates*. The technique computes a bidirectional slice from the critical predicate to find the faulty statement. The intuition behind the technique is that the program entity that caused the failure will be in the bidirectional slice. The limitation of predicate switching is that the slice can contain many program entities. Moreover, unlike the statistical techniques, predicate switching does not provide a ranking of the program entities. Jeffrey, Gupta, and Gupta [35] introduce another variant of delta debugging, value

replacement, that alters the states of variable values at program statements. Their technique subsumes the predicate-switching technique. Given a failing execution, the goal of the technique is to systematically alter the variable values at statement instances in the execution one at a time, and determine whether the output of the execution changes from failure to success. If there is an output change from failure to success then an Interesting value mapping Pair (IVMP) has been found. Unlike the predicate-switching technique, value replacement provides a ranking of program entities and presents it to the developer. Burger and Zeller [12] combine delta debugging, record and replay of failing executions, and slicing to localize the fault statements.

The first limitation of state-altering techniques is that they do not deal with the problem of semantic consistency because by altering a program state, the techniques do not guarantee that the new execution is an actual execution. Here, actual execution means that there is an input that can produce that execution. For example, if a memory location in a passing execution is replaced with one from a failing execution, there is no guarantee that the memory location will be valid in the context of the passing execution. Delta debugging [72] attempts to deal with the semantic-consistency problem by using memory graphs. A memory graph is a program state, which consists of all values and variables in a program but also represents operations such as pointer dereferencing. None of the other techniques handle this problem. The second limitation of state-altering techniques is that they require the execution of the program each time a state is altered. However, repeated executions of the program can be expensive. The third limitation of state-altering techniques is that they require the presence of an oracle (ideally automated) that is queried to determine the outcome of the executing program every time a state is altered and the program is executed. In practice, it is difficult to develop automated oracles and the absence of oracles can increase the burden on the developer during fault localization if the developer must act as an oracle.

1.2.3 Slicing Techniques

The goal of program slicing is to find the set of program entities that potentially affects the computation at a given program point (slice criterion). There are two kinds of slicing techniques: static program slicing and dynamic program slicing. In static slicing, the slice set contains program entities that may affect the computation at a given program point whereas in dynamic slicing, the slice set contains program entities that actually affect the computation at a given program point.

Weiser [69] defined program slicing and applied it to debugging. The drawback of Weiser's technique is that the slice set often contains many program entities (in some cases the whole program). The developer is thus, forced to examine many program entities. Agrawal's thesis [2] presents a number of dynamic-slicing algorithms for fault localization. Gyimóthy, Beszédes, and Forgács present relevant slicing [28], an extension of dynamic slicing. Relevant slicing extends dynamic slicing by adding to the slice computed by a dynamic slicing program those entities that may have affected the slice criterion had those program entities been evaluated differently. Zhang, Gupta, and Gupta [74] present a technique that uses a threshold to prune the backward slice computed for a particular program entity. By pruning the backward slice, their technique can reduce the size of the computed slice set.

The first limitation of the above slicing techniques is that they do not account for the strength of the dependences between program entities and how likely each program entity is the cause of the failure. The second limitation of these techniques is that the slice sets can sometimes be very large. The third limitation is that the techniques do not provide the developer with information on how to start searching for the fault. The fourth limitation is that the techniques only compute program entities that are associated with failures instead of finding program entities that caused the failure.

DeMillo, Pan, and Spafford present a slicing technique called critical slicing [18]

and apply it to fault localization. Critical slicing combines the statement deletion mutation operator technique with Executed Static Program Slicing (ESPS) to produce a slice set containing fewer program entities. The slice set computed by ESPS is the set of statements in a static slice that are executed when the program is executed with a given test case. To determine the impact of a statement s in the ESPS slice set, their technique executes the program with s deleted from the program (using the statement-deletion-mutation-operator technique). If the statement does not have an impact on the failure of the program, it is removed from the slice set. The technique inherits the limitations of the slicing and state-altering techniques.

1.2.4 Other Fault-Localization Techniques

Static-analysis based fault-localization techniques (e.g., [20, 21, 24, 32]) have been developed. These technique rely on specifications of potential faults to find potentially faulty program entities. Engler and colleagues [20] present a technique that enables the incorporation of program rules and checkers into a compiler. The checkers are used by the compiler to determine whether the program violates the rules. An example of a rule is "never to use memory that has been freed". In later work, Engler and colleagues [21] extend their previous work [20] by automatically collecting sets of programmer beliefs from the source code instead of manually specifying programming rules. Hovemeyer and Pugh [32] present a technique based on bug patterns for finding the location of faults in Java programs. Bug patterns are error-prone coding practices. Java codes that conform to the bug patterns are flagged as errors through the use of bug-pattern detectors. Their technique is similar to Engler and colleagues' work in spirit. However, it is specifically applicable to Java programs. Flannagan and colleagues [24] also present the ESC/Java technique. Their technique provides an annotation language in which the developer can express design decisions formally. ESC/Java then checks the annotation against the actual program and reports any

inconsistencies in the program.

The limitation of fault-localization techniques that rely on specifications of faults is that it is difficult, if not impossible, to provide the specifications of all faults that can occur in software. For example, the techniques are ineffective against semantic faults because semantic faults cannot be specified easily. Also, the techniques produce large numbers of false positives, which serve as a deterrent to using the techniques. In general, determining whether a program contains a fault is an undecidable problem.

Hangal and Lam [29] present a technique called DIDUCE that uses dynamic invariants to automatically detect potentially faulty program statements. The technique initially hypothesizes a strict set of invariants and as the program executes relaxes the invariants as violations are detected. Their technique essentially combines anomaly detection and fault localization. The limitation of their technique is that an anomaly at a program statement does not imply that the statement is faulty. Their technique, however, adds significant overhead to the executing program and also it is prone to producing many false positives.

Renieris and Rice [60] present a fault-localization technique that uses one passing and failing execution to find the faulty program statement. Given a passing and failing execution that are similar their technique computes the set difference between the two executions. The intuition is that the result of the set difference consists of faulty statements. The limitation of their technique is that the set difference mostly contains statements that are associated with the failure instead of the statement that caused the failure. Their technique is similar to slicing in that the developer potentially must examine many statements before locating the fault. Another limitation of their technique is that it provides no direction as to how to search for the faulty statement. Empirical results have shown that current statistical techniques perform significantly better their technique.

Banerjee and colleagues [8] present a fault localization technique that assumes

the availability of a golden (non-faulty) implementation. Given a fault as the slicing criterion, their approach computes the dynamic slice from the criterion for both the golden implementation and the faulty program. The weakest precondition, which is a conjunction of predicates, is also computed in conjunction with the dynamic slices for both programs. The weakest preconditions for the two programs are then compared and if there is a predicate in one formula that is not implied in the other formula that predicate is marked as suspicious. The limitation of their approach is that they require the existence of a golden implementation of the faulty program. Barnejee and colleagues' technique improves on the technique, DARWIN, introduced by Qi and colleagues [57]. Given two program versions (where one of versions is a stable version and the other version a modification of the stable version) DARWIN computes the path conditions executed by the failing execution for both the versions using dynamic symbolic execution. Differences in the path conditions are then marked as suspicious. DARWIN is able to deal with branch errors whereas Banerjee and colleagues' approach is able to handle both branch and assignment errors. Jose and Majumdar [39] introduce a technique that constructs a boolean formula in conjunctive normal form using a failing test case and the faulty program. The boolean formula is then partially solved using MAX-SAT solver to find the set of possible clauses that if altered will fix the failing program. Their approach is expensive and also it is difficult to determine whether their approach will work on more complex faults instead of faults such as off-by-one errors.

1.3 Summary of Limitations

The limitations exhibited by statistical, slicing, and state-altering techniques reveal an information gap between the slicing techniques and the statistical fault-localization techniques. The slicing techniques use program dependence information for fault localization. Statistical techniques use statistical information derived from passing

and failing executions for fault localization and ignore program dependences. State-altering techniques do not use any statistical or program dependence information but instead rely on the computation that occurs at a program entity and how that computation affects the failure of the program.

1.4 Thesis Statement

This dissertation presents a novel approach to statistical fault localization that significantly improve on the state-of-the-art by addressing the problem from a statistical causal-analysis perspective. The dissertation addresses several limitations of statistical, slicing, and state-altering fault-localization techniques. The thesis statement is the following:

> *Statistical causal analysis combined with program analysis is a feasible and effective approach for finding the causes of software failures.*

1.5 Contributions

This dissertation makes the following five contributions to the body of knowledge in software engineering:

1. The probabilistic program dependence graph (PPDG): A novel program structure that combines program dependence graphs with statistical information derived from program executions to fascilitate arbitrary probabilistic reasoning over program behaviors. Empirical results that demonstrate the effectiveness of algorithms that utilize the PPDG.

2. A theoretical causal framework for fault localization that combines program-analysis information with causal inference methodology for observational studies;

3. A unification of the metrics used in current statistical fault-localization techniques and an assessment of the limitations inherent in the metrics from a causal perspective;

4. Fault-localization techniques based on the causal framework that use different combinations of program-analysis information;

5. Empirical studies on several software subjects that demonstrates the effectiveness and practicality of the causal framework.

The first contribution of the research is a probabilistic program dependence graph (PPDG), which is a probabilistic representation of a program that augments program dependence graphs with statistical information, and thus, leverages the strengths of both static and dynamic analysis of programs for effective fault localization. The PPDG provides a unifying framework that enables arbitrary reasoning (probabilistic or causal) over a program's behavior. The dissertation demonstrates empirically the effectiveness of algorithms that utilize the PPDG and in general the benefits of combining statistical information with program dependence graphs. The second contribution of the research is the first theoretically-based statistical fault-localization technique that actually addresses the fault localization problem from a causal perspective. This contribution was motivated by insights gained from the PPDG. Because the approach is theoretically motivated, it provides developers with guarantees that cannot be provided by current statistical techniques. For example, by providing a theoretically-based approach this research will ensure that the results produced are not susceptible to severe external validity problems. By providing a theoretically-motivated approach, the research provides the foundation upon which to build future fault-localization techniques. This contribution will extend the current state of the art in automatic software fault localization by providing more accurate fault-localization results. More accurate fault localization will reduce the time spent by developers

during debugging, which, in turn, will lead to higher software quality. The third contribution of the research is an analysis of current statistical fault-localization metrics from a causal perspective. The research shows that the metrics are similar to each other thereby unifying the metrics and providing insights to the limitations inherent in the metrics. The insights provided by the unification and inherent limitations of the metrics will direct future techniques to avoid pitfalls of the current techniques. The fourth contribution of the research is a demonstration of the importance of combining control dependences and data dependences in the causal model. This contribution bridges the gap that exists between fault-localization techniques that rely solely on statistical information and fault-localization techniques that rely solely on program analysis information.(e.g., slicing techniques).

The fifth contribution of the research presents empirical results that demonstrate the feasibility and efficacy of this research for finding the causes of software failures.

1.6 Overview of the Dissertation

Chapter 2 presents the necessary background material to understand the remaining chapters. Chapter 3 discusses the probabilistic program dependence graph (PPDG), a novel model that demonstrates in a principled way the importance of combining statistical information with program-analysis information. The PPDG is applied to the problem of fault localization and results are presented and discussed. Chapter 4 discusses a novel causal framework for finding the causes of software failures and also discusses how models can be instantiated from the framework. Chapter 5 presents an analysis of current statistical fault-localization metrics that unifies the metrics from the perspective of the causal framework. Empirical evidence is also presented that supports the analysis. Chapter 6 discusses a classical technique called matching and how it is used in conjunction with the causal framework for effective causal analysis. Finally, Chapter 7 presents the conclusion, merit, and future work of this dissertation.

CHAPTER 2

BACKGROUND

This research builds on work in the areas of probabilistic graphical models [11], Pearl's Structural Causal Model [53], Neyman and Rubin's potential outcome model [51, 62], and program analysis [22]. This chapter presents the necessary background material.

2.1 Dependences and Dependence Graphs

This section presents the different types of dependence graphs used in this work. The dependence graphs provide structural abstractions of a program.

Definition 1. *A **directed graph** $\mathcal{G}_d$ is a pair (N, E) where*

1. *N is a finite set whose elements are called nodes or vertices, and*
2. *E is a finite set consisting of pairs of nodes called directed edges.*

If (u, v) is a directed edge in E, then there is an arrow from u to v. v is said to be a direct successor of u and u is said to be a direct predecessor of v. The *out-degree* of a node n in $\mathcal{G}_d$ is the number of edges leaving n. Two nodes u and v are said to be adjacent if there is an edge between them.

Definition 2. *A **path** of length k from node n to n' in $\mathcal{G}_d$ is a sequence $< v_0, v_1, ..., v_k >$ such that $n = v_0$ and $n' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, ..., k$.*

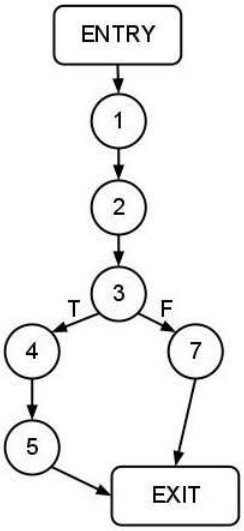
A path p $= < v_0, v_1, ..., v_k >$ in $\mathcal{G}_d$ forms a *cycle* if $v_0 = v_k$.

Definition 3. *A **directed acyclic graph** is a directed graph with no cycles.*

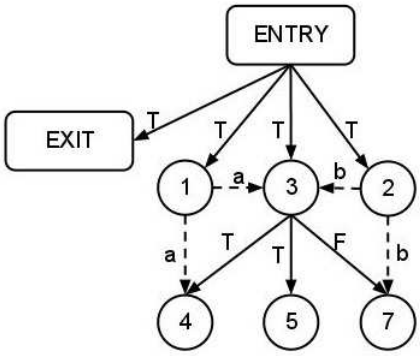
Definition 4. *A **control flow graph** (CFG) for a program $\mathcal{P}$ is a directed graph $\mathcal{G}_{cfg} = (N, E, n^s, n^e, \varrho)$ where*

```
    procedure Max( ){
1       int a = read_int();
2       int b = read_int();
3       if ( a > b ) {
4          print(a);
5          return;
6       }
7       print(b);
8   }
```

(a) Procedure Max.



(b) CFG of Max.



(c) PDG of Max.

Figure 2: Function max with its control flow graph (CFG) and program dependence graph (PDG).

1. *N is a set of nodes representing statements and predicates in $\mathcal{P}$,*

2. *E is a set of directed edges representing the flow of control between statements in $\mathcal{P}$,*

3. *$n^s \in N$ is the entry to $\mathcal{P}$ and has no predecessors,*

4. *$n^e \in N$ is the exit from $\mathcal{P}$ and has no successors, and*

5. *ϱ is a function that maps E to true (T) or false (F)— "outcomes of predicates",*

or ϵ *(*$\varrho : E \rightarrow \{T, F, \epsilon\}$*).*

Figure 2(a) shows a procedure[1] *Max*, which prints the maximum of two integers. The left column of Figure 2(a) shows the line numbers for the procedure and the right column shows the statements. Figure 2(b) shows the control flow graph. Each node in the control flow graph corresponds to a statement in the program; a node is labeled with the statement's line number. For example, node 1 represents the statement at line 1. Nodes labeled $ENTRY$ and $EXIT$ represent the entry to and exit from the program, respectively. Edges between nodes in the control flow graph represent the flow of control in *Max*. Node 3 is a predicate node and has two outgoing edges labeled "T" for true and "F" for false. If the condition at the statement at line 3 in the program evaluates to *true*, edge (3,7) is taken; if the condition evaluates to *false*, edge (3,4) is taken.

Definition 5. *A node* u *in* $\mathcal{G}_{cfg}$ ***dominates*** *a node* v *in* $\mathcal{G}_{cfg}$ *if and only if*

1. *every path from* n^s *(program entry) to* v *contains* u *and*

2. u *dominates itself.*

Definition 6. *A node* v *in* $\mathcal{G}_{cfg}$ ***post-dominates*** *a node* u *in* $\mathcal{G}_{cfg}$ *if and only if every path from* u *to* n^e *(program exit) contains* v*.*

For example, in Figure 2(b) node 3 dominates nodes 4, 5, 6, 7, and $EXIT$ and node 3 post-dominates nodes 1, 2, and $ENTRY$.

Definition 7. *A node* n_2 *in* $\mathcal{G}_{cfg}$ *is* ***control dependent*** *on node* n_1 *with label* L *("T" or "F"), if*

1. *there exists a path* p *from* n_1 *to* n_2 *in* $\mathcal{G}_{cfg}$*,*

2. n_2 *post-dominates every node in* p *except* n_1*, and*

[1]We will use procedure and program interchangeably.

3. n_2 does not post-dominate n_1.

The definition means that node n_1 has two outgoing edges and all paths to the program exit along one edge contain n_2 and at least one path along the other edge does not contain n_2. For example, in Figure 2(b) nodes 1, 2, 3, and $EXIT$ are control dependent on node $ENTRY$ with labels "T" and nodes 4, 5, and 7 are control dependent on node 3.

Definition 8. *A node n_2 is **forward control dependent** on a node n_1 if n_2 is control dependent on n_1 and n_2 does not dominate n_1.*

Intuitively, forward control dependences are control dependences that can be realized during the execution of a program without necessarily executing the dependent node (n_2) more than once.

Definition 9. *In a control flow graph $\mathcal{G}_{cfg}$, node n_2 is **data dependent** on node n_1, if*

1. n_1 defines a variable v,

2. there is a path in $\mathcal{G}_{cfg}$ from n_1 to n_2 that does not redefine v, and

3. n_2 uses v.

For example, in Figure 2(b), nodes 3 and 4 are data dependent on node 1 and node 7 is data dependent on node 2. A program dependence graph is defined using control dependence and data dependence [22].

Definition 10. *A **program dependence graph** (PDG) is a directed graph whose nodes are the nodes of the $\mathcal{G}_{cfg}$ and the edges between the nodes represent control and data dependences.*

Figure 2(c) shows the program dependence graph of *Max*. The nodes in the graph are labeled with the line numbers of the statements in *Max*. The solid edges represent control-dependence edges and dashed edges represent data-dependence edges.

Definition 11. *The **dynamic program dependence graph** (Dynamic-PDG) is a directed graph constructed from a set of program executions in which nodes represent executed statements and edges represent executed control dependences and data dependences.*

Definition 12. *A statement s_1 is a **dynamic forward-control-dependence predecessor** of s_2 if s_1 is a forward control-dependence predecessor of s_2 in the Dynamic-PDG.*

The Dynamic-PDG, which is a subgraph of the PDG, is similar to the graph obtained with Agrawal and Horgan's [3] technique (Approach 1) for dynamic slicing. However, the latter is created using one execution whereas the Dynamic-PDG described here is created using a set of executions.

Definition 13. *The dynamic forward-control-dependence subgraph is a subgraph of the Dynamic-PDG that contains only forward control-dependence edges.*

2.2 Dependency Networks

In this section, we present dependency networks. Dependency networks [30] are annotated graphs that capture the probabilistic relationships between random variables. Various kinds of probabilistic inferences (e.g., prediction) can be performed on the graph using sampling-based algorithms.

Definition 14. *A **dependency network** [30] is a triple (S, G, Ω) where S represents a set of random variables, $G = (N, E)$ is a possibly cyclic directed graph, and Ω represents a set of conditional probability distributions. N and E are the set of nodes and the set of directed edges in G, respectively, with nodes in G corresponding to random variables in S and edges in G representing dependences among the random variables.*

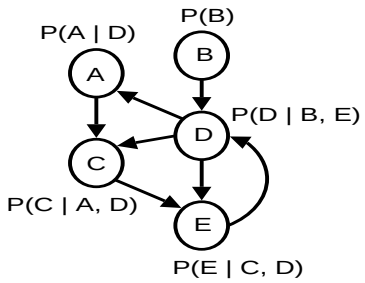


Figure 3: An example of a dependency network.

Definition 15. *Two random variables X_i and X_j are said to be* ***conditionally independent*** *given X_k, if the probability distribution of X_i does not depend on X_j given that the value of X_k is known, that is, $P(X_i|X_j, X_k) = P(X_i|X_k)$.*

Suppose $S = \{X_1, \cdots, X_n\}$ is a set of random variables. Each node in G corresponds to a variable $X_j \in S$, and directed edges between nodes represent dependences between the variables in S. Intuitively, the structure of G captures the way in which the variables in S are related to each other. In this discussion, we use X_j to denote a random variable and its corresponding node in G. The set $\mathrm{Pa}(X_j)$ (parents of node X_j) is the set of nodes X_i for which there is an edge (X_i, X_j) in G. A node in a dependency network is required to be conditionally independent of its nondescendants, given the states of its parents. We assume each X_j has a set of mutually exclusive discrete states $x = \{x_1, \cdots, x_k\}$. Thus, X_j cannot assume two states at the same time. Each node X_j in G has a conditional probability distribution, $\mathrm{P}(X_j|\mathrm{Pa}(X_j))\in \Omega$, relating the states of X_j to the states of its parents $\mathrm{Pa}(X_j)$. Conditional probability tables are used to represent the conditional probability distributions. The parameters of the network are the probabilities in the conditional probability tables, which are estimated from data.

Figure 3 shows an example of a dependency network that consists of five nodes (random variables): A, B, C, D, and E (i.e., S={A, B, C, D, E}). The figure also shows the conditional probability distributions for each node. The conditional probability

distribution for A, B, C, D, and E are P(A | D), P(B), P(C | A, D), P(D | B, E), and P(E | C, D), respectively. The nodes in the graph are conditionally independent of their non-descendants given their parents (i.e., the graph satisfies the Markov condition [49]). For example, node E is conditionally independent of {B} given its parents {C, D}.

2.3 Potential Outcome Framework

Many everyday activities in which people are engaged involve thinking causally; asking "what if" questions. For example, "will a particular diet product cause me to lose weight." In computer programs, we are interested in answering causal questions such as "what caused the program to fail" or "what is causing the program to run slowly". To answer such causal "what if" questions, several models have been developed. The potential outcome model is one such model. The potential outcome model of causality was pioneered by Neyman [51] and Rubin [62] and it has been influential in a number of fields, including economics, epidemiology, and social science [46]. For the purpose of exposition, we shall use a binary causal variable but the model is applicable to multi-causal variables. We first define a few terms.

Definition 16. *A **treatment** is an intervention an investigator may apply to a set of units to assess its effects relative to no intervention (i.e., the control).*

Definition 17. *A **unit** is a person or object on which the treatment is applied to assess the effects of the treatment or no treatment.*

Definition 18. *The **potential outcome** is the potential response of a unit to treatment or no treatment.*

For a binary causal variable, the potential-outcome model defines two states to which each unit (member) of the population could be exposed; the states are called *treatment* and *control*. These states correspond to the values of a causal random

variable, which we denote by T (treatment variable):

$$T = \begin{cases} 1, & \text{for treated units} \\ 0, & \text{for untreated units} \end{cases} \tag{1}$$

For example, suppose a researcher wants to investigate the effects of class attendance on test scores. Under the potential outcome model the causal variable indicates class attendence and the potential outcomes are test scores. The causal variable T has two states: "attend class" and "do not attend class" where the treatment state is "attend class" and the control state is "do not attend class". The units in this example are students. The state of a unit i is represented by t_i. If i is treated then $t_i = 1$, if i is untreated then $t_i = 0$. Also, treated units are said to be in the treatment group and untreated units are said to be in the control group. The relation between the observed outcome (Y) and the potential outcomes (Y^1, Y^0) of units exposed to either the treatment state or control state is:

$$Y = Y^0 + (Y^1 - Y^0)T \tag{2}$$

$$Y = \begin{cases} Y^1, \text{ if } T = 1 \\ Y^0, \text{ if } T = 0 \end{cases} \tag{3}$$

Equation (2) provides a compact representation of the observed outcome (Y) and the two potential outcomes: Y^1 and Y^0. The random variables Y^1 and Y^0 represent the population level potential outcomes for the treatment group (attend classes) and control group (do not attend classes), respectively. The population-level estimates are estimates that are derived from the entire population as opposed to being derived from a subset of the population. We represent the actual outcomes for a treated unit i and an untreated unit as y_i^1 and y_i^0, respectively. If i is treated (i.e, attend class) then y_i^0 is referred to as a *counterfactual* because it represents what the test score

would have been had unit i not attended classes. Similarly, if unit i does not attend classes (i.e, $Y^0 = y_i^0$) then (y_i^1) represents what i's test score would have been had i attended class. For a given unit i Equation (2) is:

$$y_i = y_i^0 + (y_i^1 - y_i^0)t_i \tag{4}$$

From Equation (4), the causal effect for unit i is the difference $y_i^1 - y_i^0$. Equation (4) also shows that for a given unit i it is not possible to observe both outcomes y_i^1 and y_i^0 at the same time; this problem is referred to as the *fundamental problem of causal inference* [31, 46]. The inability to observe both outcomes means that it is not possible to compute the unit-level causal effect of a unit i. For example, if a student attends class the researcher can only know what the test score is under treatment but the researcher cannot know what the test score would have been had the student not attended class (control). Although it is not possible to compute unit-level causal effects, *average treatment effects* or *average causal effects* can be estimated. (We use "mean" and "average" interchangeably.) There are two approaches by which average treatment effects are estimated: through randomized controlled experiments or through observational studies. We discuss randomized-controlled experiments and then observational studies.

2.3.1 Randomized Controlled Experiments

Randomized controlled experiments are considered the "gold standard" for estimating treatment (causal) effects. In a randomized experiment, the experimenter randomly assigns units to the treatment state and control state. Each unit should have nonzero probabilities of being assigned to both treatment and control; that is, the assignment mechanism must be stochastic. In our class attendance and test scores example, every student should have a nonzero probability of being assigned to attending class. An example of a random assignment mechanism is for the experimenter to choose $P(t_i = 1) = 1/2$; that is each unit has equal chance of being exposed to either

treatment or control. When assignment of members to treatments are random the treatment variable T is independent of the potential outcomes; the treatment assignment is considered *ignorable* [46] and the treatment and control groups are considered *exchangeable*.

$$(Y^1, Y^0) \perp\!\!\!\perp T \tag{5}$$

Random treatment assignments tend to ensure that the treatment and control groups are balanced. This *balance* means that the units in the treatment group have similar characteristics to units in the control group. Additionally, if the units are assigned randomly then knowing whether a unit is assigned to treatment or control provides no information about the outcomes (y_i^1) or (y_i^0) of the units. Equation (6) presents the average treatment effect under random treatment assignment (here we assume that the population size is infinite and there is no estimation error).

$$\tau = E[Y^1] - E[Y^0] \tag{6}$$

Equation (7) shows the population-level estimate of the sample average treatment effect.

$$\tau = E[Y|T = 1] - E[Y|T = 0] \tag{7}$$

where $E[\cdot]$ denotes the expectation operator. In practice, the population size is finite and average treatment effect is computed from a sample of the population. Suppose an observed sample of S units is selected from a population of size N where N is large and $S = S_1 + S_0$ where S_1 and S_0 are the number of units in the treatment and control groups, respectively. Equation (8) shows the sample version of the average treatment effect.

$$\tau_{sample} = \frac{1}{|S_1|} \sum_{i \in S_1} y_i - \frac{1}{|S_0|} \sum_{i \in S_0} y_i \tag{8}$$

The sample average-treatment effect is computed by exposing some units from the

population to treatment and control groups and taking the difference between the outcomes. The average treatment effect is then the effect each unit would experience if exposed to the treatment state. In our example, some students are randomly assigned to attend class (S_1) or not to attend class (S_0). The average difference in test scores of the two groups is then the sample average treatment effect.

The right side of Equation (7) consists of two components, which are the population-level estimates of their sample level counterparts in Equation (8). The sample treatment effects converge in probability to their population-level treatment effects, which is given in Equations (9) and (10). In statistics, a sequence of random variables $\{X_n\}$ converges to X (true value of the random variable) in probability if $\lim_{i\to\infty} P(|X - X_i| \geq \varepsilon) = 0$, where ε is some error.

$$\frac{1}{|S_1|}\sum_{i\in S_1} y_i \xrightarrow{p} E[Y|T=1] \tag{9}$$

$$\frac{1}{|S_0|}\sum_{i\in S_0} y_i \xrightarrow{p} E[Y|T=0] \tag{10}$$

The first component $E[Y|T = 1]$ of Equation (7) represents the average potential outcome of the units exposed to the treatment state and $E[Y|T = 0]$ represents the average potential outcome of the units exposed to the control state.

Although using randomized controlled experiments is the ideal approach for estimating average treatment effects, in practice it is often not possible to perform randomized experiments (e.g., because of ethical issues). For example, the experimenter cannot force students to attend or not to attend class. Therefore, researchers must rely on data from *observational studies* to estimate average treatment effects.

2.3.2 Observational Studies

The purpose of observational studies as defined by Cochran [17] is to explain cause and effect relationships from data for which it is not feasible to use randomized experiments. An observational study differs from a randomized experiment in that

the investigator has no control over the assignment mechanism. Units in the treatment or control group may have ended in the groups based on a variety of factors. The factors may include a common cause of the treatment (T) and the outcome (Y). To estimate causal effects from observational studies, researchers must deal with those factors known as confounding covariates (or confounders) when estimating causal effects in an observational study.

Definition 19. *A confounding covariate or confounder is a variable that influences both the treatment variable (T) and the outcome variable (Y).*

If confounders are not accounted for when estimating the average treatment effect in observational studies, Equation (7) becomes a biased and inconsistent estimate of the average treatment effect as shown in Equation (11). In our class attendance example, factors such as socio-economic backgrounds, age, and gender of the students can be considered confounders because they might dictate who attends class.

$$\tau_{biased} = E[Y^1|T=1] - E[Y^0|T=0] \tag{11}$$

Suppose N is potentially infinite and π is the proportion of units assigned to treatment then $1-\pi$ is the proportion assigned to control. Equation (12) shows a weighted decomposition of the average treatment effect (τ).

$$\begin{aligned} \tau &= \{\pi E[Y^1|T=1] + (1-\pi)E[Y^1|T=0]\} \\ &- \{\pi E[Y^0|T=1] + (1-\pi)E[Y^0|T=0]\} \end{aligned} \tag{12}$$

By performing some algebraic manipulations, the bias inherent in τ_{biased} is shown in Equation (13),

$$\begin{aligned} \tau_{biased} &= \tau + \{E[Y^0|T=1] - E[Y^0|T=0]\} \\ &+ (1-\pi)\{E[Y^1 - Y^0|T=1] - E[Y^1 - Y^0|T=0]\} \end{aligned} \tag{13}$$

The two components on the right side of Equation (12) show the bias inherent in Equation (11). The first component $\{E[Y^0|T=1] - E[Y^0|T=0]\}$ is referred to as the *baseline bias* [46] or selection bias [5]. The *baseline bias* is the difference in mean potential outcome of units exposed to treatment had they not been exposed to the treatment and the mean potential outcome of units not exposed to the treatment. The second source of bias is $(1-\pi)\{E[Y^1 - Y^0|T=1] - E[Y^1 - Y^0|T=0]$, which is referred to as the *differential treatment-effect bias*. This bias is the result of the difference in the treatment effects between those in the treatment group and those in the control group. For example, suppose students who attend class have higher test scores, there are three possible reasons. The first reason is that students who attend classes may attain high test scores; this is represented by the average treatment effect τ. The second reason is that students who attend class may naturally have high test scores had they not attended class (maybe high intelligent quotient (IQ)); this is represented by the baseline bias. The third reason is that students who attended classes may attain high test scores than students who did not attend classes had the students attended classes; this is represented by the differential treatment-effect bias.

In practice the most important bias is the baseline bias; if τ_{biased} is to be unbiased and consistent the treatment (T) must be independent of the outcome (Y). To estimate average treatment effects from observational data, the conditional independence assumption is employed [5].

Definition 20. *The conditional independence assumption (CIA) also known as selection on the observables asserts that conditional on a set of confounding covariates (X), the treatment (T) is independent of the potential outcomes* (Y^1, Y^0).

$$(Y^1, Y^0) \perp\!\!\!\perp T \mid X \tag{14}$$

Equation (14) reads: the potential outcomes are independent of the treatment T given a set of covariates X. CIA brings some of the properties of randomized experiments

to bear on observational studies by ensuring that the units involved in the causal analysis have similar characteristics, ensuring that the data is balanced. Therefore for an observational study, Equation (12) is represented as,

$$\begin{aligned} \tau &= \{\pi E[Y^1|X, T=1] + (1-\pi)E[Y^1|X, T=0]\} \\ &- \{\pi E[Y^0|X, T=1] + (1-\pi)E[Y^0|X, T=0]\} \end{aligned} \tag{15}$$

If the set of covariates X is not sufficient to render T independent of Y, then from Equation (15) the bias of the average treatment effect in observational studies is,

$$\begin{aligned} \tau_{biased} &= E[Y^1|X, T=1] - E[Y^0|X, T=0] \\ &= \tau + \{E[Y^0|X, T=1] - E[Y^0|X, T=0]\} \\ &+ (1-\pi)\{E[Y^1 - Y^0|X, T=1] - E[Y^1 - Y^0|X, T=0]\} \end{aligned} \tag{16}$$

If conditioning on a set of covariates X is sufficient to render T independent of Y, then Equation (16) becomes,

$$\begin{aligned} \tau &= E[Y^1 \mid X, T=1] - E[Y^0 \mid X, T=0] \\ &= E[Y^1 \mid X] - E[Y^0 \mid X] \end{aligned} \tag{17}$$

Because $E[\cdot|X, T] = E[\cdot|X]$ the baseline bias and the differential treatment-effect bias evaluates to zero.

2.3.3 Regression and Causality

In practice, a regression model is sometimes used to estimate the average treatment effect in observational studies. Equation (18) shows a linear regression model that can be used to estimate the average treatment effect (τ), assuming that Y and T are conditionally independent given X and that the model is correct.

$$Y = \alpha + \tau T + \beta X + \varepsilon \tag{18}$$

In the regression model, Y represents the potential outcome variables $Y = (Y^1, Y^0)$, T is the treatment variable, X is a set of covariates, α is an intercept, τ and β are

coefficients of T and X respectively, and ε is an error term that is uncorrelated with T. A linear model like Equation (18) is by no means the only choice, or necessarily the best choice for estimating the average treatment effect. Some other alternatives include a logistic regression model [11], and very sophisticated nonparametric and semiparametric models [34].

2.4 Structural Causal Model

Causal structures are at the heart of Pearl's Structural Causal Model (SCM), because they are used to clearly represent both (1) the causal assumptions that permit statistical techniques to be used with observational data to make inferences about causality, not just about associations, and (2) changes such as treatments and external interventions.

2.4.1 Causal Graphs and Models

Definition 21. *A **causal graph** is a directed acyclic graph $\mathcal{G}$ whose nodes V represent random variables (corresponding to causes and effects) and whose edges represent causal relationships.*

An edge $X_i \to X_j$ in $\mathcal{G}$ with $i \neq j$ indicates that X_i (potentially) causes X_j. Each random variable $X_i \in V$ has a probability distribution $P(X_i)$, whose form may or may not be known. (We denote the values of random variables by corresponding lowercase letters, that is X_i's value is represented by x_i.) Variables without parents in the causal graph are termed exogenous variables whereas variables with parents are termed endogenous variables.

Figure 4 is an example of a causal graph with five nodes S, X, Y, V, and Z. Given a causal graph, there are three basic patterns of causal relationships that can exist between any three connected random variables. The three patterns determine how a random variable causally affects another random variable. Figure 5 shows the

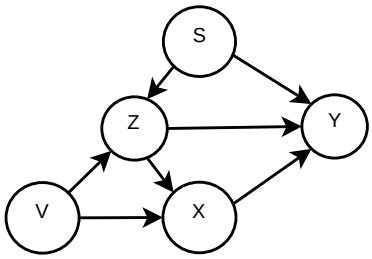


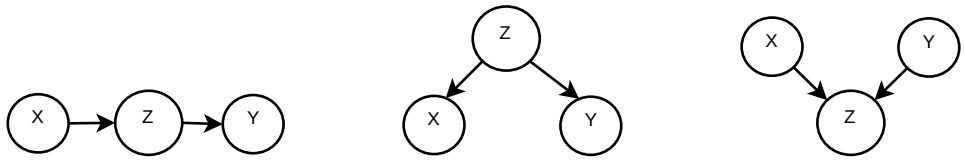
Figure 4: Causal Graph

three patterns: *mediation (chain), mutual dependence (fork), and mutual causation (collider).* In Figure 5(a) the variable X affects Y because X causes Z and Z causes Y. X and Y are unconditionally associated and Y is rendered independent of X if the value of Z is fixed (i.e, $Z = z$). In Figure 5(b) variables X and Y share a common cause Z. X and Y are unconditionally associated, however X and Y are rendered independent of each other if the value of Z is fixed. In Figure 5(c), variable Z has two causes X and Y. This particular structure is different from the others because X and Y are conditionally associated given that the value of Z is known.

All causal effects associated with the causal model $\mathcal{M} = (\mathcal{G}, P)$ are identifiable (can be estimated) if $\mathcal{M}$ is Markovian, which means that each random variable X_i is conditionally independent of all its nondescendants, given the values of its parents (immediate predecessors) PA_i in $\mathcal{G}$ (parental Markov condition) [53]. If $\mathcal{M}$ is Markovian then the joint distribution of the random variables can be factored as follows:

$$P(x_1, x_2, ..., x_n) = \prod_i P(x_i|pa_i) \tag{19}$$

Here, the probability distribution P is compatible with $\mathcal{G}$ in that the independences expressed in $\mathcal{G}$ are admitted in P. Pearl and Verma [55] proved that $\mathcal{M}$ will satisfy the parental Markov condition if it corresponds to a functional causal model, in the sense that for each node $X_i \in \mathcal{G}$, the relationship between X_i and its parents can be



(a) Mediation (chain) (b) Mutual dependence (fork) (c) Mutual causation (collider)

Figure 5: Three main causal structures given three nodes X, Y, and Z.

described by a structural equation of the form

$$x_i = f_i(pa_i, u_i) \tag{20}$$

Here f_i, which represents a causal process, may be any function, and for $i = 1, ..., n$ the random variables U_i, which represent random errors due to unobserved variables, are independent of each other. Equation (20) is also known as a functional causal model. (If X_i has no parents, we write $x_i = f_i(u_i)$.) Put another way, $\mathcal{M}$ is Markovian if it represents functional relationships among a set of random variables and if any external sources of error are mutually independent. For the purposes of causal inference, the functions f_i and the distributions of the error variables U_i need not be known. In this sense, $\mathcal{M}$ is nonparametric. Moreover, the U_i need not be explicitly represented in $\mathcal{G}$. For example, the causal model in Figure 4 is Markovian therefore the joint distribution factors into $P(s, v, x, y, z) = P(y|s, x, z) \cdot P(z|s, v) \cdot P(x|v, z) \cdot P(s) \cdot P(v)$.

2.4.2 Back-door Criterion

Suppose the causal effect of variable X on Y in Figure 4 is to be estimated, then the causal effect will be biased because of the presence of covariates V and Z (i.e., variables that influence both the cause (X) and the effect (Y)). To facilitate the estimation of a causal effect without bias, Pearl introduces the concept of the *Back-Door Criterion*. Pearl's *Back-Door Criterion* [53] defines, in terms of causal graphs, the characteristics that make a set of confounding covariates sufficient to adjust for

confounding bias. Before presenting the Back-Door Criterion, we must first define the concepts of "blocking" and "d-separation" [53]. Note that, contrary to usual directed graph terminology, this definition refers to causal graph "paths" whose arrows (edges) may be directed either forward or backward along the path.

Definition 22. *A set S of nodes in a causal graph $\mathcal{G}$ is said to* ***block*** *a path p if either (1) p contains a* ***chain*** $U \to M \to V$ *or a* ***fork*** $U \leftarrow M \to V$ *whose middle node M is in S, or (2) p contains at least one* ***collider*** $U \to M \leftarrow V$ *such that the middle node M is not in S and no descendant of M is in S. If S blocks all paths from A to B, it is said to* ***d-separate*** *A and B.*

Pearl showed that if S d-separates X and Y then X and Y are *conditionally independent* given S, that is, $P(Y|X,S) = P(Y|S)$. For example in Figure 4, the variable V is d-separated from Y given $\{X, Z, S\}$. Conditioning only on $\{X, Z\}$ does not d-separate V from Y because by conditioning on Z, V and Y become dependent because Z is a collider and the path $V \to Z \to S \to Y$ becomes open.

Definition 23. *A set S of nodes satisfies the* ***Back-Door Criterion*** *relative to a pair of nodes A, B in a causal graph G if*

1. *No node in S is a descendant of A; and*

2. *S blocks every path between A and B that contains an edge into A.*

Similarly, if A and B are two disjoint subsets of nodes in G, then S is said to satisfy the Back-Door Criterion relative to A, B if it satisfies the criterion relative to any pair $A_i \in A$, $B_j \in B$.

The name of the Back-Door Criterion comes from condition (2), which requires that paths that enter A through the "back door" be blocked. For example in Figure 4, to estimate the causal effect of X on Y all back-door paths from X to Y must be blocked. There are four back-door paths from X to Y: $\{X \to Z \to Y\}$, $\{X \to Z \to$

$S \rightarrow Y\}$, $\{X \rightarrow V \rightarrow Z \rightarrow Y\}$, and $\{X \rightarrow V \rightarrow Z \rightarrow S \rightarrow Y\}$. To block all the back-door paths the sets $\{V, Z, S\}$ or $\{Z, S\}$ must be conditioned on.

The Back-Door Criterion unifies a number of strategies including conditioning, stratification, and matching [46]. In general if a set of variables blocks all back door paths in a causal graph between a treatment indicator T (X is the treatment indicator in Figure 4) and an outcome variable Y, then those paths do not contribute to the association between T and Y [46]. Conditioning on the covariates that block all back-door paths, the average treatment effect of T upon Y can in principle be estimated without confounding bias. In practice, the causal model may be contingent, and one may not be certain that a given set of covariates blocks all back-door paths between T and Y.

CHAPTER 3

PROGRAM DEPENDENCES AND STATISTICAL INFORMATION

A variety of graphical models have been used in software-engineering applications to abstract relevant relationships between program elements, and thereby, facilitate program analysis and understanding. These models include control flow graphs, call graphs, finite-state automata, and program dependence graphs. Graphical models produced by static analysis generally indicate that certain occurrences are possible at runtime (e.g., control transfers, calls, state occurrences, state transitions, and information flows), whereas models produced by dynamic analysis indicate what actually does occur during one or more executions. However, commonly used graphical models of internal program dynamics do not support making inferences about the likelihood of a particular program behavior. The lack of support severely limits their utility for reasoning about the causes and effects of inherently uncertain program behaviors, such as runtime failures.

Program dependence graphs (PDGs) [22], which have proven useful in software-engineering applications, such as testing [10], debugging [69], and maintenance [26], model potential semantic dependences [56] between program elements. However, they do not model the strengths of any corresponding statistical dependences between the program elements. Probabilistic graphical models have been useful in several fields (e.g., medicine [25] and robotics [67]) because of their ability to model both the presence of certain dependences between variables of interest and the way in which the variables are probabilistically conditioned on other variables. A probabilistic graphical model derived from a program dependence graph provides a natural framework

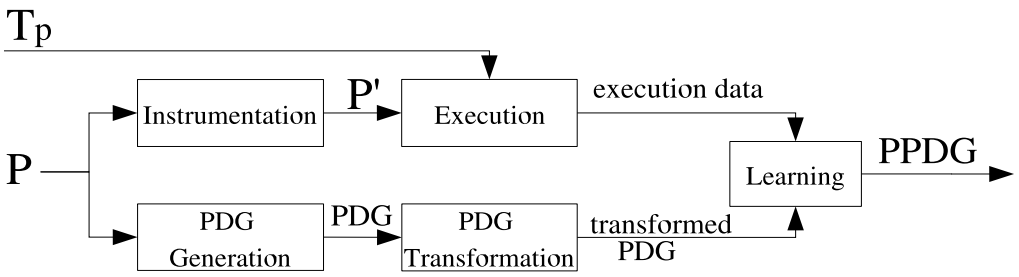


Figure 6: Steps in the construction of a PPDG.

for modeling both the presence of dependences and their statistical strengths.

In this chapter, we show how augmenting program dependence graphs with statistical dependence (and independence) information in the principled way provided by probabilistic graphical models [47] can substantially increase the utility of program dependence graphs in some software-engineering applications. The probabilistic model captures the conditional statistical dependence and independence relationships among program elements in a way that facilitates making probabilistic inferences about program behaviors. We call this model a *Probabilistic Program Dependence Graph* (PPDG). The technique produces the PPDG for a program by augmenting its program dependence graph automatically. The technique associates a set of abstract states with each node in the PPDG. Each abstract state represents a (possibly large) set of concrete nodes states, in a way that is chosen to be relevant to one or more applications of PPDGs. Each node has a conditional probability distribution that relates the states of the node to the states of its parent nodes. The technique estimates the parameters of the probability distribution by analyzing executions of the program, which are induced by a set of test cases or captured program inputs.

The ability of PPDGs to facilitate probabilistic reasoning about program behaviors makes them potentially valuable for software-engineering tasks. For example, (1) the PPDG can be used as a knowledge base from which contextual information can be generated to facilitate program understanding (2) To reason about the likely causes of software failures, and to enable software testing tools generate useful test cases.

In this dissertation, we focus on the fault-localization problem. We present evidence indicating that PPDGs can be useful for fault localization. Intuitively, PPDGs are potentially well suited for finding the causes of software failures because they can indicate how a failing execution differs from successful ones, both structurally and statistically.

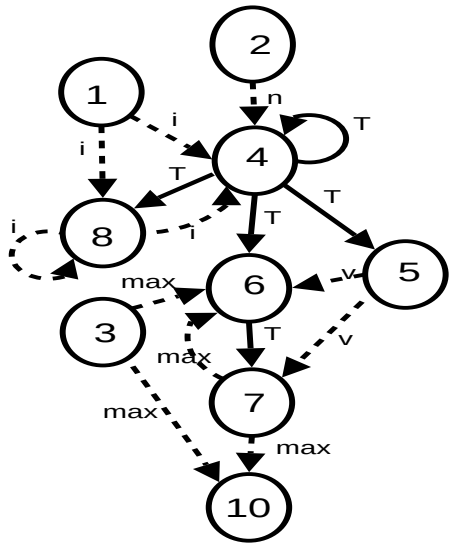
3.1 The Probabilistic Program Dependence Graph

The probabilistic program dependence graph (PPDG) is a general statistical model of a program that combines a program's program dependence graph (PDG) and statistical information obtained from dynamic analysis of the program. The model captures the conditional statistical dependence and independence relationships among program statements. In theory, the model facilitates arbitrary probabilistic reasoning about program behaviors, such as reasoning about cause and effects of program failures. There are different types of probabilistic program dependence graphs that can be constructed based on probabilistic graphical models. In this work, we construct the PPDG by transforming the PDG of a program into a dependency network [30]. A dependency network is used because it permits directed cycles, which are present in the PDGs of typical programs because of loops. Henceforth, the terms "loop" and "cycle" are used interchangeably.

The process of producing a PPDG consists of five main steps, as illustrated in Figure 6. First, the PDG-generation step generates the PDG of the input program P. Second, the PDG-transformation step takes the PDG, and transforms it by structurally changing the PDG and specifying states at nodes in the PDG, which results in a transformed PDG. Third, the Instrumentation step inserts probes into P to gather the execution data needed to estimate the parameters of the PPDG, and produces the instrumented program P'. Fourth, the Execution step executes P' with its test suite T_P to generate the execution data. Finally, the Learning step generates a PPDG

```
 0 void findmax() {
 1   int i = 0;
 2   int n = read_int();
 3   int max = 0;
 4   while (i < n){
 5     int v = read_int();
 6     if (v > max)
 7       max = v;
 8     i++;
 9   }
10   print(max);
11 }
```

(a) findmax



(b) PDG

Figure 7: Example program findmax (a) and its PDG (b).

based on the execution data and the transformed PDG by estimating the parameters of the PPDG. The resulting PPDG is formally defined as follows.

Definition 24. *A **Probabilistic Program Dependence Graph** (PPDG) for program $\mathcal{P}$ is a triple $(\mathcal{G}, \mathcal{S}, \mathcal{Q})$ where*

1. *$\mathcal{G} = (N, E)$ is the transformed PDG of $\mathcal{P}$, N is a set of nodes representing statements in $\mathcal{P}$ and E is a set of directed edges representing control and data dependences between statements in $\mathcal{P}$,*

2. *$S : N \rightarrow$ states, represents mappings from nodes to states, where states are discrete abstractions of program behaviors, and*

3. *$Q : N \rightarrow CPDs$, represents mappings from nodes to conditional probability distributions (CPDs).*

We use the example program *findmax* in Figure 7(a) to facilitate the discussion of the PPDG. The next Section presents detailed steps of the PPDG-generation process that are novel for the technique: PDG Transformation and Learning. We refer the

reader to Ferrante, Ottenstein, and Warren [22] for a detailed discussion on how the PDG is constructed.

3.1.1 PDG Transformation

During this step, the technique (1) structurally transforms the PDG into a dependency network by adding nodes and edges to it and (2) specifies the states of the nodes. We call the graph that results after transforming the PDG the *transformed PDG*. The technique assigns to each node in a program's transformed PDG a finite set of discrete abstract states, each of which represents a set of related concrete states of the corresponding statement. Hereafter, the term "state" refers to an abstract state. The states of a node must be mutually exclusive (i.e., a node cannot be in two different states at the same time). The technique initially assigns a default state denoted by the symbol $\perp$ to each node. The $\perp$ state is the state a node assumes when it has not been executed. When a node is executed, it is assigned a state distinct from $\perp$.

The state of a PPDG node abstracts a part of the program's state that pertains to the node when the program executes. There are different ways to model this "local" concrete state. In this work, we model it in one or both of two ways depending on whether the node represents a branch predicate, a statement that uses one or more variables, or both. These characterizations are intended to reflect certain aspects of a node's concrete state that are relevant to applications, such as fault localization. (Other aspects may also be relevant, but we shall not consider them in this work.)

The technique characterizes the state of a node representing a branch predicate by the outcome of the predicate. The technique characterizes the state of a node representing a statement s that uses one or more variables by the set of variable definitions that reach those uses during execution (i.e., by the definitions on which s is dynamically data dependent.)

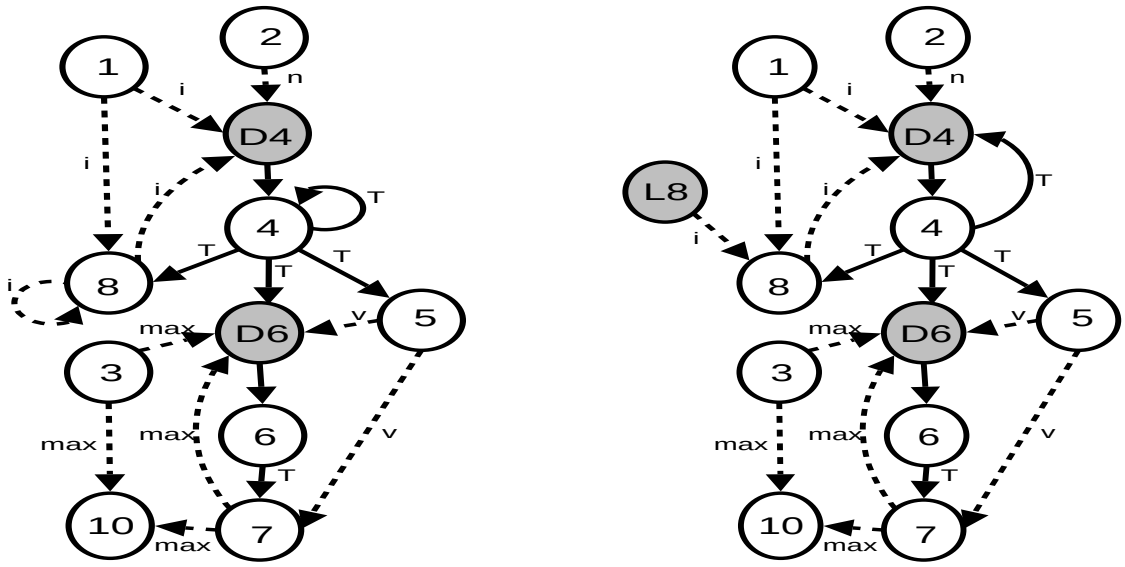


Figure 8: Structurally transforming PDG of *findmax*.

Definition 25. *A statement s is **dynamically data dependent** on a set of definitions if, during the program's execution, the definitions actually reach s and are used at s.*

Dynamic data dependences are a subset of static data dependences. The technique transforms all predicates into simple predicates.

Definition 26. *A **simple predicate** is a predicate of the form "v_1 relop v_2" where v_1 and v_2 are program variables.*

The technique assumes that all conditions with compound predicates (i.e., conjunctions or disjunctions of simple predicates) are transformed into conditions with simple predicates. If a condition (e.g., "$if(v_1)$") consists of a single variable (i.e., v_1), the technique treats the condition as "$if(v_1 == 0)$". Hence, the predicate for the condition is "$v_1 == 0$" (i.e., v_2 is 0). Transforming all predicates into simple predicates simplifies the transformation of the PDG.

3.1.2 Structural Transformation

The technique adds nodes and edges to the PDG in two cases: (1) if a node has two state components (i.e., a predicate component and a data-dependence component) or (2) if there are self-loops (i.e., nodes that are control or data dependent on themselves) in the PDG.

3.1.2.1 Transforming Nodes With Two State Components

The state of a predicate node can be characterized by both a predicate outcome and a set of dynamic data dependences. Thus, the state of a predicate node may have two state components (i.e., a predicate component and a data-dependence component). If so, the technique introduces a new node into the PDG and assigns the data-dependence component to the new state (removing it from the predicate node). The technique makes the new node the immediate successor of the predicate node's immediate predecessors, and makes the original predicate node an immediate successor of the new node. Note that the predicate node retains its connection to its immediate successors.

For example, the predicates "$i < n$" and "$v > max$" at nodes 4 and 6, respectively, in Figure 7(b), each have two state components. Predicate "$i < n$" has two state components because of the predicate computation at the node and because of its dynamic data dependences on nodes 1, 2, and 8. Predicate "$v > max$" has two state components because of the predicate computation at the node and because of its dynamic data dependences on nodes 3, 5, and 7. Figure 8(a) shows the results of the structural transformation of the PDG of *findmax* that introduces new nodes D4 and D6.

3.1.2.2 Transforming Self Loops

Loops in a program may cause the program's PDG to contain self-loops. However, self-loops are not permitted in the dependency network formalism on which PPDGs

are based. Therefore, the technique eliminates self-loops from a PDG by introducing new nodes and edges. A self-loop in a PDG may involve either a control dependence or a data dependence. If a node n is data dependent on itself, with respect to a program variable v, the technique removes the self-loop and adds a new node. The new node is made an immediate predecessor of n. The edge from the new node to n is a data-dependence edge with respect to variable v. For example, in Figure 7(b), node 8 is data dependent on itself. The self-loop is removed and a new node L8 is introduced. L8 is made the immediate predecessor of node 8 (i.e., node 8 becomes data dependent on L8 with the data-dependence variable being i). Figure 8(b) shows the result of the self-loop transformation.

If a node is control dependent on itself and the predicate at the node has two state components, the technique does not add a new node to the PDG. Because the predicate node had two state components, in the previous step it was already transformed and a node was added. The technique removes the self-loop and connects a control-dependence edge from the original predicate node to the node added in the previous step. For example, Figure 8(a) shows that node 4 is control dependent on itself. However, a new node is not added because the predicate at the node has two state components and has therefore already been transformed. Instead, the technique adds a control-dependence edge from node 4 to node D4. Figure 8(b) shows the result of this transformation.

Figure 8(b) shows the result of structurally transforming the PDG of example program *findmax* in Figure 7(b). This graph structure forms the structure of the PPDG (a dependency network).

3.1.3 State Specification

The technique models states at each of the nodes after the PDG of the program has been structurally transformed. The techniques assigns states to all statements in the

program, which we partition into predicate nodes and non-predicate nodes.

3.1.3.1 Predicate Nodes

The technique models the states at all predicate nodes in the transformed PDG using predicate outcomes. The predicate outcomes depend on how the program variables involved in the predicate computation relate to each other in terms of the relational operators (i.e., $<$, $>$, $\leq$, $\geq$, $==$, and $\neq$). The technique places the simple predicates into two categories based on the state assignments.

1. For nodes whose predicates involve primitive variables (e.g., ($v1$ *relop* $v2$) where $v1$ and $v2$ are char, int, float, or double variables) and *relop* is a relational operator, the outcomes of the predicate computation are based on how $v1$ relates to $v2$. The technique assigns $<$, $>$, $==$, and $\perp$ as the set of states to each predicate node whose operands are primitive variables. Given concrete values of $v1$ to $v2$, the technique compares the values to determine the kind of *relop* that is satisfied. The *relop* that satisfies the comparison becomes the predicate outcome or state of the predicate node. For example, Figure 8(b) represents a predicate "$v > max$" at node 6. If $v = 2$ and $max = 7$ when node 6 is executed, the predicate outcome is $<$.

2. If the variables involved in the predicate are pointers or references, the technique introduces states that model pointer or reference equality and inequality, and thus assigns the states $==$, $\neq$, and $\perp$ to the node.

3.1.3.2 Non-Predicate Nodes

The characterization of the states of non-predicate nodes that are dynamically data-dependent on other nodes is based on a data-flow modeling technique proposed by Laski and Korel [41] as a guide to program testing. Laski and Korel first define the *data environment* of a statement s.

Table 1: Nodes in transformed PDG with corresponding states.

Nodes	States
1, 2, 3, 5, L8	$\top$, $\bot$
D4	$(d_1(i), d_2(n))$, $(d_8(i), d_2(n))$, $\bot$
4, 6	$<$, $>$, $==$, $\bot$
D6	$(d_5(v), d_3(max))$, $(d_5(v), d_7(max))$, $\bot$
7	$(d_5(v))$, $\bot$
8	$(d_1(i))$, $(d_L8(i))$, $\bot$
10	$(d_3(max))$, $(d_7(max))$, $\bot$

Definition 27. *The **data environment** of a statement s is the set of variable definitions that reach s, along any paths, and are used at s.*

To more precisely model potential dynamic data flows, Laski and Korel introduced the concepts of elementary data context and data context for a statement s.

Definition 28. *An **elementary data context** of a statement s is the set of definitions that reach and are used at a given occurrence of s along some path.*

Definition 29. *The **data context** of a statement s is the set of all elementary data contexts of s.*

To illustrate, consider Figure 8(b), and suppose $d_i(x)$ denotes a definition of a variable x at node i. For node 10, the data environment is $\{d_3(max), d_7(max)\}$ because the definitions of max at nodes 3 and 7 are potentially used at node 10. The elementary data contexts of node 10 are $\{d_3(max)\}$ and $\{d_7(max)\}$ because only one of the definitions can reach node 10 at a time in an execution. The data context, which is the set of its elementary data contexts, is $\{\{d_3(max)\}, \{d_7(max)\}\}$. For node D6, the data environment and data context are $\{d_3(max), d_7(max), d_5(v)\}$ and $\{\{d_3(max), d_5(v)\}, \{d_7(max), d_5(v)\}\}$, respectively. The data contexts of nodes D4, 7, and 8 are $\{\{d_1(i), d_2(n)\}, \{d_8(i), d_2(n)\}\}$, $\{d_5(v)\}$, and $\{\{d_1(i)\}, \{d_{L8}(i)\}\}$, respectively.

The set of states for a non-predicate node that is dynamically data-dependent on other nodes corresponds to the data context of that node, augmented with the $\perp$ state. (Recall that $\perp$ means that the node was not executed in a given execution.) If a non-predicate node is not dynamically data-dependent on any node, then by default the technique assigns $\{\top\}$ as its data context. Hence, the states of the node are $\top$ and $\perp$. The state $\top$ means that during a given execution the node was executed. For example, nodes 1, 2, 3, 5, and L8 in Figure 8(b) are not dynamically data-dependent on any node. Hence, the nodes have the states $\top$ and $\perp$. Table 1 shows the nodes in the transformed PDG with their corresponding states.

3.1.4 Learning

During this step, the technique estimates the parameters of the PPDG from the set of execution data ($D = \{D_k\}_{k=0}^{n}$) generated by executing the instrumented program P' with its test suite T_P. Each $D_k \in D$ corresponds to a test case in T_P. Different kinds of execution data (e.g., coverage or trace information) might be used to estimate the parameters of the PPDG. In this work, the technique uses node-state traces. Therefore, the execution data D is the set of all node-state traces generated by executing P' with T_P.

Definition 30. *A **node-state trace**[1] is a sequence of executed nodes, along with their active states, in the transformed PDG.*

The technique uses node-state traces to estimate the parameters of the PPDG so that the PPDG will capture some of the temporal behaviors of the program. Each $D_k \in D$ is a node-state trace. A node can appear multiple times in the trace, and the states that the node assumes can be different. To learn the parameters of the PPDG, we present a batch-learning algorithm, LearnParam (Figure 9). However, the algorithm can be modified easily to an on-line learning algorithm.

[1] *We denote each node-state in the trace as "(node:state)".*

```
Input: D = {D_k}_{k=1}^n, transformed PDG
Output: PPDG
foreach D_k ∈ D do
    for j = 1 to Length(D_k) do
        if Pa(X_j) = ∅ then
            increment n(X_j = x_ji) by 1, where x_ji is the current
            state of X_j;
        else
            increment n(X_j = x_ji, Pa(X_j) = pa_ji) by 1, where x_ji is
            the current state of X_j; and where pa_ji represents the
            current state configuration of the parents of X_j;
        end
    end
end
Compute probabilities of X_j using Equations (21) and (22);
```

Figure 9: The LearnParam algorithm

3.1.4.1 Estimating Parameters of the PPDG

Learning the parameters of the PPDG consists of estimating conditional probability distributions, which are represented as tables, called *conditional probability tables* (CPTs), because the states of the nodes in the transformed PDG are discrete. Suppose $X = \{X_1, \cdots, X_n\}$ denotes the set of nodes in the transformed PDG. We denote the ith state associated with node X_j by x_{ji}, the *parents* (immediate predecessors) of a node X_j by $Pa(X_j)$, and the ith assignment of states to the parents of X_j by pa_{ji}. For a node with no parents, the technique estimates the probabilities ($p(X_j = x_{ji})$) of the nodes as

$$p(X_j = x_{ji}) = \frac{n(X_j = x_{ji})}{n(X_j)} \tag{21}$$

where $n(X_j = x_{ji})$ is the number of times node (X_j) is in state x_{ji} across all node-state traces and $n(X_j)$ is the number of times the node X_j occurs across all node-state traces. For a node with parents, the technique estimates the probabilities ($p(X_j =$

Table 2: Nodes in transformed PDG with corresponding conditional probability distributions.

Nodes	Conditional probability distributions
1, 2, 3, L8	P(1), P(2), P(3), and P(L8)
D4	P(D4 | 1, 2, 8)
4	P(4 | D4)
5	P(5 | 4)
D6	P(D6 | 3, 4, 5, 7)
6	P(6 | D6)
7	P(7 | 5, 6)
8	P(8 | 1, 4, L8)
10	P(10 | 3, 7)

$x_{ji}|Pa(X_j) = pa_{ji}))$ of the node as

$$p(X_j = x_{ji}|Pa(X_j) = pa_{ji}) = \frac{n(X_j = x_{ji}, Pa(X_j) = pa_{ji})}{n(Pa(X_j) = pa_{ji})} \tag{22}$$

where $n(X_j = x_{ji}, Pa(X_j) = pa_{ji})$ is the number of times node X_j and its parents assume a specific state configuration across all node-state traces, and $n(Pa(X_j) = pa_{ji})$ is the number of times $Pa(X_j) = pa_{ji}$ across all node-state traces. A *state configuration* is a set of states assigned to a set of nodes in the PPDG. The CPTs of the nodes must satisfy Equation 23, which means that the sum over the states of node X_j given that its parents are in a specific state configuration pa_{ji} must equal 1.0.

$$\sum_{s=x_{j1}}^{x_{jn}} P(X_j = s|Pa(X_j) = pa_{ji}) = 1.0 \tag{23}$$

*3.1.4.2 Learning Algorithm (*LearnParam*)*

Figure 9 shows LearnParam, which estimates the parameters of a PPDG. The algorithm takes as input the set of all node-state traces D (execution data), generated by executing an instrumented program P' with its test suite T_P, and the program's transformed PDG. The algorithm outputs the PPDG of the program. LearnParam traverses each $D_k \in D$ from the beginning of the node-state trace to the end, updating

Table 3: Example input for program *findmax* with corresponding node-state trace.

Input	Node-state trace
n=1, v={1}	1: $\top$, 2: $\top$, 3: $\top$, D4: $\{d_1(i), d_2(n)\}$, 4: $<$, 5: $\top$, D6: $\{d_5(v), d_3(max)\}$, 6: $>$, 7: $d_5(v)$, 8: $d_1(i)$, L8: $\top$, D4: $\{d_8(i), d_2(n)\}$, 4: $==$, 10: $d_7(max)$

the parent states of nodes and the necessary counts depending on whether a node in a trace has parents (lines 1 to 9). After LearnParam processes D, it computes the conditional probabilities of each node in the transformed PDG (line 10). Finally, it outputs the PPDG. Table 2 shows the conditional probabilities distribution representations of each node in the transformed PDG of the example program (*findmax*). For example, the conditional probability distribution for node 6 in Figure 8(b) is denoted as P(6 | D6) because node 6 is dependent on node D6. Thus, the technique estimates the probabilities of the states of node 6 given the states of its parent node D6.

3.1.4.3 Worst-Case Space and Execution Time of LearnParam

Suppose $|N|$ is the number of nodes in the transformed PDG and N_i represents any node in the transformed PDG. Suppose N_i has S_i states and it has $|K|$ parents with the *k-th* parent having P_k states. The space required to store all the CPTs is $O(\sum_{i=1}^{|N|}(S_i \times \prod_{k=1}^{|K|}(P_k)))$. Suppose the number of edges is $|E|$ then the worst-case space required to store the PPDG is $O(E + N + \sum_{i=1}^{|N|}(S_i \times \prod_{k=1}^{|K|}(P_k)))$. Suppose $|T|$ represents the total number of test cases then, the worst time required to learn the CPTs is $O(\sum_{k=1}^{|T|} D_k)$.

3.1.4.4 Learning Example

Suppose the example program *findmax* (Figure 7(a)) receives the following inputs: $(n = 1, v = \{1\})$, $(n = 2, v = \{1, -1\})$, $(n = 2, v = \{-1, 1\})$, and $(n = 1, v = \{0\})$.

Table 4: CPT of node 6.

D6	6			
	$<$	$>$	$==$	$\perp$
$(d_5(v), d_3(max))$	1/5	3/5	1/5	0.0
$(d_5(v), d_7(max))$	1/2	1/2	0.0	0.0
$\perp$	0.0	0.0	0.0	1.0

These inputs cause *findmax* to execute correctly. Table 3 shows an example node-state trace. The first column shows the inputs to *findmax*, where n is the number of inputs that *findmax* reads at line 5 and v is the set of integers input into *findmax*. The second column shows the corresponding trace. To estimate the probabilities in the conditional probability tables, LearnParam processes the traces from the beginning of the trace until the end, updating the states of nodes and their parent states.

We use the node-state trace in Table 3 to illustrate how LearnParam estimates the CPT of node 6. Note that node 6 is dependent on node D6 in the transformed PDG as shown in Figure 8(b). For the node-state trace shown in Table 3, the first occurrence of node 6 has the state "$>$" and the state of node 6's parent at that occurrence is ($d_5(v)$,$d_3(max)$). Therefore, the algorithm increments n(6="$>$", D6=($d_5(v)$,$d_3(max)$)) by 1. LearnParam continues processing the trace until it reaches the end. After all the traces have been processed, LearnParam normalizes the counts to produce the probabilities. Table 4 shows the conditional probability table for node 6. The first column shows the states of node D6 and the second column shows the states of node 6. For example, the table shows that P(6="$>$"|D6=$(d_5(v), d_3(max))$ is 3/5, which means that the probability of node 6 assuming the state "$>$" given that node D6 has assumed the state $(d_5(v), d_3(max)$ is 3/5. As Table 4 shows, the sum of the probabilities for each row in the CPT for node 6 satisfies Equation 23.

```
   Input: Node-State-Trace: {X_j : x_ji}_{j=1}^n, PPDG
   Output: Ranked Nodes
1  for j = 1 to n do
2  |  prob = p(X_j = x_ji | Pa(X_j) = pa_ji);
3  |  if prob < lowest_prob(X_j) then
4  |  |  lowest_prob(X_j) = prob;
5  |  |  index(X_j) = j;
6  |  |  configuration(X_j) = {x_ji ∪ pa_ji};
7  |  end
8  end
9  Rank nodes in ascending order, break ties using indices;
```

Figure 10: The RankCP algorithm

3.1.4.5 Fault Localization Algorithm (RankCP)

Figure 10 shows RankCP, which analyzes a single failing execution at a time, and ranks nodes in the PPDG according to how likely the nodes are to be faulty. RankCP ranks nodes based on the conditional probabilities of nodes given the states of their parent nodes (i.e., $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$), which reflect how the parents influence their children.

The hypothesis behind RankCP is that RankCP will often detect the first place in a failing execution where a node (X_j) assumes an unusual state, given the states of its parents, thus indicating a possible cause of the failure. RankCP ranks a node X_j that has a state whose probability is low, given the states of X_j's parents, as highly suspicious. The choice of this conditional probability as an inverse measure of suspiciousness is based on preliminary studies we conducted that showed that faults tend to be associated with low probability nodes. The intuition here is that because the PPDG is trained with only passing executions, states that are executed mostly in failing executions will tend have very low probabilities.

For a given program, RankCP inputs its PPDG and a node-state trace generated

by a failing execution, and it returns a list of nodes ranked from most suspicious to least suspicious. Each node is also associated with a node-parent state configuration. RankCP processes a trace from beginning to end. As it processes the trace (line 1), it computes the conditional probability of a node's current state (x_{ji}) given the current state configuration (pa_{ji}) of its parents (i.e., $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$) (line 2). Then, RankCP records for each node, the lowest value *lowest_prob* of this probability (lines 3 and 4). Note that RankCP computes $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$ using the conditional probability table for node X_j. RankCP also keeps track of the index of a node in the trace in the *index* variable (line 5). RankCP associates a node-parent state configuration with a node using the *configuration* variable (line 6). After RankCP has processed the trace, it ranks the nodes by their *lowest_prob* values, and if two nodes have the same *lowest_prob* values, the algorithm ranks the node with the lower *index* value higher (line 10). The algorithm ranks lower indices higher because the lower indices potentially indicate where the deviation from normal program behavior occurred, hence most likely to be faulty. Finally RankCP ouputs the ranked nodes.

To test the hypothesis underlying RankCP, in Study 1 of Section 3.2.2.2, we compare RankCP to existing fault-localization techniques. In Study 2 of Section 3.2.2.3 and Study 4 of Section 3.2.4.2, we compare RankCP to two variants that respectively use marginal and joint probabilities, instead of conditional probabilities, to rank nodes.

3.1.4.6 Worst-Case Space and Execution Time of RankCP

This section presents the worst-case space and execution time for the RankCP algorithm. Suppose $|N|$ is the number of nodes in the node-state trace and R is the cost of sorting the nodes in descending order of probabilities. The execution-time cost of RankCP is $(|2N| + R)$ with complexity being $O(|N| + R)$. The space cost of

RankCP is $O(|2N|)$ because the algorithm needs to store the N sorted nodes and their probabilities. Therefore the space complexity of RankCP is $O(N)$.

3.2 Empirical Studies

In this Section, we demonstrate the importance of combining program dependences with statistical information by evaluating the effectiveness of the PPDG for fault localization. To evaluate the PPDG, we implemented it and performed several fault-localization studies. This section first describes the implementation of the technique and then presents the fault-localization studies.

3.2.1 Implementation

We used the CIL framework [50] to analyze the source files of C programs. CIL analyzes ANSI C programs, and it provides tools to compute the control-flow graph and to extract data-flow information from C programs. The implementation uses this control-flow graph and the data-flow information to compute the PDG of the C programs. CIL also transforms all conditions that contain compound predicates (i.e., conjunctions or disjunctions of simple predicates) into conditions with simple predicates. If a predicate in a condition consists of a function call, CIL evaluates the function separately and stores the return value in a temporary variable, which is then used in the predicate.

The implementation uses CIL to instrument the source files of the C programs. For each statement, a probe is inserted to capture the data flows from statement to statement. For each simple predicate, a probe is inserted to capture the values of the variables used in the predicate. The implementation uses the values of the variables to compute the predicate outcomes. Note that information about the new nodes in the PPDG are not in the original trace. The implementation inserts the node information in the trace and computes predicate outcomes on-the-fly when processing the trace during the learning phase.

Table 5: Subjects used for empirical studies.

Program	Faulty versions	LOC	PPDG size	Test cases	Description
Print-tokens	7	472	930	4130	lexical analyzer
Print-tokens2	10	399	290	4115	lexical analyzer
Replace	32	512	397	5542	pattern replacement
Schedule	9	292	201	2710	priority scheduler
Schedule2	10	301	212	2650	priority scheduler
Tcas	41	141	130	1608	altitude separation
Tot-info	23	440	252	1052	information measure

For this implementation, we did not perform precise pointer analysis. Thus, the data-flow information the implementation computes is an under-approximation of the actual data flow in the program. We implemented all the algorithms used to construct the PPDG in the `Objective Caml` language.

3.2.2 Fault Localization

To evaluate the effectiveness and efficiency of the PPDG when applied to the fault-localization problem, We performed two fault-localization studies (presented in Sections 3.2.2.2 and 3.2.2.3 and one efficiency study (presented in Section 3.2.3). To evaluate the effectiveness of the PPDG on larger software subjects, we also performed a scalability case study (presented in Section 3.2.4). For the fault-localization studies, we evaluated `RankCP`'s hypothesis by comparing it to other fault-localization techniques (Study 1 in Section 3.2.2.2). We also evaluated the hypothesis by comparing RankCP to a ranking algorithm based on marginal probabilities (RankM) and to an algorithm based on joint probabilities (RankJ) (Study 2 in Section 3.2.2.3 and Study 4 of Section 3.2.4).

3.2.2.1 Empirical Setup

We used the *Siemens suite* [33] as the subject programs for Studies 1, 2, and 3. This set of subjects has been used often to study the effectiveness of fault-localization techniques. Table 5 shows the characteristics of the seven Siemens programs: the

name of the program, the number of faulty versions, the number of lines of code (LOC), the size of the PPDG in terms of its nodes, the number of test cases, and a description of the program. There are 132 faulty versions in total and each program is associated with a matrix that indicates which test cases pass and which test cases fail. Each faulty version has exactly one fault.

For the experiments, we omitted eight versions: versions 8, 14, and 32 of Replace, versions 4 and 6 of Print-tokens, version 9 of Schedule, version 9 of Schedule2, and version 38 of Tcas. The versions were eliminated because (1) there were no syntactic differences between the C file of the correct version and the faulty versions of the program (e.g., change in header file), (2) no traces could be gathered because the faulty versions had segmentation faults when executed on their test suite, or (3) none of the test cases failed when executed on the faulty version of the program.

For fault localization, the technique builds a PPDG for each faulty version of the program. The technique uses the traces of passing test cases to estimate the parameters of the PPDG. Using passing test cases enables the PPDG to capture the correct behaviors of the program. After building the PPDG, we ran the RankCP algorithm on the trace of each failing test case. (Recall that RankCP analyzes a single trace at a time.)

3.2.2.2 Study 1: Effectiveness of RankCP Compared to Other Techniques

The goal of this study is to compare the effectiveness of RankCP to existing fault-localization techniques: SOBER [43], Tarantula [38], and Cause Transitions (CT) [16]. We obtained the fault localization results for Tarantula, CT, and SOBER from published papers [16, 37, 43].

To compare the effectiveness of RankCP to the other fault-localization techniques,

Table 6: Percentage of faults found to the percentage of code examined.

Score	RankCP-best	RankCP-worst	RankCP-median	Tarantula	CT	SOBER
0-1%	41.94	17.74	30.65	13.93	4.65	8.46
1-10%	31.45	27.42	33.06	41.80	21.71	43.84
10-20%	13.71	25.81	17.74	5.74	11.63	21.54
20-30%	2.42	4.84	6.45	9.84	13.18	3.85
30-40%	2.42	4.84	3.23	8.20	1.55	4.62
40-50%	5.65	8.06	2.42	7.38	6.98	0.77
50-60%	1.61	2.42	3.23	0.82	3.10	0.77
60-70%	0.0	5.65	2.42	0.82	7.75	2.31
70-80%	0.8	2.42	0.8	4.10	4.65	2.31
80-90%	0.0	0.81	0.0	7.38	6.98	2.31
90-100%	0.0	0.0	0.0	0.00	17.83	9.23

we use the metric *Score* used by References [16, 37, 60]. *Score* represents the percentage of nodes [2] that must be examined by the developer to find the fault, assuming the developer starts from the highest ranked suspicious node and examines nodes in decreasing order of suspiciousness until the faulty node is found. We use percentages instead of raw counts because we would like to present the results of the various subjects on the same graph. However, raw counts can be computed from the percentages.

For example, a *Score* range of $0\% - 1\%$ means that the developer needs to examine less than 1% of the code to find the fault. The *score* is computed as

$$\text{Score} = \frac{\mid \text{N} \mid}{\mid \text{PPDG} \mid} \times 100 \tag{24}$$

where $\mid$ N $\mid$ is the number of nodes examined to find faulty node and $\mid$ PPDG $\mid$ is the number of nodes in the PPDG. Because RankCP analyzes a single failing trace at a time, we show its best, worst, and median case performances on the set of failing test cases for each faulty version. For example, for version 31 of Replace, there are 210 failing test cases. For 95.3% (i.e., approximately 200) of the failing test cases,

[2] We assume that each node corresponds to a single statement in a program.

less than 1% of the code must be examined to find the faulty statement. For the remaining 4.7% (i.e., approximately 10) of the failing test cases, between 1% and 10% (exclusive) of the code must be examined. This result implies the best *score* range for RankCP for the faulty version is $0\% - 1\%$ and the worst is $1\% - 10\%$.

Table 6 shows the percentage of faults found at each *score* range for each of the techniques. RankCP-best, RankCP-worst, and RankCP-median represent the best, worst, and median performance of RankCP, respectively. Under RankCP-best, for 41.94% of the faulty versions, less than 1% of the program must be examined to locate the faulty statement. Under RankCP-worst, for 17.74% of the faulty versions, less than 1% of the program must be examined to find the faulty statement. For RankCP-median, for 30.65% of the faulty versions, less than 1% of the code has to be examined to find the faulty statement. When less than 1% of the code must be examined under RankCP-best, the technique is approximately 9, 5, and 3 times more effective than CT, SOBER, and Tarantula, respectively. Under RankCP-worst, the technique is approximately 4, 2 and 1.2 times more effective than CT , SOBER, and Tarantula respectively.

Figure 11 shows the cumulative results of Table 6. The horizontal axis represents the percentage of a program's statements that must be examined to find the fault it contains and the vertical axis represents the percentage of faulty versions that are found given a *score* on the horizontal axis. Note that the vertical axis can also be interpreted as the percentage of faults found if no more than a given percentage of the program is examined. The legend lists the fault-localization techniques. Figure 11 shows that RankCP-best and RankCP-worst are more effective than CT. This is significant because RankCP and CT both analyze a single failing execution at a time.

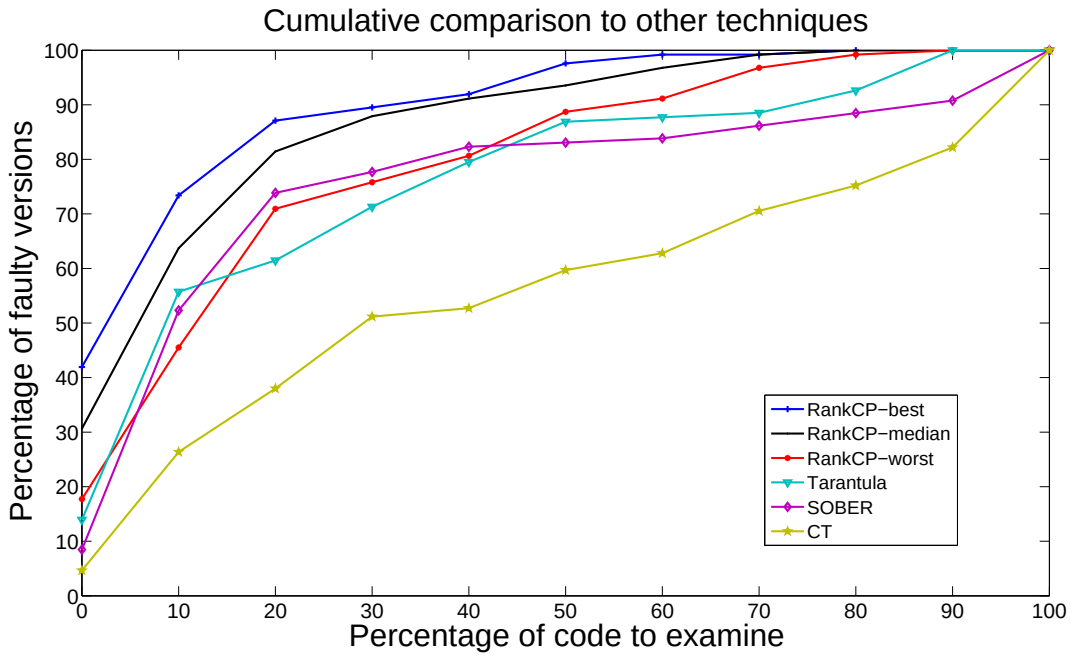


Figure 11: Cumulative comparison with other techniques on the Siemens subjects.

3.2.2.3 Study 2: Effectiveness of RankCP Compared to Other Probability Rankings

The goal of this study is to compare RankCP's effectiveness to RankM and RankJ algorithms that are based on marginal and joint probabilities, respectively. Intuitively, RankM indicates the first place in a failed execution where a node assumes a suspicious state, and RankJ indicates the first place where a node and its parents assume a suspicious configuration of states. We computed the marginal and joint probabilities directly from the node-state trace information generated by the passing executions. We computed the joint probability estimates for nodes and their parents, because we wanted to capture their joint behavior. For a node (X_j) in a given state x_{ji}, we estimated its marginal probability as

$$p(X_j = x_{ji}) = \frac{n(X_j = x_{ji})}{n(X_j)} \tag{25}$$

where $n(X_j = x_{ji})$ is the number of times node X_j is in state x_{ji} and $n(X_j)$ is the number of times node X_j occurs across all node-state traces. We also estimated joint probabilities as

$$p(X_j = x_{ji}, Pa(X_j) = pa_{ji}) = \frac{n(X_j = x_{ji}, Pa(X_j) = pa_{ji})}{n(X_j, Pa(X_j))} \tag{26}$$

where $n(X_j = x_{ji}, Pa(X_j) = pa_{ji})$ is the number of times that the node X_j and its parents assume a specific state configuration and $n(X_j, Pa(X_j))$ is the number times that the node and its parents occur across all the node-state traces.

Figures 12, 13, and 14 show the results of the comparisons between RankCP, RankM, and RankJ. Figures 12, 13, and 14 show the best, median, and worst performances of the ranking approaches respectively. We used equation 24 to compute the *scores* for each approach. As before, the horizontal axes represent the percentage of program statements that must be examined to find the fault and the vertical axes represent the percentage of faulty versions that are found given a *score* on the horizontal axes.

As the figures show, RankCP performed best, followed by RankM, with RankJ being the least effective. RankCP performed best apparently because it often indicates the first statement where something unusual happened. It may be that the *score* metric is somewhat unfair to RankJ. With RankCP and RankM, the developer is expected to examine one node at a time. But with RankJ the developer is expected to examine a node and its parents at the same time. RankJ provides more contextual information than RankCP and RankM, but the *score* metric does not account for this. Hence, RankJ seems to be least effective.

3.2.2.4 Relationship between `RankCP`, `RankM`, and `RankJ`

In this section, we explore the relationship between `RankCP`, `RankM`, and `RankJ`. Suppose node X_j is in state x_{ji} and its parents $\mathrm{Pa}(X_j)$ are in state configuration pa_{ji}.

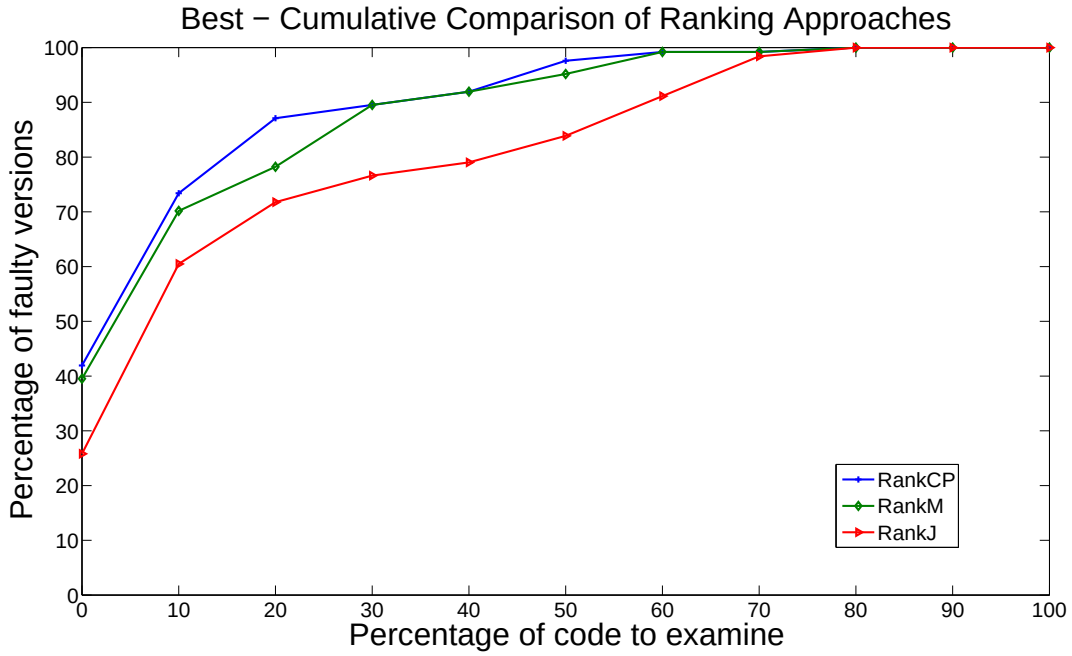


Figure 12: Best cumulative comparison of RankCP, RankM, and RankJ on the Siemens subjects.

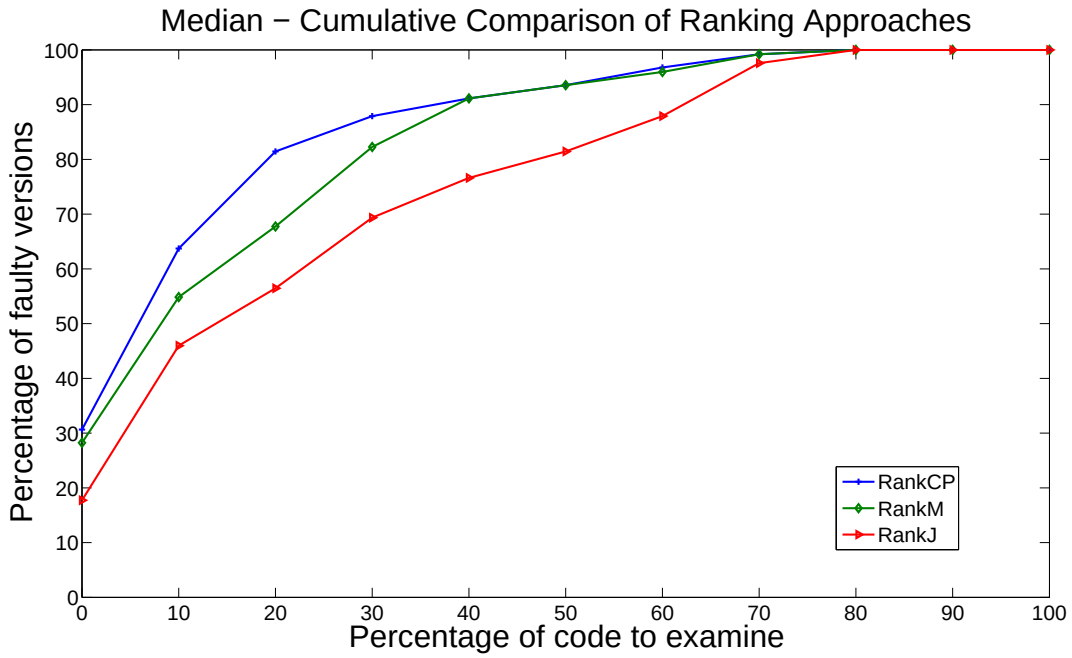


Figure 13: Median cumulative comparison of RankCP, RankM, and RankJ on the Siemens subjects.

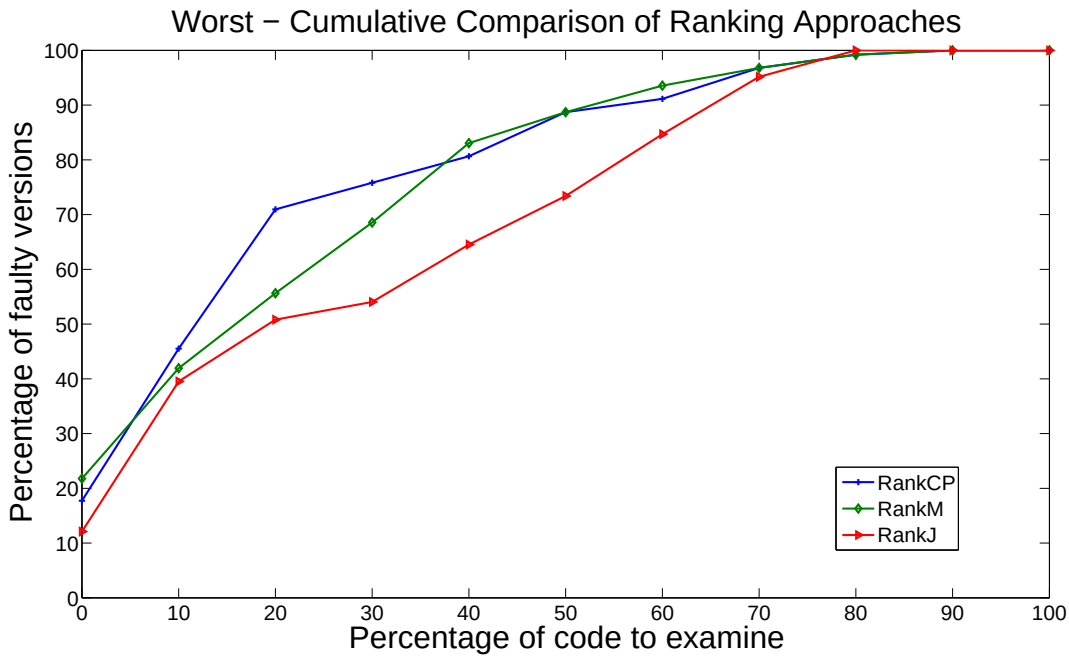


Figure 14: Worst cumulative comparison of RankCP, RankM, and RankJ on the Siemens subjects.

Equation 28 shows the relationship between `RankCP` and `RankJ`.

$$\begin{aligned} RankCP &= \frac{n(X_j = x_{ji}, Pa(X_j) = pa_{ji})}{n(Pa(X_j) = pa_{ji})} \\ &= \frac{n(X_j = x_{ji}, Pa(X_j) = pa_{ji})}{n(Pa(X_j) = pa_{ji})} \times \frac{n(X_j, Pa(X_j))}{n(X_j, Pa(X_j))} \\ &= RankJ \times \frac{n(X_j, Pa(X_j))}{n(Pa(X_j) = pa_{ji})} \end{aligned} \tag{27}$$

$$RankCP = RankJ \times \frac{n(X_j, Pa(X_j))}{n(Pa(X_j) = pa_{ji})} \tag{28}$$

From Equation 28, $n(X_j, Pa(X_j)) = n(X_j)$. Therefore, Equation 28 is

$$RankCP = RankJ \times \frac{n(X_j)}{n(Pa(X_j) = pa_{ji})}$$

$$RankCP = RankJ \times \frac{n(X_j)}{n(Pa(X_j) = pa_{ji})} \times \frac{n(X_j = x_{ji})}{n(X_j = x_{ji})}$$

$$RankCP = \frac{RankJ}{RankM} \times \frac{n(X_j = x_{ji})}{n(Pa(X_j = pa_{ji}))} \tag{29}$$

Equation 29 shows that `RankCP` is directly propotional to RankJ and inversely proportional to `RankM`.

3.2.3 Study 3: Efficiency of the Technique

The goal of this study is to evaluate the efficiency of the technique and to compare it to the efficiency of other fault-localization techniques. We conducted efficiency experiments on a `3.2 GHz Intel Pentium-4 PC` with `2 GB` of memory. We obtained timings for Tarantula and CT from published results [16, 37]. Note that because of differences in hardware, the comparing the various techniques may be difficult, however, the timings still show how efficient our approach can be.

Table 7 summarizes the results of the study. The columns show the programs, the average time taken to process all traces and build the PPDG, the average computation time taken by RankCP to analyze a single failing execution, the computation time of Tarantula, and the average computation time of CT, respectively. All the timings are in seconds.

As the results show, the time required to process all traces and build the PPDG, along with the computation time required by RankCP to localize the fault in a given failing execution, is less than the computation time of CT. For example, for Replace, the technique requires, on average, less than 6 minutes to process all traces and build the PPDG. The computation time for CT for Replace is approximately 1 hour. However, the computation time for RankCP is, on average, 0.0327 seconds. The timings are signficant because both CT and RankCP analyze a single failing execution at a time. Of all the techniques, Tarantula is the most efficient. Note that both RankCP and Tarantula take milliseconds to finish their fault-localization analysis. Therefore, in practice the improvement in the efficiency of Tarantula might not be significant from a user's perspective when compared to RankCP. Note that none of the implementations have been optimized. Furthermore, differences in computing

Table 7: Efficiency of technique in seconds.

Program	PPDG (Process traces & build)	RankCP (Computation time)	Tarantula (Computation time)	CT (Computation time)
print_tokens	846.5	0.2176	0.0040	2590.1
print_tokens2	243.6	0.0574	0.0037	6556.5
replace	335.3	0.0327	0.0063	3588.9
schedule	77.3	0.0082	0.0032	1909.3
schedule2	199.5	0.0217	0.0030	7741.2
tcas	1.7	0.0003	0.0025	184.8
tot_info	97.6	0.0605	0.0031	521.4

environments (e.g., operating systems and programming languages) might affect the results. Therefore, the efficiency results should not be viewed as definitive.

3.2.4 Scalability of PPDG

To further explore RankCP's hypothesis and also the scalability of the PPDG on larger subjects, we applied the technique to three additional software subjects: Sed, Grep and Space, and we compared it to RankM and RankJ.

3.2.4.1 Empirical Setup

For the scalability study, we used Sed, Grep and Space. we obtained the subjects from the Software-artifact Infrastructure Repository [19]. Each of the three subjects comes with a fault matrix that indicates the test cases that pass and the test cases that fail.

Sed is a stream editing utility for the Unix Operating System platform that contains 14K lines of code. Sed has seven versions, each with a number of seeded faults that can be activated individually. For the study, we randomly chose versions 4, 5, and 6. We activated all faults in the versions individually, which resulted in 14 faulty versions of Sed. Each faulty version had a single fault. Out of the 14 faulty versions, we omitted three versions because none of the test cases failed on them. The number

of test cases was between 360 and 370 for each of the versions.

Grep is a text-search utility on the Unix platform that contains about 10K lines of code and 146 procedures. Grep has five versions seeded with faults and 470 test cases. We used three versions of Grep because many of the versions had no failing executions when their seeded faults were activated.

Space is a software subject developed by the European Space Agency that has 6K lines of code with 136 procedures and 13585 test cases. Space also comes with different types of test suites. We used testplans.bigcov, which contains 1000 test suites that achieve branch coverage. We randomly picked a test suite from testplans.bigcov and executed it on Space. The number of test cases in the test suites selected ranged from 4312 to 4407. Space also has a total of 38 fault-seeded versions but we used 26 versions for this study because the selected test suites did not expose the faults in the versions.

For the scalability study, the total number of versions (i.e., of Sed, Grep, and Space) we used was 40. Also for the study, the technique builds a PPDG for each faulty version of the program. The technique uses the traces of passing test cases to estimate the parameters of the PPDG. After building the PPDG, we ran the RankCP algorithm on the trace of each failing test case. We also computed the results for RankM and RankJ for each failing test case. (Recall that RankCP, RankM and RankJ analyzes a single trace at a time.)

3.2.4.2 Study 4: Effectiveness of Technique on Larger Subjects

The goal of this study is to evaluate the scalability of the PPDG on larger subjects and to further investigate RankCP's hypothesis when compared to RankM and RankJ.

Tables 8, 9, and 10 show the results of the study for Sed, Grep, and Space, respectively. The first columns (Faulty Version) for Sed and Grep show which version of the subjects was used and which fault was activated. For example, V6-F1 for

Sed means version 6 of Sed was used with the first fault activated. The column labeled MBT (model building time) gives the time it took to build the PPDG; the columns labeled RankCP-best, RankCP-worst, and RankCP-median give the best, worst, and median fault-localization results, respectively, for the faulty versions. We also computed the best, worst, and median outcomes for RankM and RankJ. Tables 8, 9, and 10 also show the number of nodes in the PPDG that must be examined to find the faulty statement instead of the percentage of the program that must be examined. Using the number-of-nodes metric gives us a detailed view of the effectiveness of the ranking approaches.

Table 8 shows the effectiveness of RankCP, RankM, and RankJ on Sed. As the table indicates, the ranking approaches were effective in localizing the faults in some faulty versions of Sed that we examined but not in others. For example, for V5-F2 under RankCP-best only the top two nodes must be examined to find the faulty node. For the best of the RankM and RankJ rankings, only two and three nodes must be examined, respectively. Under RankCP-worst and RankCP-median, only 15 and 4 nodes must be examined, respectively, to find the fault. Also under the worst and median performances for RankM rankings, only 23 and 8 nodes, respectively, must be examined. For the worst and median RankJ rankings, only 45 and 18 nodes, respectively, must be examined to find the faulty node. However, for some versions none of the ranking methods are effective. For example, for V6-F6, the developer must examine 2093 and 2483 nodes under RankCP-best and RankCP-worst, respectively. For RankM, the best and worst results are 2215 and 2601 respectively. For RankJ, the best and worst are 2506 and 2669, respectively.

As Table 9 shows, all three ranking approaches were effective in localizing the faults in some versions of Grep. For example, for V3-F10 under RankCP-best only 14 nodes has to be examined to find the faulty node. For the best of the RankM and RankJ rankings, only 5 nodes and 73 nodes has to be examined respectively.

Table 8: Results of the scalability case study on the Sed subject.

Faulty Version	MBT (seconds)	PPDG Size	RankCP-best	RankCP-worst	RankCP-median	RankM-best	RankM-worst	RankM-median	RankJ-best	RankJ-worst	RankJ-median
V4 - F2	1779.91	7049	2	2	2	2	2	2	3	3	3
V5 - F1	713.40	9137	10	429	20	15	353	21	196	1004	471
V5 - F2	723.02	9138	2	15	4	2	23	8	3	45	18
V5 - F3	745.71	9137	264	1370	429	389	1505	542	514	2014	699
V5 - F4	750.13	9138	317	318	317	417	477	417	459	526	459
V6 - F1	614.14	9142	1	143	4	1	238	1	1	334	14
V6 - F2	330.81	9138	7	2941	2015	2	3072	2219	16	1996	1042
V6 - F3	334.24	9143	7	2940	2015	2	3072	2143	16	1964	1091
V6 - F4	735.62	9142	7	7	7	4	4	4	18	18	18
V6 - F5	569.43	9142	1983	2702	2237	2113	2855	2377	2	830	32
V6 - F6	798.47	9137	2093	2483	2247	2215	2601	2366	2506	2889	2669

Table 9: Results of the scalability case study on the Grep subject.

Faulty Version	MBT (seconds)	PPDG Size	RankCP-best	RankCP-worst	RankCP-median	RankM-best	RankM-worst	RankM-median	RankJ-best	RankJ-worst	RankJ-median
V3 - F10	2874.3	9801	14	26	72	5	18	9	73	342	184
V3 - F18	3261.1	9801	573	596	590	1048	1081	1059	1996	2057	2010
V4 - F12	2537.1	9839	97	153	124	88	104	96	248	481	368

Table 10: Results of the scalability case study on the Space subject.

Faulty Version	MBT (seconds)	PPDG Size	RankCP-best	RankCP-worst	RankCP-median	RankM-best	RankM-worst	RankM-median	RankJ-best	RankJ-worst	RankJ-median
V10	532.06	4211	11	263	193	91	138	108	239	402	297
V11	538.63	4211	1301	1775	1458	1371	1824	1526	1462	1911	1640
V12	594.65	4211	1516	1766	1573	1593	1821	1630	131	240	138
V13	541.95	4211	9	39	11	8	78	10	11	246	14
V14	493.04	4210	962	1401	1071	972	1411	1081	1002	1429	1102
V15	385.08	4210	1036	1486	1146	1046	1496	1156	1077	1513	1177
V16	563.36	4211	10	1837	61	7	1871	58	29	1968	105
V17	576.26	4210	106	177	130	253	357	284	518	747	586
V18	593.95	4211	1524	1755	1554	83	91	88	237	270	256
V19	524.86	4213	1	4	1	1	4	1	32	80	47
V20	587.17	4211	1541	1887	1664	1568	1914	1691	8	26	26
V21	578.61	4211	1541	1887	1635	1568	1914	1662	6	24	24
V22	593.53	4206	1	1	1	1	1	1	1	1	1
V23	569.33	4211	2	47	20	61	83	69	5	114	61
V24	555.83	4196	50	171	101	86	316	202	5	545	370
V25	345.56	4209	1	1848	1263	63	1848	1263	1	1890	1281
V26	447.87	4211	886	1155	945	886	1215	945	911	1452	965
V27	613.35	4211	1694	1848	1808	73	106	96	201	340	317
V28	243.80	4211	932	1311	1017	971	1348	1057	215	390	263
V29	564.61	4211	242	328	271	303	394	330	484	644	526
V30	82.34	4211	25	1216	42	23	1216	34	45	1224	61
V31	514.4	4211	24	30	24	60	68	60	134	150	134
V33	592.87	4211	1587	1662	1632	1633	1708	1678	1727	1823	1774
V34	0.024	4211	10	10	10	10	10	10	10	10	10
V35	584.81	4203	1158	1678	1370	61	1678	1369	207	1722	1406
V36	596.33	4211	30	436	433	40	436	433	94	441	434

With RankCP-worst and RankCP-median, only 72 and 26 nodes must be examined, respectively, to find the fault. Also with the worst and median outcomes for RankM rankings, only 18 and 9 nodes have to be examined. For the worst and median RankJ rankings, only 342 and 184 nodes have to be examined to find the faulty node. However, for some versions none of the ranking methods were effective (e.g., V3-F18).

Table 10 shows the results of the study for Space. As the table shows, the ranking approaches were again effective in localizing the faults in some faulty versions of Space. For example, for V19 with RankCP's best, worst, and median outcomes, only 1, 4, and 1 nodes have to be examined respectively, to find the faulty node. For V19 under RankM's best, worst, and median performances only 1, 4, and 1 nodes have to examined respectively to find the faulty node. Finally, with RankJ's best, worst, and median performances 32, 80, and 47 nodes have to examined respectively, to find the faulty node.

We also compared the cumulative results for RankCP, RankM, and RankJ. For this comparison, we used the percentage of code examined instead of the number of nodes examined. Figures 15, 16, and 17 show the best, median, and worst cumulative comparisons between RankCP, RankM, and RankJ. For the three figures (i.e., 15, 16, and 17), the horizontal axis represents the percentage of a program's statements that must be examined to find the fault it contains and the vertical axis represents the percentage of faulty versions that are found given a *score* value on the horizontal axis. The *score* was computed using Equation 24. Note that the vertical axis can also be interpreted as the percentage of faults found if no more than a given percentage of the program is examined. Overall, RankJ performed better than RankCP and RankM. However, RankCP performed better than RankM and RankJ between 0% and 1% in all three figures. Also note that the best, worst, and median cumulative performances of RankM were better than RankCP but not by a large margin.

The MBT column of Tables 8, 9, and 10 show that the technique can scale to larger

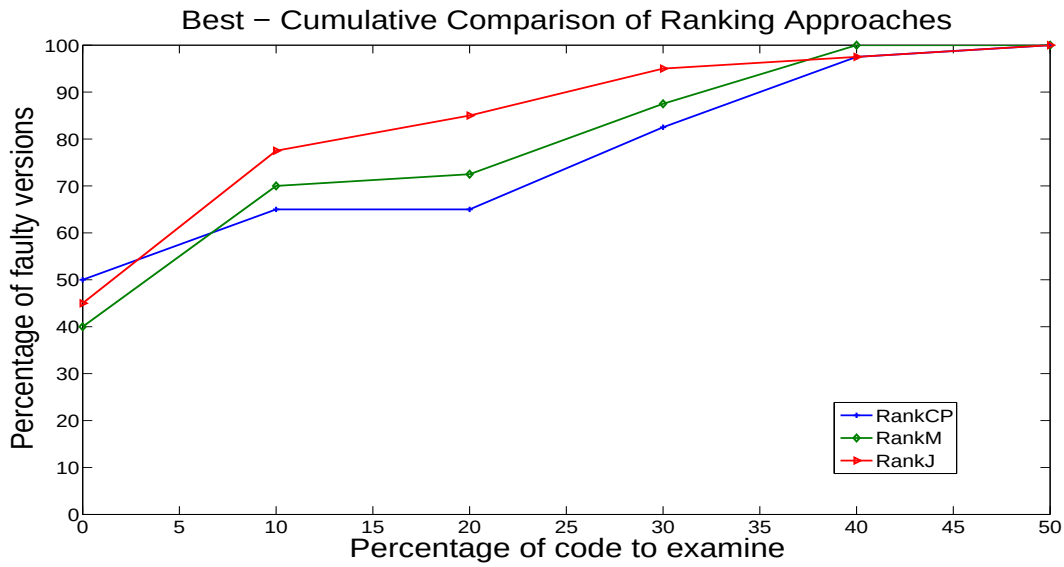


Figure 15: Best cumulative comparison of RankCP, RankM, and RankJ on the scalability subjects.

programs—the technique required less than an hour to build the PPDG for each faulty version of Sed, Grep, and Space. The scalability studies also provide preliminary evidence that rankings based on the joint probability of a collection of nodes that are connected in the PDG has the potential of being more useful than rankings based on only single nodes. Ranking based on joint probabilities can potentially be more useful because the rankings automatically provide contextual information, which are useful to programmers during fault localization.

3.2.5 Threats to Validity

There are three main types of validity threats that affect the studies: internal, external, and construct.

Threats to internal validity concern factors that affect dependent variables without the researchers' knowledge. There is the possibility that there might be errors in the implementation (specifically, the process of generating a PPDG) that might affect the experimental results. To address potential errors in the generation of the PPDG, we compared manually generated PPDGs of smaller subjects to their PPDGs generated

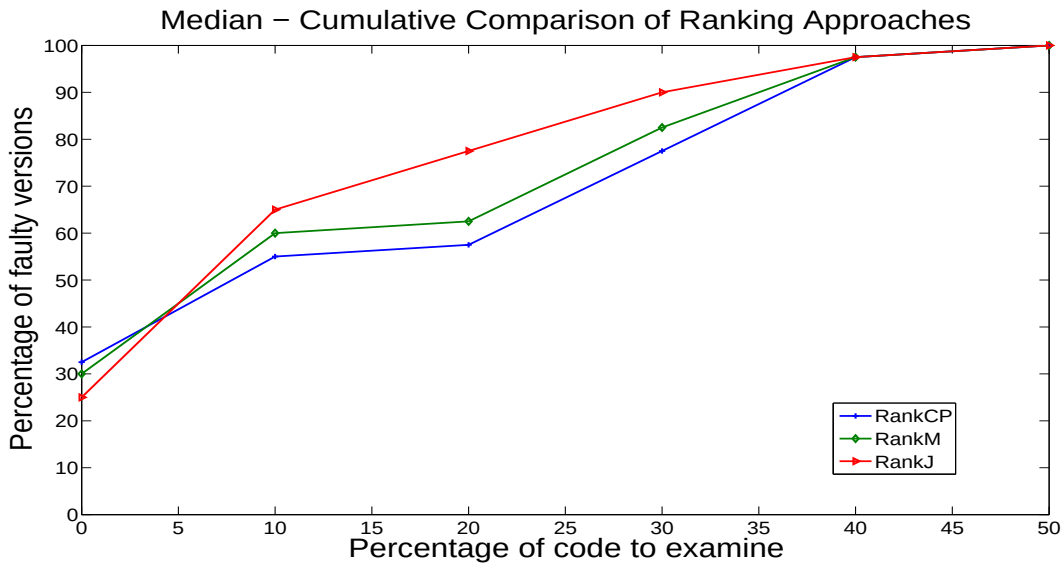


Figure 16: Median cumulative comparison of RankCP, RankM, and RankJ on the scalability subjects.

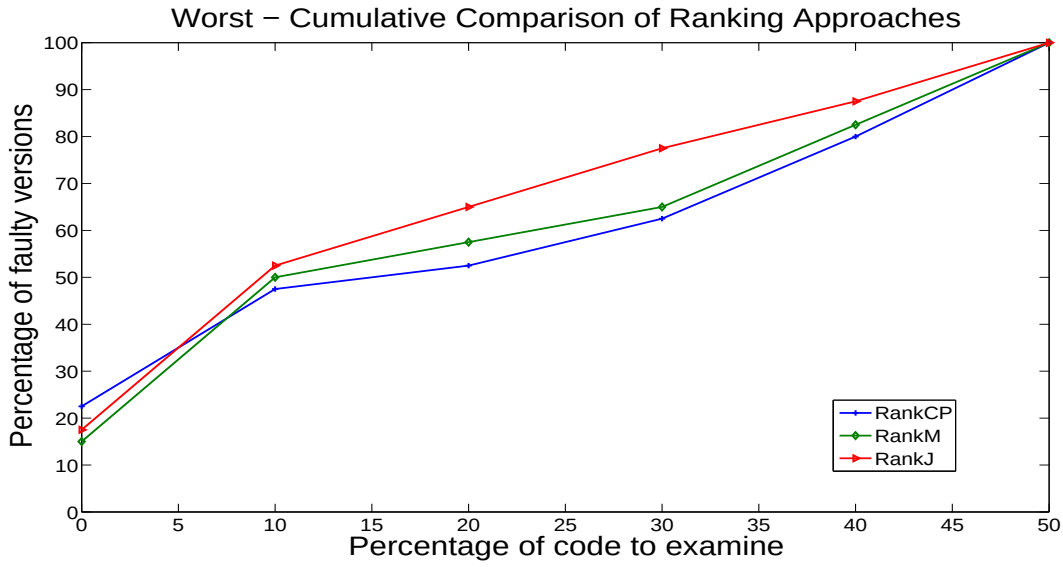


Figure 17: Worst cumulative comparison of RankCP, RankM, and RankJ on the scalability subjects.

automatically by the technique to ensure that the PPDGs match (which they did).

Another threat to internal validity concerns the potential for errors in the experimental results based on the implemented algorithms in the approach. To address the potential for errors in the experimental results, we manually checked the behavior of `RankCP`, `RankM`, and `RankCP` on the smaller subjects, and compared the algorithm's results to manually computed results to ensure that they matched (which they did).

Threats to external validity occur when the results of the experiments cannot be generalized. We performed the experiments and case studies on the Siemens suite and several versions of the Sed, Grep, and Space software. Thus, we cannot claim that the effectiveness of the approach can be generalized to other software subjects. However, the Siemens suite has been used in many fault-localization studies and thus, it provided a way to compare the technique to previous approaches. Also, Sed, Grep, and Space are real programs, and thus, the studies demonstrate to some extent that the technique can be effective on real programs. Clearly, additional studies are needed to determine how well the approach can be generalized. We discuss why the technique was not successful in Section 3.2.6.

Threats to construct validity concern the appropriateness of the metrics used in the evaluation. We used the *score* metric to determine the effectiveness of the fault-localization results of `RankCP`, `RankM`, and `RankJ` because it is the metric used by all other fault-localization techniques. The `score` metric is essentially a ranking-based metric. However, it is difficult to determine whether it conforms to the way in which programmers perform fault localization. For example, Parnin and Orso [52] demonstrated to some extent that some developers do not debug using this top-down based ranking approach. However, the studies are inconclusive because the number of developers used were statistically insignificant and also they did not take into account other exogenous factors such as background of the developers. More extensive studies are required to determine the appropriateness of the `score` metric

for evaluating fault-localization techniques.

3.2.6 Discussion of the PPDG and RankCP

As the empirical results show, the effectiveness of fault localization can be improved by combining control dependences, data dependences, and some state information with statistical information. However, the PPDG model has several limitations.

The first limitation is that the PPDG is a probabilistic graphical model of a program and therefore fault-localization algorithms such as RankCP that work on the PPDG can have limitations. The algorithms can be limited because to compute the causal effects of statements on program failure requires a causal graph, which is a directed acyclic graph [53]. However, converting the PPDG into an acyclic graph may result in loss of dependence information, which is likely to affect algorithms that use the PPDG for fault localization. Also the RankCP algorithm is an associative algorithm: it finds the program entity most associated with failure. The main issue here is that the problem of fault localization is a causal problem and as such causal algorithms are required to operate on the PPDG. Using causal algorithms can significantly improve the fault-localization results.

A second limitation is that it is difficult to scale the PPDG to large software systems (i.e., software with thousands of statements). The limitation occurs because the sizes of the CPTs of nodes can be very large depending on the number of predecessor nodes a node has in the program dependence graph and the state abstraction associated with each of the nodes.

A third limitation that is specific to the current PPDG is that because local program states are abstracted to obtain CPTs of manageable size, the resulting statistical dependences between nodes may not correspond exactly to the dependences in the PDG. This lack of correspondence could adversely affect applications of the PPDG, particularly those involving more complex inference algorithms than those employed

in the case studies.

A fourth limitation of the PPDG that is shared with all statistically-based fault-localization techniques is its reliance on the test suite. The probability estimates used in constructing a PPDG are based on the test suite and therefore, reflect the probability distribution from which it was selected or generated. For example, if the test cases were captured in the field, the CPTs reflect operational node-state probabilities. The effect of the test-suite on the effectiveness of fault localization has received little study, although Yu and colleagues [71] found that when performing fault localization in conjunction with test-suite reduction, a stratified, vector-based form of test-suite reduction was superior to statement-based reduction. The accuracy of the probability estimates used in a PPDG also depends on the effective sample sizes used to compute them. If the sample size for a particular node is too small, the probabilities of its states cannot be estimated accurately. Consequently, suspiciousness measures derived from those estimates may be misleading. One of the features of the PPDG is that it requires a single failing execution and multiple passing executions and in practice many passing executions can be obtained.

Finally, the results shows the importance of combining statistical information with program dependences and program states. However, combining program-analysis information with statistical information is not sufficient for finding the causes of program failures. In the next chapter, we present a causal framework for fault localization that utilizes the information in the PPDG (although in a different way) to find the causes of program failures.

CHAPTER 4

CAUSAL FRAMEWORK FOR PROGRAMS

The fault-localization problem is a causal problem because we seek to find the program element or elements (e.g., statement(s), function(s)) that *caused* the program to fail. According to the causal literature, one of the ideal ways to search for the cause of an event is to perform a randomized experiment [27, 46]. This randomized-experiment approach was first proposed by Fisher in his book "The Design of Experiments" [23], and it has formed the basis of further work in causal analysis in the statistical literature. Ideally to search for the cause of program failures, experiments should be performed on programs through the manipulation of program elements. Experimental manipulation of program elements provides concrete evidence as to whether a particular program element is responsible for the program's failure. State-altering techniques take this experimental approach, however, as we discussed in Chapter 1, the experimental approach is not always feasible and has serious limitations.

In this Chapter, we develop a novel causal causal framework for programs that combines program-analysis information [22] with Pearl's Structural Causal Model [53, 54], and Neyman and Rubin's potential outcome model [51, 62] to identify the causes of program failures from data collected from the programs. This approach is purely observational in nature and as such overcomes the limitations of state-altering techniques. The approach also overcomes the limitations of statistical fault-localization and slicing techniques because it attempts to find the program elements or elements that caused the program to fail instead of program elements associated with failure.

Before presenting the framework, we present a motivating example that illustrates the fundamental problem with statistical fault-localization techniques. We illustrate

	t_1 (6,3)	t_2 (5,0)	t_3 (-2,0)	t_4 (0,4)	t_5 (-1,3)	$Ta(s)$	$\hat{\tau}(s)$
void Proc1() {							
1 int x=read();	1	1	1	1	1	0.50	0.40
2 int y=read();	1	1	1	1	1	0.50	0.40
3 if(x > y){	1	1	1	1	1	0.50	0.40
4 print(x);	1	1	0	0	0	0.60	0.17
5 }							
6 if(x != 0){	1	1	1	1	1	0.50	0.40
7 print(y×y); //y+y	1	1	1	0	1	0.60	0.50
8 }							
9 }							
	F	P	P	P	F		

Figure 18: Procedure with test cases, execution data, and causal-effect estimates. The error at statement 7 should be y+y.

the fundamental problem inherent in current statistical fault-localization approaches because as discussed in Section 1, they have been shown through numerous studies to be more effective than non-statistical approaches. We also provide the formal definition of the *failure-causing effect* of a program element.

Definition 31. *The failure-causing effect of a program element, e_i, is the causal effect of e_i on program failure.*

4.1 Motivating Example

We use the procedure (`Proc1`) in Figure 18 to motivate and explain the development of the causal framework. Figure 19 represents the dynamic program dependence graph (Dynamic-PDG) of `Proc1`. `Proc1` reads two integers x and y at lines 1 and 2, respectively, and has an error at line 7. Columns 2 to 6 in Figure 18 show the statement-coverage information gathered by executing the program on five test cases. The top row and bottom row from columns 2 to 6 show the test cases (t_k) and their inputs ((x, y)) to the procedure and the outcomes P (pass) and F (fail) of the procedure, respectively. The value 1 indicates that a statement was covered in a given

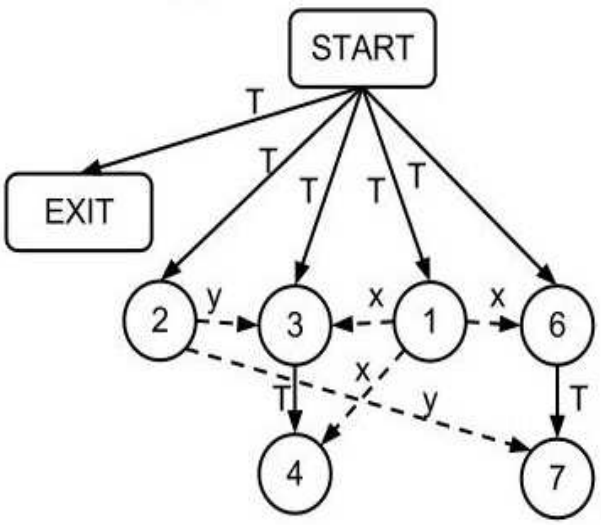


Figure 19: Dynamic-PDG of `Proc1`.

execution and 0 indicates that it was not covered in that execution. For example, column 2 shows that the test case, t_1, covered all the statements and the procedure failed.

Suppose we want to find the statement responsible for the failure, an approach will be to use a heuristic, such as the Tarantula metric by Jones, Harrold, and Stasko [38]. The intuition behind the metric is that statements executed primarily by failing test cases are most likely to be faulty. The Tarantula metric is

$$Ta(s) = \frac{\frac{f_s}{f}}{\frac{f_s}{f} + \frac{p_s}{p}} \tag{30}$$

where f_s and p_s are the number failing and passing test cases covering statement s, respectively. f and p denote the total number of failing and passing test cases, respectively.

Column 7 in Figure 18 shows the suspiciousness score of statements computed with the Tarantula metric. As the Figure shows, statements 4 and 7 have the same suspiciousness score although statement 7 is the faulty statement. Statement 4 has the same suspiciousness as statement 7 because the suspiciousness scores computed

by Tarantula can be divided into two main components: the failure-causing effect of the statement plus a confounding bias.

$$Ta(s) = \text{Failure-Causing Effect}(s) + \text{Confounding Bias}(s) \tag{31}$$

Therefore, to estimate the failure-causing effect of a program element, confounding bias should be mitigated. The confounding bias is due to the interaction between program elements. For example, statements 4 and 7 are control dependent on statements 3 and 6, respectively. We provide detailed analysis of the Tarantula metric and other statistical fault-localization metrics from a causal perspective in Section 5.

4.2 Developing the Framework

The development of our causal framework consists of five main steps. The first three steps identify the primitives of the potential outcome model [63] (i.e., treatments, units, and potential outcomes). The fourth step identifies the confounders of a program element and the fifth step specifies the causal graph that establishes the direction of causal edges between variables in the causal framework. Each step is informed by program-analysis information.

4.2.1 Step 1: Treatment

A treatment in the potential outcome model is an intervention whose effects a researcher would like to ascertain as compared to no intervention. For a program, the goal is to intervene (e.g., manipulate a program element) in such a way as to determine whether a program element is the cause of failure.

A program consists of different types of program elements, such as functions, conditional statements, and assignment statements. The framework accommodates a potentially infinite number of treatments. For example, runtime events (e.g., coverage of program elements) that are used in statistical fault localization are numerous, and all these events can be considered as treatments. However, the treatment should be

Table 11: Observational data gathered for Statement 4.

Feature (Unit)	Treatment (T_4)	Confounder (X_3)	Outcome (Y_4)
u_1	1	1	1
u_2	1	1	0
u_3	0	1	0
u_4	0	1	0
u_5	0	1	1

homogenous. A homogenous treatment is desired because the treatment should be applicable to all program elements so that failure-causing efffects can be compared across different program elements.

An example of a treatment is "coverage of a program element." This treatment satisfies the homogeneity condition. In general, Equation (32) compactly represents the treatment states (T_e) for a program entity e where $T_e = 1$ implies the treatment state and $T_e = 0$ implies the control state.

$$T_e = \begin{cases} 1, & \text{if test covers } (e) \\ 0, & \text{if test does not cover } (e) \end{cases} \tag{32}$$

To estimate the failure-causing effect of a program element, units of the program element (e) should be exposed to the treatment $T_e = 1$ (treatment group) or not exposed to the treatment $T_e = 0$ (control group).

4.2.2 Step 2: Units

According to the potential outcome model, units are entities on which treatments are applied. As discussed in Section 2, a unit can either be exposed to treatment or control but not both. For a program, a set of units is associated with each program element (e) (i.e., $\{u_1, u_2 \cdots, u_n\} \in e$). For our framework, each unit u_k corresponds to a test case t_k. Therefore, the total number of units is the total number of test cases in the test suite.

Table 11 shows the observational data of statement 4 that is derived from the coverage information in Figure 18. The subscripts of the variables represent nodes in the PDG of `Proc1`. The first through fourth columns represent the units of statement 4, the treatment indicator (specifies whether u_k is exposed to treatment state or control state), confounder of statement 4, and observed outcomes of the various units when exposed to the treatment or control state, respectively. As Table 11 shows, statement 4 has five units: two units (u_1, u_2) are in the treatment group because they are covered by two test cases (t_1, t_2) in Figure 18 and three units (u_3, u_4, u_5) in the control group because the units are not covered by the three test cases (t_3, t_4, t_5) in Figure 18.

4.2.3 Step 3: Potential Outcomes

According to the potential outcome model, potential outcomes are the potential responses of units to the treatment. In the framework, each unit (u_k) has a response when it is covered or not covered by a test case (t_k). A unit (u) has two potential-outcome random variables: y^1 if u is exposed to the treatment and y^0 if u is exposed to the control. We use the values y_p and y_f to represent the observed outcomes of the units; $\{y_p, y_f\} \in \mathcal{R}$.

For example, suppose u is in the control group, then its observed outcomes are y_f or y_p if the program failed or passed, respectively. To facilitate the explication of our framework, we use population-level potential outcomes Y^1 and Y^0 for a program element. Note that Y^1 and Y^0 are random variables. Equation (33) provides a compact representation of the potential outcomes.

$$\{Y^1, Y^0\} = \begin{cases} y_p \in \mathcal{R}, & \text{if program passed} \\ y_f \in \mathcal{R}, & \text{if program failed} \end{cases} \tag{33}$$

Designing potential outcomes is challenging because the potential-outcome variables (Y^1, Y^0) should reflect the response of a unit at a given program element and the

response should be represented by a real value. The potential outcome is a function that maps the behavior of a unit (u_k) to a real-valued space ($Y^1(u_k) \mapsto \mathcal{R}$. The challenge comes from having an appropriate real-valued space $\mathcal{R}$ to which to map the potential outcomes. For example, a binary outcome function can be defined by assigning 1 as the response of u_k if the program fails and 0 if the passes.

4.2.4 Step 4: Confounders

As discussed in Chapter 2, to estimate the causal effect without confounding bias, the potential-outcome variables ($\{Y^1, Y^0\}$) should be independent of the treatment variable (T). However, it is not certain whether the dynamic (e.g., coverage) information gathered from the test cases ensures that potential outcomes are independent of the treatment. The uncertainty occurs because the coverage of program elements may have been influenced by other program elements. However, the independence of the potential outcomes from the treatment can be established if Equation (14) holds. Equation (14) expressed in terms of a program element (e) is given by Equation (34).

$$\{Y_e^1, Y_e^0\} \perp\!\!\!\perp T_e \mid X_e \tag{34}$$

According to Equation (34), the causal effect of covering a program element (e) on the outcome of a program can be estimated with reduced confounding bias if factors that affect the treatment and the potential outcomes can be identified in programs. These factors are referred to as confounders and they are represented by X_e. The final step in constructing our framework is, in general, to identify the confounders of a program entity.

The problem of confounding bias arises in programs because of the semantic dependences between program elements induced by syntactic dependences [56]. Finding confounders of program elements can be tedious because, in general, there can be many classes of confounders. However, some classes of confounders can be identified

by relying on information from static or dynamic program analysis. For example, Podgurski and Clarke [56] showed theoretically that a necessary condition for a statement s_1 to semantically influence another statement s_2 is for a chain of control and/or data dependences (syntactic dependences) to exist from s_1 to s_2. However, a chain of control and/or data dependences from s_1 to s_2 is not a sufficient condition for a semantic dependence to exist between s_1 and s_2. The syntactic dependences provide approximate information about how a unit at a given program element is affected directly or transitively by computations at other program elements.

Based on the theoretical results of Podgurski and Clarke [56], our framework identifies classes of confounders of a program element by assuming that causal influences are carried by control and data dependences in a program's program dependence graph (PDG). Using only control and data dependences, our framework identifies two main classes of confounders: confounders induced by control dependences and confounders induced by data dependences and the values carried by the data dependences. For example in the dynamic program dependence graph of procedure `Proc1` shown in Figure 19, statement 7 is control dependent on statement 6 and data dependent on statement 2. The confounders of statement 6 are statement 5 because statement 6 determines whether statement 7 is executed and statement 2 because the computation at statement 6 is determined by the value flowing from statement 2. Statements 5 and 2 belong to the control-dependence class and data-dependence class, respectively.

Also, because each confounder (X_e) influences the treatment and the potential outcomes through the computations that occur at the confounder, the framework represents the computations that occur at a confounder using a set of abstract states, $X_e = \{x_1, x_2, ..., x_n\}$, that are discrete and exhaustive. For example, suppose the abstract state used to characterize the value of a confounder is coverage, then X_e is 1 if the program element is covered and X_e is 0 otherwise.

4.2.5 Step 5: Causal Graphs

As discussed in Section 2.4.1, causal graphs are at the heart of Pearl's Structural Causal Model [53]. Because our framework is based on Pearl's model, causal graphs are an important component of our framework. Causal graphs make explicit the direction of the causal relationships between variables (i.e., the direction of causality between program elements). As mentioned earlier in Section 4.2.4, our framework assumes that the edges in the program dependence graph are causal edges. For each program element, our framework constructs a causal graph for that program element that shows the direction of causal edges between the confounders, program elements, and potential outcomes. The causal graphs enable our technique to identify confounders that introduce spurious associations during causal analysis. These spurious associations can be mitigated by blocking back doors in the causal graph, which is tantamount to conditioning on the identified confounders.

For example, in Figure 19, `Proc1`'s causal influences of program elements upon the occurrence of failure are carried by `Proc1`'s program dependences and Y_e is associated with a program element (e) such as an output statement or the exit point of `Proc1`. Therefore, any back-door paths from T_e to Y_e must begin with an edge $T_e \leftarrow pred(e)$, where $pred(e)$ is a predecessor of e in the program dependence graph of `Proc1`. If considering only control dependences in the program dependence graph, the causal relationships among `Proc1`'s program elements and between them and Y_e can be represented by the forward control-dependence subgraph G_{fcd} of G.

As discussed in Section 2.4.1 of Chapter 2, a causal graph is a directed acyclic graph but the program dependence graph may have loops. Our framework, transforms a subgraph of the program dependence graph consisting of the program element (e), its *immediate predecessors*, and the potential outcome variable into a causal graph (directed acyclic graph). The transformation occurs during the instantiation of a given causal model from the framework. The framework includes only immediate

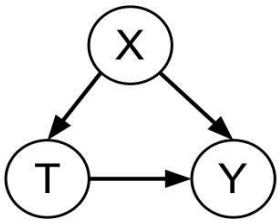


Figure 20: The causal graph of any statement s.

predecessors in the instantiated model based on the "d-separation" property from probabilistic-graphical-model theory discussed in Section 2.4.2. For example, statement 6 is data dependent on statement 1. However, statement 1's effect on statement 7 is blocked by statement 6 if the state of statement 6 is fixed or known. Using the "d-separation" property the framework is able to generate concise causal graphs that facilitate understanding of the causal results. Next, we present an instantiation of the causal framework, and show how that instantiation is used to estimate the failure-causing effect of a program element.

4.3 An Instantiation of the Causal Framework

Many statistical fault-localization techniques use control-flow information about program elements as the data set from which to compute suspiciousness of program elements. Techniques that use control-flow information also referred to as coverage-based fault-localization techniques (CBFL) compute suspiciousness scores in an inexpensive but imprecise manner. They are imprecise because they do not find the failure-causing effect of program elements. In this section, we instantiate a causal model based on control dependences from our causal framework to find the causes of program failures. We present a causal-effect algorithm based on the causal model.

4.3.1 Control-Dependence Causal Model (Causal-CD)

An instantiation of the causal framework that uses control-flow information generates a causal model whose causal graphs are based on control-dependence information (i.e.,

causal edges are control dependences). We call the control-dependence causal model `Causal-CD`. The framework generates a causal model based on control dependences that uses the forward control-dependence predecessor as a confounder in the causal graph. The attributes of the control-dependence model are

1. Treatment: *coverage of a statement*
2. Units: *test cases*
3. Potential outcomes: *1 if the program fails and 0 otherwise*
4. Confounders: *forward control-dependence predecessor*

In choosing the treatment, our framework makes the coverage trigger assumption. The *coverage trigger assumption* states that the coverage of a program element is necessary to trigger a failure if the program element is faulty. The assumption is not sufficient because executing a faulty statement may not cause an invalid program state or the invalid program state may not propagate to the program's output.

A causal graph based on control dependences consists of the treatment indicator for a program element e, the forward control-dependence predecessor of e (X_e), and the potential-outcome variable for e (Y_e). Figure 20 shows a conceptual causal graph of any statement in a program. Nodes X, T, and Y represent the control-dependence predecessor, the treatment variable, and the potential outcome of a statement, respectively. The causal graph in Figure 20 shows that the failure-causing effect of T on Y along the path $T \rightarrow Y$ is confounded by X. The presence of the confounder X causes T to be associated with Y along the path $T \leftarrow X \rightarrow Y$. However, conditioning on X blocks the back-door path and therefore, the failure-causing effect along the path $T \rightarrow Y$ can be estimated accurately. From a program-analysis perspective, conditioning on X is tantamount to removing the semantic dependence between a statement and its forward control-dependence predecessor. Removing the semantic

dependences between statements in a program ensures that the failure-causing effect of a statement can be estimated as if the program consisted of only that statement.

As discussed in Section 4.2.5, a causal graph is a directed acyclic graph but some of the control-dependence causal graphs can have loops. For example, in the following program snippet, statement 2 is control dependent on statement 1, statement 3 is control dependent on statement 2, and statement 4 is control dependent on statement 3.

```
1 if (...)
2   while (...){
3       if (...){
4          break;
5       }
6   }
```

However, there is a control-dependence carried loop from statement 3 to statement 2 because of the *break* statement. Therefore, statement 2 is control dependent on statements 1 and statement 3; statement 2 has two confounders. The framework eliminates control-dependence loops when generating `Causal-CD` by using the forward control-dependence predecessor ($pred_{fcd}(e)$) of a program element (e) in the program dependence graph as a confounder. The framework uses the forward control-dependence predecessor because for a statement to be covered, its $pred_{fcd}(e)$ must first be covered Next, we present how we estimate failure-causing effects of program elements using `Causal-CD`.

4.3.2 Estimating Causal Effects

Suppose we would like to estimate the failure-causing effect of statement 4 in the Dynamic-PDG of `Proc1` in Figure 19 using `Causal-CD`. Figure 21 shows, the causal graph for statement 4. As the graph shows there is a back-door path from statement

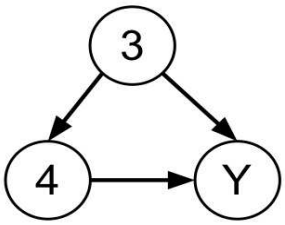


Figure 21: Causal graph of statement 4 in `Proc1`.

4 through statement 3 to the outcome variable (Y) ($4 \leftarrow 3 \rightarrow Y$). According to Pearl, to mitigate the effects of statement 3 on statement 4, the path should be blocked; the path is blocked by conditioning on statement 3 during the causal estimation process.

Suppose the values of the observed outcomes of a unit u_k are represented by the outcome of the procedure: ($y_f = 1$) and ($y_p = 0$) if the program fails and passes, respectively. Also, suppose that the state of a confounder is 1 if it is covered by a test and 0 otherwise. As Table 11 shows, all test cases reach the $pred_{fcd}$ of statement 4, and hence, the confounder (statement 3) has the value of 1 for each test. To demonstrate the estimation of the failure-causing effect of statement 4, we use Equation (17): X_1 denotes the confounder statement 1, $\{Y_2^1, Y_2^0\}$ are the potential outcomes, and T_2 is the treatment applied to statement 2.

$$
\begin{aligned}
\tau_2 &= E[Y_4^1 | X_3, T_4 = 1] - E[Y_4^0 | X_3, T_4 = 0] \\
&= \sum_{X_3} \left[\sum_{Y_4^1} Y_4^1 \Pr(Y_4^1 | X_3, T_4 = 1) - \sum_{Y_4^0} Y_4^0 \Pr(Y_4^0 | X_3, T_4 = 0) \right] \\
&= \Pr(Y_4^1 = 1 | X_3 = 1, T_4 = 1) - \Pr(Y_4^0 = 1 | X_3 = 1, T_4 = 0) \\
&= 0.50 - 0.3333 \\
&\approx 0.17
\end{aligned}
$$

Suppose conditioning on statement 1 is sufficient to remove all confounding bias associated with statement 4 then conditioning on statement 1 is sufficient to compute

the failure-causing effect of statement 4. As the computation shows, the failure-causing effect of statement 4 is now 0.17. The last column of Figure 18 shows the failure-causing effects of the statements in `Proc1`. As the Figure shows, statement 7 is correctly identified as the faulty statement.

In practice, a regression model such as Equation (35) can be used to represent `Causal-CD`. The regression model is intended to include a set X_e of confounders that block all back-door paths in the causal graph between the treatment indicator T_e and the potential-outcome variable Y_e.

$$Y_e = \alpha_e + \tau_e T_e + \beta_e X_e + \varepsilon_e \tag{35}$$

The coefficient τ_e is the failure-causing effect of covering e_i. The failure-causing effect of program elements can then be used as suspiciousness scores. The linear model in Equation (35) is by no means the only choice, or necessarily the best choice, for modeling the causal effect of a program element on failures. A logistic regression model [11] is a natural alternative, and sophisticated nonparametric and semiparametric models have been proposed for causal inference [34]. We chose a linear model as a preliminary measure to demonstrate the relevance of causal inference to fault localization because of its simplicity and the availability of fast, robust software and excellent diagnostics for linear models. The next section presents `Causal-CD`'s fault-localization algorithm.

4.3.3 Causal Fault-Localization Algorithm

Given the binary outcomes from executing a set of tests y_f denoting failure of a test and y_p denoting success and given corresponding program-element coverage profiles, we propose the following basic approach:

1. For each program element e in a faulty program Q, fit a separate linear model M_e having the form of Equation (35), with the following stipulations:

   (a) The outcome variable Y_e is y_f for a test if it fails and is y_p otherwise.

```
   Input: DCDG, D_obs
   Output: Sorted Effects
1  foreach e ∈ DCDG do
2  |  D_e = D_obs(e);
3  |  if pred_fcd(e) then
4  |  |  fit Equation (35) to D_e;
5  |  else
6  |  |  fit Y_e = α_e + τ_e T_e + ε_e to D_e;
7  |  end
8  |  effects(e) = τ_e;
9  end
10 Sort effects in descending order;
```

Figure 22: The LocalizeFault-DCDG algorithm

(b) The "treatment" indicator T_e is 1 for a test if it covers e_i and is 0 otherwise.

(c) If e has a forward control-dependence predecessor $pred_{fcd}(e)$, then M_e has a single binary covariate X_e, which is 1 for a test if it covers $pred_{fcd}(e)$ and is 0 otherwise; if e has no forward control-dependence predecessor then M_e has no covariates (i.e., $Y_e = \alpha_e + \tau_e T_e + \varepsilon_e$)

2. For each program element e, use the least-squares estimate $\hat{\tau}_{ls,e}$ of the coefficient τ_e of T_e in M_e like a suspiciousness value. That is, rank statements (for inspection by developers) in nonincreasing order of $\hat{\tau}_{ls,e}$.

Figure 22 shows the fault-localization algorithm, LocalizeFault-DCDG, based on the control-dependence causal model. LocalizeFault-DCDG takes as input the dynamic control dependence graph (DCDG) and the observational data (D_{obs}) for each of the program elements. Suppose each program element (e_i) corresponds to a node in the control dependence graph. For each e, LocalizeFault-DCDG gets the observational data for e (D_e) from D_{obs} at line 2. If e has a $pred_{fcd}(e)$, LocalizeFault-DCDG fits Equation (35) to the observational data (Line 4) otherwise LocalizeFault-DCDG

fits a regression model to the observational data of e without $pred_{fcd}(e)$ (Line 6). LocalizeFault-DCDG stores the failure-causing effects of the program elements (Line 8) in $effects$. After all the program elements have been processed, the failure-causing effects are sorted in descending order (Line 10) and presented to the developer. Note that this is by no means an efficient algorithm as the algorithm computes the failure-causing effect for each program element. A specific optimization that can be performed is to compute the failure-causing effect of a given control-dependence region [22] and impute the effect to all program elements in that region. This optimization is possible because all the program elements in the same control-dependence region have the same forward control-dependence predecessor (confounder) and coverage profiles.

4.3.4 Worst-case space and execution time for `LocalizeFault-DCDG`

In this section, we present the space and execution-time complexity of the `LocalizeFault-DCDG` algorithm. Suppose |DCDG| is the number of nodes in the dynamic control dependence graph, $|R|$ is the cost of fitting the regression models to the observational data, and S is the time to sort causal effects. The worst-case execution time for the algorithm is $O(|\text{DCDG}| \times |R| + |S|)$ because for each program element (e) a call is made to fit the regression model and also the causal effects are sorted at the end. The worst-case space required to store D_{obs} is $O(3 \times |T| \times |\text{DCDG}|)$ because we need to store the treatment variable, confounding variable, and the outcome variable for each of the units of a program element. The total number of units of a program element is T. The space complexity for the DCDG is $O(|\text{DCDG}| + |E|)$, where E is the number of edges in the dynamic control dependence graph.

CHAPTER 5

UNIFYING THE FAULT-LOCALIZATION METRICS

This chapter presents the analysis and unification of the most well-known fault-localization metrics (suspiciousness metrics), from a causal perspective. We analyzed six metrics: the Tarantula metric [37, 38], the Ochiai metric [1], the Jaccard metric [1], a metric called the F_1-measure, Liblit and colleagues' Importance (p) metric [42], and the Wong, Debroy, and Choi metrics [70]. The analyses show that each metric "embeds" in some way an estimator for the probability that a program Q fails given that a program entity e is covered, (i.e., Pr(Q fails| e covered), also denoted as Pr($F \mid e$) for brevity). Because the techniques use specific program elements, we analyze each metric based on the program elements used. We represent the output of the metrics by $Score_t$, which measures the suspiciousness of a program entity e; the subscript t represents the technique. Most of the techniques (with the exception of Liblit and colleagues' metric) use a statement s as the program element. We analyze each metric to determine the metric's properties and relationship to the other metrics and later unify from the perspective of our causal model.

The following notation characterizes a test suite used for fault localization: n is the total number of test cases; n_e is the number of test cases covering a particular program element e; p is the total number of test cases that pass (succeed); $p_{\overline{e}}$ is the number of test cases that pass and that do not cover e; f is the total number of test cases that fail; p_e is the number of test cases that pass and that also cover e; f_e is the number of test cases that fail and that also cover e; $f_{\overline{e}}$ is the number of test cases that fail and that do not cover e. In the probability expressions below, P denotes the event that a test case passes and F denotes the event that a test case fails. e denotes

the event that a test case covers the program entity.

5.1 Tarantula Metric

The Tarantula suspiciousness metric [37, 38] is

$$Score_{\text{Ta}}(e) = \frac{\frac{f_e}{f}}{\frac{p_e}{p} + \frac{f_e}{f}} \tag{36}$$

By multiplying the numerator and denominator by f to simplify, we get

$$= \frac{f_e}{\left(\frac{p_e}{p} \times f\right) + f_e} \tag{37}$$

Suppose the number of test cases that pass and the number of test cases that fail are equal (i.e., $p = f$) in the test suite used for fault localization (e.g., because it was deliberately made equal). Then, the Tarantula metric has a simpler form

$$\begin{aligned} Score_{Ta}(e) &\approx \frac{f_e}{p_e + f_e} \\ &= \frac{f_e}{n_e} \end{aligned} \tag{38}$$

$\frac{f_e}{n_e}$ is the sample estimator for the population estimator $\Pr(F|e)$.

$$Score_{Ta} \approx \Pr(F|e) \tag{39}$$

Thus, if the number of test-case successes and failures are equal, the Tarantula metric is simply an estimator for the probability of failure given that a program entity e is covered.

5.2 Ochiai Metric

The Ochiai metric [1] is

$$Score_O(e) = \frac{f_e}{\sqrt{f(f_e + p_e)}} \tag{40}$$

Squaring and simplifying Equation (40), we get

$$\begin{aligned} &= \frac{(f_e)^2}{f \cdot n_e} \\ &= \frac{f_e}{n_e} \times \frac{f_e}{f} \end{aligned} \tag{41}$$

$$(Score_O)^2(e) = \left(\frac{f_e}{f}\right) \Pr(F|e) \tag{42}$$

As the derivations show, the Ochiai metric is a weighted version of the $\Pr(F \mid s)$ with the weight being $\left(\frac{f_e}{f}\right)$. Program entities are assigned more weight if they are covered by more failing test cases.

5.3 Jaccard Metric

The Jaccard metric [1, 14] is

$$Score_J(e) = \frac{f_e}{f_e + p_e + f_{\bar{e}}} \tag{43}$$

Simplifying and dividing the numerator and denominator by n_s gives

$$\begin{aligned} &= \frac{f_e}{f + p_e} \\ &= \frac{\frac{f_e}{n_e}}{\frac{n_e}{n_e} + \frac{f}{n_e}} \\ &= \left(\frac{n_e}{f + p_e}\right) \Pr(F \mid e) \\ Score_J(e) &= \left(\frac{f_e + p_e}{f + p_e}\right) \Pr(F \mid e) \end{aligned} \tag{44}$$

The Jaccard metric is essentially a weighted version of $\Pr(F \mid e)$ with the weight being $\left(\frac{f_e + p_e}{f + p_e}\right)$. The key observation here is that f is a constant across all statements and therefore the suspiciousness of e increases if less passing test cases do not cover e and more failing tests cover e.

5.4 Relationship among Tarantula, Jaccard, and Ochiai

There are connections among Tarantula, Jaccard, and Ochiai. To show the connections, we analyze the coefficients of $\Pr(F|s)$ in the metrics. Beginning with the Jaccard metric, suppose $p_s \to 0$,

$$lim_{p_s \to 0} \left(\frac{f_s + p_s}{f + p_s} \right) = \frac{f_s}{f}$$
$$Score_J(s) \approx \left(\frac{f_s}{f} \right) \Pr(F|s) \tag{45}$$

Equation (45) implies that if the number of passing test cases is significantly less than the number of failing test cases that cover a given statement s, the Jaccard metric behaves like the Ochiai metric. Suppose that $p_s \to +\infty$,

$$lim_{p_s \to \infty} \left(\frac{f_s + p_s}{f + p_s} \right) = 1$$
$$Score_J(s) \approx \Pr(F|s) \tag{46}$$

Equation (46) implies that if the number of passing test cases that cover a statement s are far more than the number failing test cases that cover s the Jaccard metric behaves like the Tarantula metric. Also analyzing the Ochiai metric, if $f_s \to f$, the behavior of the Ochiai metric approximates the Tarantula metric (Equation (47)).

$$lim_{f_s \to f} \left(\frac{f_s}{f} \right) = 1$$
$$Score_O(s) \approx \sqrt{\Pr(F|s)} \tag{47}$$

5.5 Cooperative Bug Isolation (CBI) Metric

We analyze Liblit and colleagues' [42] CBI metric by using concepts and terminology from the field of information retrievel [45]. Suppose we view a test covering e as being analogous to a document retrieved with a given query and a test that fails as being analogous to a *relevant* document then $\Pr(F \mid e)$ represents precision and $\Pr(e \mid F)$ represents recall. A standard way of balancing recall and precision when evaluating

the effectiveness of a query is to use the *F-measure* [45], which is the (unweighted) harmonic mean of recall and precision. In the present context, it can be written as

$$F(e) = \frac{2}{\frac{1}{\Pr(e|F)} + \frac{1}{\Pr(F|e)}} \tag{48}$$

The fault-localization metric named *Importance* that was defined by Liblit and colleagues [42] is based on the *F-measure*. Liblit and colleagues applied their metric to program predicates rather than arbitrary statements, and so their metric is not fully comparable to the suspiciousness metrics described above. Nevertheless, their technique embeds the measure $\Pr(F \mid e)$ in the Importance metric. For a given predicate q, $Importance(q)$ metric is

$$Importance(q) = \frac{2}{\frac{1}{Increase(q)} + \frac{log(f_q)}{log(f)}} \tag{49}$$

where $Increase(q)$ measures precision as

$$Increase(q) \approx \Pr(F|q \text{ true}) - \Pr(F|q \text{ evaluated}) \tag{50}$$

The first term of this difference is identical to $\Pr(F|e)$ if q is true if and only if e is covered. The second term, denoted $Context(q)$, is a sort of 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 [42]. *Increase(q)* measures the strength of the association between a predicate being true and the failure of a program. *Increase(q)* shows that CBI metric embeds the estimator, $\Pr(F|e)$, in its importance metric.

5.6 Wong, Debroy, and Choi's Metric

Wong, Debroy, and Choi [70] present three techniques each with its own fault-localization metric. We analyze the third technique because it addresses the limitations of the two other techniques. The analysis shows that the third technique's

metric embeds the estimator, $\Pr(F \mid e)$. The intuition behind the third technique is that failing and passing test cases contribute differently to the suspiciousness value of a faulty program entity. For a given program entity e, the technique constructs bins containing different numbers of failing test cases and passing cases. The number of failing test cases in the bins is f_e and the number of passing test cases in the passing bins is p_e. Different weights are then assigned to each bin with the constraint that the sum of the weights of the failing test cases that cover e should be greater than the weights of the passing test cases that cover e.

For clarity, we employ the following notation in analyzing the metric: w_i is the weight of each failing bin i; v_j is the weight of each passing bin j; $|B_f|$ is the total number of failing bins; $|B_p|$ is the total number of passing bins; $|B_i^f|$ is the number of test cases in failing bin i; and $|B_i^p|$ is the number of passing test cases in passing bin j. Given the notation, Wong, Debroy, and Choi metric is,

$$score_{WDC}(e) = \sum_{i=1}^{|B_f|} \left(w_i \times |B_i^f| \right) - \sum_{j=1}^{|B_p|} \left(v_j \times |B_j^p| \right) \tag{51}$$

The constraint is

$$\sum_{i=1}^{f_s} w_i > \sum_{j=1}^{p_s} v_j \tag{52}$$

The two components (failing and passing) of Equation (51), $\sum_{i=1}^{|B_f|} \left(w_i \times |B_i^f| \right)$ and $\sum_{j=1}^{|B_p|} v_j \times |B_j^p|$, are linear equations that, given the weights w_i and v_j, evaluate to a constant, repectively. Simplifying the failing component results in

$$\begin{aligned} \sum_{i=1}^{|B_f|} \left(w_i \times B_i^f \right) &= w_1|B_1^f| + w_2|B_2^f| + \cdots + w_n|B_n^f| \\ &= \alpha f_s \end{aligned} \tag{53}$$

where α in Equation (53) is a variable that depends on the weights w_i.

Simplifying the passing component gives

$$\begin{aligned}\sum_{j=1}^{|B_p|}\left(v_j \times B_j^p\right) &= v_1|B_1^p| + v_2|B_2^p| + \cdots + v_n|B_n^p| \\ &= \beta p_s \end{aligned} \tag{54}$$

where β is a variable that depends on the weights v_j. Equation (51) can then be represented as

$$\begin{aligned} Score_{WDC}(e) &= \alpha f_e - \beta p_e \\ &= \alpha f_s - \beta(n_e - f_e) \\ &= (\alpha + \beta) f_e - \beta n_e \\ &= n_e\{(\alpha + \beta)\frac{f_e}{n_e} - \beta\} \\ &= n_e\{(\alpha + \beta)\Pr(F \mid e) - \beta\} \end{aligned} \tag{55}$$

Equation (55) shows that Wong, Debroy, and Choi's metric also embeds the sample estimator of $\Pr(F \mid e)$.

5.7 Causal Analysis of the Metrics

Each of the proposed suspiciousness metrics discussed in the previous sections embeds a sample estimator for $\Pr(F|e)$. $\Pr(F \mid e)$ measures the strength of the association between a program entity e and the failure of a program. The metric $\Pr(F \mid e)$ is the common thread that unifies all the other metrics. The weights associated with $\Pr(F \mid e)$ is an attempt to account for the test suite composition. The analysis in this section shows that $\Pr(F \mid e)$ does not account for confounders and therefore, cannot be used to estimate the failure-causing effect of a program element.

We use the same notations as in Chapter 4: Y_e^1 and Y_e^0 represent the potential outcomes for program states at program element e in the treatment group and the control group, respectively. T_e is the treatment indicator (coverage of a program

element). Suppose T_e is defined as

$$T_e = \begin{cases} 1, & \text{if a program entity } e \text{ is covered by a test case} \\ 0, & \text{if a program entity } e \text{ is not covered by a test case} \end{cases} \tag{56}$$

and the potential outcomes under coverage of e are defined as

$$\{Y_e^1, Y_e^0\} = \begin{cases} 1, & \text{if a test case/program fails} \\ 0, & \text{if a test case/program passes} \end{cases} \tag{57}$$

Suppose we do not account for the covariates (X_e) of the program entity e, then the biased causal effect is

$$\begin{aligned} \tau_{biased} &= E[Y_e^1|T_e = 1] - E[Y_e^0|T_e = 0] && (58) \\ &= \sum_{Y_e^1 \in \{0,1\}} Y_e^1 \Pr(Y_e^1 \mid T_e = 1) - \sum_{Y_e^0 \in \{0,1\}} Y_e^0 \Pr(Y_e^0 \mid T_e = 0) \\ &= \Pr(Y_e^1 = 1 \mid T_e = 1) - \Pr(Y_e^0 = 1 \mid T_e = 0) \\ &= \Pr(F \mid e) - \Pr(F \mid \overline{e}) && (59) \end{aligned}$$

Suppose $\Pr(F \mid \overline{e}) = 0$, that is the probability of failure given that e is not covered is not accounted for, then Equation (59) reduces to Equation (60).

$$\tau_{biased} = \Pr(F \mid e) = E[Y_e^1|T_e = 1] \tag{60}$$

From Equation (60), $\Pr(F \mid e)$ is clearly related to the biased causal estimator in Equation (11) in Section 2.3.2. To determine the bias inherent in Equation (60), we apply Equation (13),

$$\begin{aligned} E[Y^1 \mid T_e = 1] - E[Y_e^0 \mid T_e = 0] &= \tau + \{E[Y_e^0|T_e = 1] - E[Y_e^0|T_e = 0]\} \\ &+ (1 - \pi)\{E[Y_e^1 - Y^0|T_e = 1] - E[Y_e^1 - Y^0|T_e = 0]\} && (61) \\ \Pr(F|e) = E[Y_e^1|T_e = 1] &= \tau + \{E[Y_e^0|T_e = 1] - E[Y_e^0|T_e = 0]\} \\ &+ (1 - \pi)\{E[Y_e^1 - Y_e^0|T_e = 1] - E[Y_e^1 - Y_e^0|T_e = 0]\} \\ &+ E[Y_e^0|T_e = 0] && (62) \end{aligned}$$

Equation (62) shows that the current metrics have three main sources of bias instead of two: (1) the differential treatment-effect bias, (2) the selection or baseline bias, and (3) $E[Y_e^0 \mid T_e = 0]$, which is the average potential outcome of program entities not covered by test cases. Using $\Pr(F \mid e)$ as a causal estimator also implies that $E[Y_e^0 \mid T_e = 0] = 0$, which is not valid. What Equation (62) shows is that the analyzed metrics do not estimate the causal effect of a program entity on the failure of a program because units are only exposed to treatment (the program entity is covered).

The significance of the presence of $\Pr(F \mid e)$ in the metrics will become clear in results of the empirical studies in Section 5.8. The empirical studies also show that some of the weights attached to $\Pr(F \mid s)$ are useful.

5.8 Empirical Studies

To evaluate the effectiveness of our causal framework for finding causes of program failures at the statement level (i.e., a program element is a statement), we implemented the framework, instantiated the control-dependence causal model (`Causal-CD`) with different potential-outcome functions, and conducted several empirical studies involving several subject programs. The studies investigated four main research questions:

RQ1: How effective is `Causal-CD` compared to the associative fault-localization metrics?

RQ2: How effective is integrating the causal estimator of `Causal-CD` into the associative fault-localization metrics?

RQ3: How do different potential-outcome functions affect the accuracy of fault localization?

RQ4: How efficient is our causal approach?

Table 12: Subjects used for empirical studies.

Program	Num. of Vers. Used / Num. of Vers.	Num. of Test Cases	Num. of DCDG Nodes (Min / Max)	Num. of Executed Lines (Min / Max)	Description
Cal	19 / 20	162	117 / 139	80 / 96	calendar printer
Col	29 / 30	156	147 / 252	95 / 177	filter-line reverser
Comm	10 / 12	186	28 / 133	22 / 88	file comparer
Look	9 / 14	193	131 / 143	69 / 78	word finder
Spline	13 / 13	700	243 / 250	139 / 145	curve interpolator
Tr	11 / 11	870	145 / 151	79 / 81	character translator
Uniq	17 / 17	431	111 / 136	65 / 81	duplicate line remover
Print-tokens	5 / 7	4130	422 / 428	207 / 210	lexical analyzer
Print-tokens2	10 / 10	4115	336 / 343	196 / 204	lexical analyzer
Replace	28 / 32	395	412 / 420	271 / 277	pattern replacement
Schedule	9 / 9	2710	215 / 220	153 / 156	priority scheduler
Schedule2	9 / 10	2650	222 / 230	138 / 141	priority scheduler
Tcas	41 / 41	1608	128 / 141	55 / 58	altitude separation
Tot-info	23 / 23	1052	247 / 254	122 / 124	information measure
Sed	10 / 10	363	2700 / 4365	1619 / 2218	stream editing utility
Grep	15 / 15	470	696 / 3579	775 / 1895	text-search utility
Gzip	18 / 18	211	2301 / 2751	1238 / 1482	compression utility
Space	30 / 38	157	1330 / 4183	1092 / 3583	ADL interpreter

Next, we present the empirical setup, implementation, how we measure the effectiveness of our causal approach, and the results of the empirical studies.

5.8.1 Empirical Setup

We used the Unix suite (Cal, Col, Comm, Look, Spline, Tr, Uniq),[1] the Siemens suite, Sed, Gzip, Grep, and Space[2] as subject programs in our studies. Table 12 shows their characteristics. For each subject, the first to sixth columns shows, the program name, the ratio of versions used to the total number of versions, the number of test cases, the minimum and maximum number of nodes in the dynamic control dependence graph, the minimum and maximum number of executed lines of code, and a brief description.

The Space subject comes with 38 faulty versions and different coverage-based test suites. We randomly chose a test suite that achieves branch-coverage and executed it on Space. We used 30 faulty versions of Space.

The Sed, Grep, and Gzip are large Unix subjects that come with multiple versions with multiple faults per version. Each fault in a version can be activated separately. For Sed, we randomly chose three different versions and activated 10 faults. For Grep and Gzip, we activated 15 and 18 faults, respectively.

We omitted some of the faulty versions from the studies either because (1) there were no syntactic differences between the C file of the correct version and the faulty versions of the program (e.g., because the fault was in the header file) or (2) none of the test cases failed when executed on the faulty version of the program or (3) the test suite we randomly chose had only passing or failing test cases when executed on some versions or (4) the size of the generated coverage data was too large (gigabytes) to process in a reasonable amount of time. In total, we performed our empirical studies

[1] We obtained the Unix suite from Eric Wong of University of Texas at Dallas

[2] We obtained the Space, Gzip, Grep, and Sed programs from the Software-artifact Infrastructure Repository [19].

on 305 faulty versions.

5.8.2 Implementation

To construct the control-dependence causal model `Causal-CD`, our technique extracts the control-dependence graphs and instruments the programs using the CIL framework [50], which supports the analysis of ANSI C programs. We implemented the control-dependence algorithms in the *Objective Caml language*, because it is required for interfacing with the CIL framework.

Our technique instruments each faulty version to enable construction of the dynamic control-flow graph for each function. The technique then computes the dynamic control-dependence graph for each function from its dynamic control-flow graph. Our technique computes dynamic control dependence graphs instead of static control-dependence graphs. Using the dynamic control-dependence graphs ensures that only control dependences that are exercised at runtime appear in the control dependence graph. We implemented the LocalizeFault-DCDG algorithm and the fault-localization metrics using *R* [58], which is a system for statistical computation that consists of a language and a run-time environment.

Our technique computes a node-coverage matrix for each faulty version. The node-coverage matrix shows the coverage of the nodes in the dynamic control dependence graph. Note that there is a many-to-one correspondence from nodes to statements. For example, a compound condition consisting of conjunctions or disjunctions of predicates. A statement is identified by the line number where it occurs in the program and each node has a line number as part of its identifier. For detailed explanations of CIL transformations, see Reference [50]. Our technique computes the fault matrices for each faulty version that indicates for each faulty version, which test cases pass and fail.

We implemented two variants of `Causal-CD` that use two different potential outcomes PO-1 and PO-2. For brevity, we refer to the two variants as CD-1 (uses PO-1) and CD-2 (uses PO-2), respectively. Note that PO-1 is the default potential-outcomes function used by the LocalizeFault-DCDG.

5.8.3 Measuring Effectiveness

To measure the effectiveness of the various metrics with respect to fault localization we perform two steps. During the first step, we map nodes in the dynamic control dependence graph onto the line numbers associated with the node. We assume that each line number corresponds to a single statement. We map nodes to statements because developers often examine program statements during fault localization. Second, we use the cost-measuring metric (*Cost*) in Equation (63) used by References [6, 16, 38, 60].

$$Cost = \frac{|S_e|}{|S_T|} \times 100\% \tag{63}$$

S_e represents the number of statements examined by the developer until the faulty statement is found. We assume that the statements are presented in nondecreasing order of suspiciousness and the developer starts by examining the most suspicious statement. S_T represents the total number of executed statements.

To compare two metrics A and B for effectiveness, we first use one of the metrics (say B) as the reference metric. We then subtract the *Cost* value for A from the *Cost* value for B. A positive value means that A performed better than B and a negative value means B performed better than A. The difference corresponds to the magnitude of improvement. For example, for a given version, if the *Cost* of A is 30% and the *Cost* of B is 40%, then the improvement of A over B is 10%, which means that developers would examine 10% fewer statements if they used A.

Tables 13, 16 show a summary of the results of comparing CD-1 and CD-2 to the associative fault-localization techniques and also the results of integrating the causal

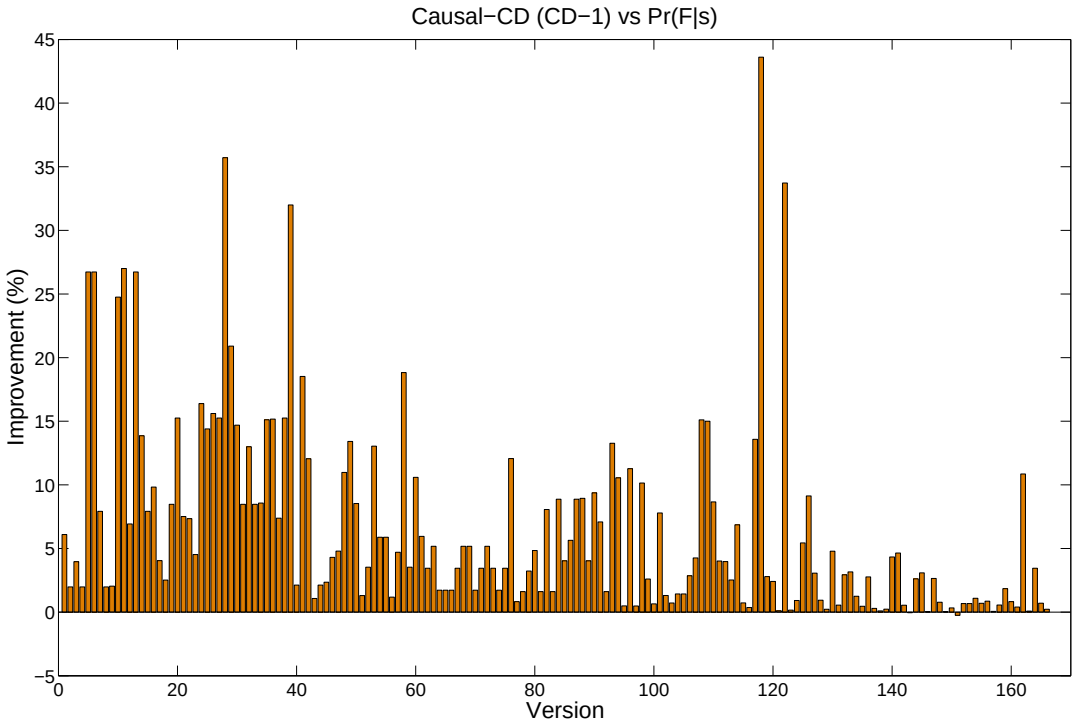
effects (i.e., $\hat{\tau}_{ls,s}$) of CD-1 and CD-2 into some of the associative techniques. We also refer to Causal-Ochiai, Causal-Jaccard and Causal-F1 as CO-x, CJ-x and CF1-x, respectively; So "x" is 1 or 2 depending on whether the $\hat{\tau}_{ls,s}$ of CD-1 or CD-2 is used, respectively. The first column shows comparison of the fault-localization techniques; the comparison of A and B implies B is used as the reference metric. The second, third, and fourth columns show the percentage of faulty versions that A performed better, worse, and the same as B, respectively.

Tables 14 and 17, show a summary of the various statistics that capture in detail the percentage of improvements of the techniques. The first column shows comparison of the fault-localization techniques; comparison of A and B implies B is used as the reference metric. The second to fifth columns show minimum, median, maximum, and mean improvements of A over B, respectively.

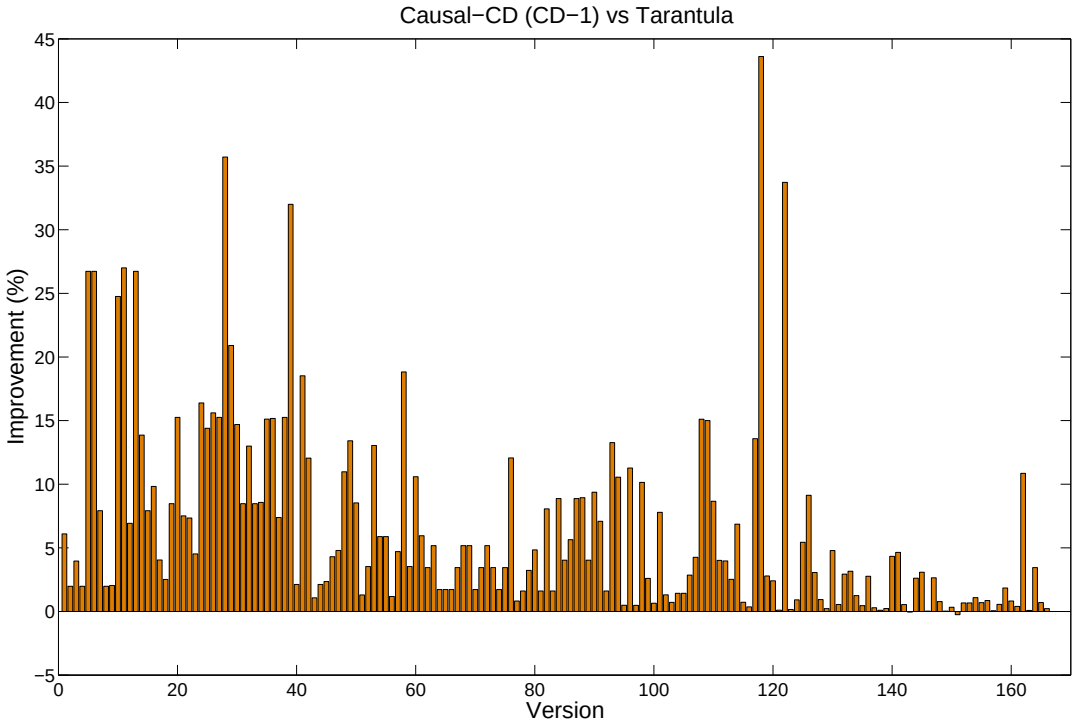
There is a complication that should be mentioned here when measuring the effectiveness of fault-localization techniques when the failure is due to missing code. In fault-localization research, a fault involving missing code is by convention associated with an existing, often correct, program element near where it is believed the missing code should be located. Because missing code can often reasonably be placed in different locations, it may not be the case that such a program element is even moderately associated with failures.

5.8.4 RQ1

The goal of RQ1 is to determine the fault-localization effectiveness of `Causal-CD` with respect to the existing associative fault-localization metrics: the estimator f_s/n_s of $\Pr(F|s)$, Tarantula, Ochiai, Jaccard, and F1-measure. To answer this research question, we measure the Costs of the control-dependence causal model (CD-1) and compare it to the Costs of $\Pr(F|s)$ and Tarantula. We used $\Pr(F|s)$ and Tarantula as the reference metrics and subtracted the Costs of CD-1 from the Costs of $\Pr(F|s)$



(a) Comparison of Causal-Effect Estimator to $\Pr(F|s)$.



(b) Comparison of Causal-Effect Estimator to Tarantula.

Figure 23: Comparison of Causal-Effect Estimator to $\Pr(F|s)$ and Tarantula.

and Tarantula.

Figures 23(a) and 23(b) show the results of comparing CD-1 to $\Pr(F|s)$ and Tarantula, respectively. The horizontal axes represent the number of versions that show differences in the Cost of fault localization. The vertical axes represent the percentage difference in Costs, which is the magnitude of improvement. The zero-level lines represent the performance of the reference metric. Bars above the zero-level lines represent versions for which our technique performed better than the reference metric, and bars below the zero-level line represent versions for which our technique performed worse. As the graph shows, CD-1 shows improvements over both $\Pr(F|s)$ and Tarantula. The graphs also show that the performances of $\Pr(F|s)$ and Tarantula are the same, which further confirms the accuracy of our analysis in Chapter 5.

The first four rows of Table 13 shows the summary of the results of comparing CD-1 to the other associative techniques. As Table 13 shows, CD-1 performs better than $\Pr(F|s)$ and Tarantula on 53.59% of the faulty versions, worse on 0.65%, and performed the same on 45.75%. The results show that conditioning on the control-dependence predecessor reduces the confounding bias. The first four rows of Table 14 shows the various improvements of the faulty versions for which technique A performed better than B. For example, for the faulty versions for which CD-1 performed better than Tarantula, half of the faulty versions had improvements between 0.03 and 3.99 and the other half had improvements between 3.99 and 43.61; the mean performance was 6.69.

Table 13 also shows that CD-1 performs worse when compared to Ochiai, Jaccard, and the F1-measure. We have identified three main reasons why CD-1 performed worse than these associative techniques. The first reason relates to the coefficients associated with the precision measure ($\Pr(F|s)$ associated with Ochiai, Jaccard, and the F1-measure. The results indicate that an associative measure that combines both a recall and precision estimator ($\Pr(F|s)$) is more effective than only a precision-based

Table 13: Comparison of fault-localization models

Fault Loc. Tech.	Better (%)	Worse (%)	Neutral (%)
CD-1 vs Pr(F | s)	53.59	0.65	45.75
CD-1 vs Tarantula	53.59	0.65	45.75
CD-1 vs Ochiai	9.48	44.12	46.41
CD-1 vs Jaccard	17.97	31.37	50.65
CD-1 vs F_1-measure	17.97	31.37	50.65
CO-1 vs Ochiai	23.20	0.33	76.47
CJ-1 vs Jaccard	29.08	0.33	70.59
CF1-1 vs F_1-measure	39.21	1.31	59.48

Table 14: Distribution of "Better" improvements

Fault Loc. Tech.	Minimum (%)	Median (%)	Maximum (%)	Mean (%)
CD-1 vs Pr(F | s)	0.03	3.99	43.61	6.69
CD-1 vs Tarantula	0.03	3.99	43.61	6.69
CD-1 vs Ochiai	0.08	2.35	13.15	3.27
CD-1 vs Jaccard	0.06	2.41	15.11	4.01
CD-1 vs F_1-measure	0.06	2.41	15.11	4.01
CO-1 vs Ochiai	0.05	1.72	12.06	2.91
CJ-1 vs Jaccard	0.05	2.17	15.11	3.56
CF1-1 vs F_1-measure	0.05	4.39	62.10	7.37

estimator. To explain why the recall is important for fault localization, consider a statement s for which $\Pr(s|F)$ 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 $\Pr(F|s)$ or the causal effect of s on failures is likely to be quite small. If there are many passing test cases that cover s, the overall sample of test cases covering s will be imbalanced (i.e., more passing test cases than failing test cases). If there are few passing test cases that cover s, the overall sample of test cases covering s will be small, even if it is not imbalanced. In both cases, an estimate of either $\Pr(F|s)$ or the causal effect of s is likely to be untrustworthy. The way estimates of recall are used in the Ochiai and F_1 metrics partially addresses this issue by reducing the suspiciousness values of statements for which the recall $\Pr(s|F)$ is low and by increasing the suspiciousness values of statements for which it is high.

Such weighting is likely to be much more practical than the alternative of attempting to roughly balance failures and successes covering each statement. We investigate further the importance of the recall in RQ2.

The second reason for the causal estimator performing worse relates to the effects of different potential-outcome functions of units of program elements in the treatment and control groups. We present a detailed study that explains the second reason in RQ3. The third reason concerns the fundamental issues, overlap, balance, and confounders, which are central to our causal framework. We provide the details of the third reason in Section 5.9.

5.8.5 RQ2

The goal of RQ2 is to determine the effectiveness of integrating the causal estimator ($\hat{\tau}_{ls,s}$) of CD-1 into the Ochiai metric, the Jaccard metric, and the F_1-measure. The results demonstrate the importance of the recall measure as explained in Section 5.8.4. Table 15 shows the associative techniques (Ochiai, Jaccard, and F_1-measure) and their causal counterparts (Causal-Ochiai, Causal-Jaccard, and Causal-F_1). We replace the precision estimator $\Pr(F|s)$ in the Ochiai, Jaccard, and F_1-measure with $\hat{\tau}_{ls,s}$. For the F_1-measure, we use the *inverse-logit*[3] of the causal estimator (τ_s). We use the inverse-logit function to convert the causal-effect estimate into a probability value because the F_1-measure has $precision + recall$ in its denominator. The *recall* and *precision* values are both probabilities, and thus, the causal-effect estimate must be a probability (i.e., its value must lie between 0.0 and 1.0 inclusive).

Table 13 shows that CO-1, CJ-1, and CF1-1 performed better than their non-causal counterparts. For example, in Table 13 CO-1 performs better on 23.20% of faulty versions and worse on 0.33% of faulty versions. Table 14 shows that, for CO-1, half of the faulty versions had improvements between 0.05% and 1.72% and the other

[3]The inverse-logit function is $invlogit(x) = exp(x)/(1 + exp(x))$.

Table 15: Associative techniques and their causal counterparts

	Associative Technique	Causal Technique
Jaccard	$\left(\frac{f_s + p_s}{f + p_s}\right) \Pr(F \mid s)$	$\left(\frac{f_s + p_s}{f + p_s}\right) \times (\hat{\tau}_{ls,s})$
Ochiai	$\left(\frac{f_s}{f}\right) \times \Pr(F \mid s)$	$\left(\frac{f_s}{f}\right) \times (\hat{\tau}_{ls,s})$
F1-measure	$\frac{2}{\frac{1}{\Pr(s \mid F)} + \frac{1}{\Pr(F \mid s)}}$	$\frac{2}{\frac{1}{\Pr(s \mid F)} + \frac{1}{invlogit(\hat{\tau}_{ls,s})}}$

half from 1.72% to 12.06%.

The results show that the improvement in fault-localization effectiveness of CO-1, CJ-1, and CF1-1 is from conditioning on the forward control-dependence predecessor of a statement. Thus, conditioning on the forward control-dependence predecessor reduces confounding bias when finding the causes of program failures. Second, the results show that associative techniques for which it is possible to incorporate the causal estimator of CD-1 can be significantly improved. Third, the recall measure is useful in providing some sort of balance to the ratio of passing and failing test cases in the test suite.

5.8.6 RQ3

The goal of RQ3 is to investigate how different potential-outcome functions affect the accuracy of fault localization. To answer this question, we constructed a new potential-outcome function: PO-2 and compared it to the default potential-outcome function used by LocalizeFault-DCDG. The default potential-outcome function (Equation (64)) assigns the value of 1 to units in the treatment group and control group that are covered by failing test cases and 0 to units covered by passing test cases. The underlying intuition is that all the units features of different statements

have the same response when exposed to treatment or control.

$$\{Y_s^1, Y_s^0\} = \begin{cases} y_f = 1, & \text{if program failed} \\ y_p = 0, & \text{if program passed} \end{cases} \tag{64}$$

The results of using PO-1 in the control-dependence causal model are shown by the results of CD-1 in Tables 13 and 14, whose results we have explained in RQ1.

The second potential-outcome function (PO-2) in (Equation (65)) is motivated by the fact that units of different statements respond differently when exposed to treatment or control. For example, suppose a fault occurs in a program and it is a segmentation fault, computations that do not involve pointers should not have the same potential outcomes as computations involving pointers. PO-2 models in a limited way the probability of faultiness of units of statements. However, because it is difficult, if not impossible, to determine the probability of faultiness of a unit, we approximate using Equation (65).

$$\{Y_s^1, Y_s^0\} = \begin{cases} y_f = f_s/f, & \text{if unit is treated} \\ y_f = f_{\bar{s}}/f, & \text{if unit is untreated} \\ y_p = 0, & \text{otherwise} \end{cases} \tag{65}$$

This function assigns a weight to the units of a statement depending on the ratio of failing test cases that cover units in the treatment group and units in the control group. The outcome $y_f = f_s/f$ is imputed to all units in the treatment group for which the program failed, $y_f = f_{\bar{s}}/f$ is imputed to all units in the control group for which the program failed, and $y_p = 0$ is imputed to all units in both the treatment and control group for which the program passed. y_f implies that the higher the ratio the higher the probability of faultiness.

The results of using PO-2 in the control-dependence causal model are shown by the results of CD-2 in Tables 16 and 17. As the results show, CD-2 performs better than CD-1 on 48.69% of the faulty versions, worse on 1.96%, and neutral on 49.35%.

The minimum, median, maximum, and mean improvements of CD-2 over CD-1 are 0.07%, 3.23%, 29.47%, and 5.66%, respectively. Although Table 13 showed that CD-1 performed worse when compared to the Ochiai metric, Jaccard metric, and the F1-measure, Table 16 shows that CD-2 performs better when compared to the three metrics.

Overall, the results indicate that the choice of the potential-outcome function is important for accurate fault localization; the closer the potential outcomes are to modeling the actual outcomes of units of program elements the higher the accuracy of the causal estimates. The results also indicate that the potential-outcome functions potentially enables the integration of diverse information that effectively characterizes the behavior of computations at program elements. Note that PO-2 is a heuristic and that it has the potential to violate the SUTVA (Stable Unit Treatment Value Assumption) [46]. SUTVA means that the potential outcome of a unit i should not be influenced by the treatment of a unit j. In PO-2, depending on whether a unit is in the treatment or control group influences the potential outcome of another unit.

5.8.7 RQ4

The goal of RQ4 is to determine the scalability and efficiency of `Causal-CD`. To do this, we performed two timing studies on four large Unix subjects: Grep, Gzip, Sed, and Space. For the first timing study (preprocess stage), we measured the time it took for our technique to analyze all the gigabytes of trace data generated by the test suites of the subjects, constructed the dynamic control dependence graphs, and constructed the observational data for each node in the dynamic control dependence graph. For Grep, Gzip, Sed, and Space, worst times our technique took are approximately 4, 5, 2, and 0.07 hours, respectively. The second timing study (causal-inference stage) measured the time it took for LocalizeFault-DCDG to return a sorted list of failure-causing effects of statements to the developers. The LocalizeFault-DCDG

Table 16: Comparison of fault-localization models

Fault Loc. Tech.	Better (%)	Worse (%)	Neutral (%)
CD-2 vs CD-1	48.69	1.63	49.67
CD-2 vs Pr(F | s)	64.38	0.98	34.64
CD-2 vs Tarantula	64.38	0.98	34.64
CD-2 vs Ochiai	20.58	0.65	78.75
CD-2 vs Jaccard	33.99	0.65	65.36
CD-2 vs F_1-measure	33.99	0.65	65.36
CO-2 vs Ochiai	28.10	1.31	70.59
CJ-2 vs Jaccard	35.62	0.98	63.40
CF1-2 vs F_1-measure	38.56	1.31	60.13

Table 17: Distribution of "Better" improvements

Fault Loc. Tech.	Minimum (%)	Median (%)	Maximum (%)	Mean (%)
CD-2 vs CD-1	0.08	3.45	28.57	5.33
CD-2 vs Pr(F | s)	0.06	5.20	64.29	9.56
CD-2 vs Tarantula	0.06	5.20	64.29	9.56
CD-2 vs Ochiai	0.05	1.61	12.06	2.64
CD-2 vs Jaccard	0.05	3.35	16.55	4.45
CD-2 vs F1-Measure	0.05	3.35	16.55	4.45
CO-2 vs Ochiai	0.05	3.07	22.58	22
CJ-2 vs Jaccard	0.08	3.44	19.80	4.90
CF1-2 vs F1-Measure	0.05	4.56	62.10	7.50

algorithm took milliseconds to a few seconds to return a sorted list of causal effects to the developer. Note that these times depend largely on the number of test cases and the number of nodes in the dynamic program dependence graph as shown by the complexity measures in Section 4.3.4. Also, none of our algorithms have been optimized.

5.8.8 Threats to Validity

There are three main types of validity threats that affect our studies: internal, external, and construct. Threats to internal validity concern factors that affect dependent variables without the researchers' knowledge. There is the possibility that there might be errors in our implementation—specifically, in the process of generating dynamic control dependence graphs—that might affect the experimental results. To address

potential errors in the generation of the dynamic control dependence graphs, we compared manually generated dynamic control-dependence graphs of the functions of some of the test subjects to the dynamic control dependence graphs the technique generated automatically, to ensure that the dynamic control dependence graphs matched (which they did).

Threats to external validity occur when the results of our experiments cannot be generalized. Although we performed the empirical studies on 18 subjects with a total of 305 faulty versions, we cannot claim that the effectiveness of `Causal-CD` can be generalized to other faults in other software subjects. This is because we cannot claim to have accounted for all confounders affecting a given statement and the failure of a program. However, because our framework is theoretically motivated the threat to external validity is reduced. Also the results of our studies demonstrate the power of causal analysis as compared to non-causal suspiciousness metrics.

Threats to construct validity concern the appropriateness of the metrics used in our evaluation. It is difficult to determine how useful human developers will find ranking metrics, such as those we have studied, even if their accuracy is improved considerably. More studies need to be performed to address this issue and the issues of how best to present available fault-localization results and how to integrate them with other information useful for debugging. However, the more accurate fault-localization methods are, the more meaningful such studies are likely to be.

5.9 Discussion

The results of the empirical studies confirm that causal-inference techniques are relevant to fault localization, and that the causal-effect estimator $\hat{\tau}_{ls,s}$ is useful in itself as a fault-localization metric and is superior to estimators of the precision measure $\Pr(F|s)$. The studies also indicate that the recall measure $\Pr(s|F)$ is important in fault localization and that a measure that combines the causal-effect estimator with

an estimator for recall is superior to estimators based on precision alone and estimators that combines precision and recall.

Although the `Causal-CD` performed well in our studies, from a program-semantics perspective there are foreseeable and quite possibly common cases in which it should not perform well, namely faults that violate the coverage trigger assumption. In general, faults that on rare occasions cause failures may not be amenable to any type of coverage-based fault-localization technique.

To address such faults, it will be necessary to instantiate a causal model that identifies and controls for all confounders of a program element. Such confounders may include data dependences and the variable values they carry, not just control-dependence predecessors. In incorporating variable values, it may be necessary to partition (or "bin") the values of variables. In any case, using additional, non-binary predictors is likely to necessitate employing more sophisticated techniques of causal-inference methodology, such as matching/stratification and propensity scores [46].

Additionally, in our studies we observed two fundamental causal problems that contribute to the inability of `Causal-CD` to perform better on all faulty versions. The first problem is that there were certain cases in which a statement had only units in the treatment group or control group. This problem is referred to as the *overlap* problem. An example of this problem occurs if, for a given statement, all test cases whose execution reaches the forward control-dependence predecessor of the statement cover only that statement. From a causal-analysis perspective, it is impossible to reliably estimate the causal effect of a statement on program failure without making significant assumptions when this problem arises. In the implementation, we assumed that if a statement had units only in the treatment group or in the control group, the expected outcome of the group with no units is 0.

The second problem occurs when a statement has units in both the treatment and control groups but the units have different confounding variables or values (i.e.,

different characteristics). This problem is referred to as the *balance* problem. An example of this problem occurs if test cases that reach the forward control-dependence predecessor of a statement cover the statement and all other test cases do not reach the forward control-dependence predecessor. In this example, units of the statement in the treatment group will have the forward control-dependence predecessor as the confounder and units in the control group will have other confounders.

To address these two problems, ideally it is necessary to automatically generate new test cases. Another way to address this problem is to leverage the classical causal-analysis technique called *matching*, which we present in detail in the next chapter (Chapter 6 but in certain cases, matching may not be sufficient and must be combined with test-case generation. Note that in certain cases, it might not be possible to mitigate the overlap or balance problems because of program semantics. For example, the semantics of a program might be such that the true branch of an if-statement is mostly executed and the false branch is rarely or not executed. New techniques will need to be developed that help to mitigate such issues. For example, techniques may need to analyze groups of statements and how they causally relate to each other to handle this issue.

The causal framework addresses some of the limitations of the PPDG framework because instantiations of the framework analyze only causal graphs consisting of a program entity and its predecessors in the program dependence graph. Therefore, the framework can scale to large software subjects.

CHAPTER 6

COVARIATE MATCHING ON PROGRAM DEPENDENCES

In Chapter 4, we presented a causal framework for programs and showed how a causal model that uses only control dependences can be instantiated from the framework and used for accurate fault localization. However, it is easy to see that the previous model does not address all possible sources of confounding in fault localization, such as run-time patterns of data dependences (data flows), the values taken on by inputs and program variables, and non-determinism because of concurrency [7]. Making the best use of causal-inference methodology in software fault localization will require determining the relative importance of such factors and devising effective and reasonably efficient ways of controlling for them. As demonstrated in Chapter 3, static and dynamic data and control dependences have long been recognized as important factors in software testing and debugging, because they contribute both to triggering the effects of faults and to propagating those effects to a program's output [56, 59, 61].

In this Chapter, we present a new technique that is an instantiation of the causal framework that accounts and controls for local patterns of both dynamic data dependences and dynamic control dependences, which can confound the estimated causal effect of covering a program entity on the outcome of the program. This new technique uses information about dynamic data and control dependences to reduce confounding bias—more than is possible with our previous causal model that relied only control dependences—and to thereby rank statements more effectively for fault localization. This new technique also reveals an important problem in causal analysis called *balance* and how it affects our new technique. For effective causal analysis the units in

	t_1 (-1,1)	t_2 (2,3)	t_3 (1,-3)	t_4 (0,5)	t_5 (3,4)	$\hat{\tau}$	$\hat{\tau}_{match}$
void Proc2() {							
1 int x=read();	1	1	1	1	1	0.40	NA
2 int y=read();	1	1	1	1	1	0.40	NA
3 if(x > 0){	1	1	1	1	1	0.40	NA
4 if(y < 0)	0	1	1	0	1	0.67	0.67
5 y = 2;	0	0	1	0	0	-1.00	-1.00
6 print("Out:");	0	1	1	0	1	0.67	0.67
7 print(y+y); // y×y	0	1	1	0	1	0.67	1.00
8 }							
9 }							
	P	F	P	P	F		

Figure 24: Procedure with test cases, execution data, and causal-effect estimates. Error at statement 7; correct computation should be y×y.

the treatment group and control group must be balanced. That is the units in the two groups should have similar patterns of dynamic data and control dependences. In general units in the two groups should have similar characteristics. In practice, because the units in the treatment and control groups are generated by test cases or operational executions the units in the two groups cannot assumed to be balanced.

To achieve relative balance between the treatment and control groups and thus reduce confounding bias, our technique employs a classical causal-inference technique called *matching*. For each program entity e, matching reorganizes the units in the treatment and control groups so that they are similar with respect to dynamic data and control dependences that directly affect e. Each unit in the treatment group is *matched* with a unit in the control group that is most similar, as measured by applying a distance measure to the dynamic dependence information collected from the program's executions. After matching, the failure-causing effect of e is estimated from the reorganized data.

6.1 Motivating Example

Consider procedure `Proc2` in Figure 24, which has a fault at line 7. `Proc2` should print the square of the value of `y` at line 7 but instead prints two times the value of `y`. The first column shows `Proc2` with line numbers associated with each of its statements. Columns 2 through 6 represent test cases t_1–t_5, respectively. The top entry of each column shows the values of `x` and `y` that are read at lines 1 and 2, respectively. The numbers in the column for a test case indicate whether the corresponding program statement is covered by the test case (1 for covered, 0 for not covered). The bottom row shows the outcome of each test-case execution, with P indicating the procedure passed and F indicating that the procedure failed.

Suppose the causal effects of all the statements in `Proc2` are computed using the LocalizeFault-CDG algorithm given in Section 4.3.3. The column labeled $\hat{\tau}$ in Figure 24 indicates the causal-effect estimates obtained for the statements. The estimate for statement 7, which is faulty, is 0.67. However, the estimates for statements 4 and 6 are also 0.67, even though they are not faulty. Statements 4 and 6 have the same estimate as statement 7 because they are in the same control-dependence region [22] and hence, are covered by the same test cases. This example illustrates that serious confounding may occur even after conditioning on each statement's dynamic forward control-dependence predecessor.

Our new technique addresses the inadequacy of the control-dependence causal model for fault localization. The technique consists of two main components: an instantiation of our causal framework for program dependences and a matching technique.

6.2 Program-Dependence Causal Model

The program-dependence causal model extends our control-dependence causal model by addressing dynamic data dependences as well as dynamic control dependences.

Dynamic data dependences are important because they carry the values that are used at a given statement. Whereas a statement's forward control-dependence predecessor determines whether the statement is covered, the statement's data dependences determine the computation it propagates. Conceptually, the additional causal influences that we want to account for can be represented by including dynamic data dependences in the causal graph for a program, in addition to dynamic forward control dependences. A complication is that loop-carried data dependences [22] give rise to directed cycles in dependence graphs. However, to control confounding when estimating the causal effect of a particular program element e_i, it suffices to consider acyclic dependence chains terminating at e_i, because if there is a causal path from e_j to e_i that contains one or more cycles, there must also be a cycle-free path from e_j to e_i. For example, consider a program loop of the form

```
1 while(...)
2    if (...)
3       x = f(y);
4    else
5       y = g(x);
```

Each edge in the data dependence cycle 3→5→3 may be relevant to localizing a distinct fault. However, in seeking to control confounding while estimating the causal effect of statement 3, we can ignore the edge (3, 5). Similarly, when estimating the causal effect of statement 5, we can ignore the edge (5, 3).

The kind of causal graph that is required can be conceptualized in terms of the transitive reduction of a digraph [4]. A *transitive reduction* of a digraph D is an acyclic, spanning subdigraph H of D with no redundant arcs such that the transitive closures of D and H are isomorphic [9]. Klamt, Flassig, and Sundmacher employ a form of transitive reduction to model causality in biological networks with cycles [40].

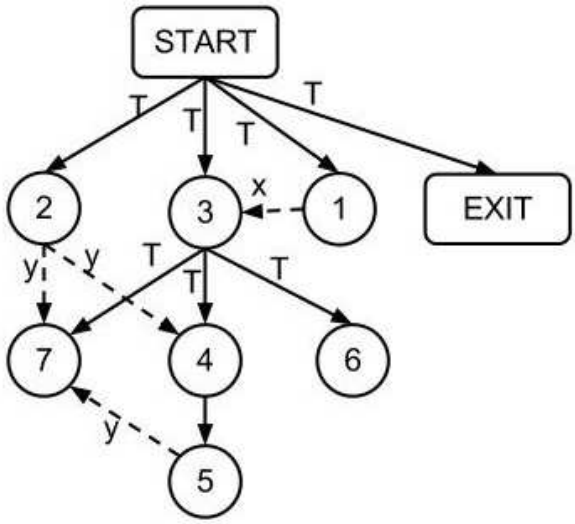


Figure 25: Dynamic-PDG of `Proc2`.

Aho, Garey, and Ullman [4] present a transitive reduction algorithm that, for a digraph with cycles, replaces each equivalence class of vertices appearing in the same cycle with a new vertex. Consider a slight variant of this algorithm, applied to a graph of dynamic data and control dependences between program statements. For a given statement s whose failure-causing effect we want to estimate, this variant preserves all nodes in s's equivalence class (if there is more than one) rather than collapsing them to a single node. However, it breaks any cycles involving s by deleting s's outgoing edges. Therefore, for each statement s in the dynamic program dependence graph, an acyclic subgraph H_s can be constructed that reflects all causal influences on s. H_s can be transformed into a proper causal graph by augmenting it with a potential-outcome node Y. We call the augmented H_s the integrated causal graph for s and denote it $ICG(s)$.

Definition 32. *An **integrated causal graph** is a graph obtained by augmenting H_s with the potential-outcome node Y.*

In the integrated causal graph $ICG(s)$, there will be multiple back doors to s if it is dependent on multiple statements. Observe, however, that the set of predecessors

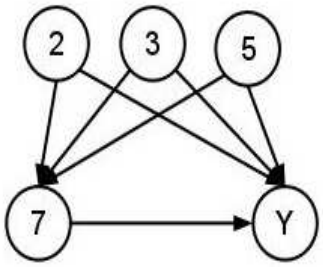


Figure 26: ICG of statement 7.

Table 18: Observational data gathered for Statement 7.

Program State	Unit	Treatment (T_7)	Confounders ($\hat{X}$)			Output (Y_7)
			S_2	S_3	S_5	
u_1	1	1	1	1	0	0
u_2	2	1	1	1	0	1
u_3	3	0	1	1	1	0
u_4	4	0	1	1	0	0
u_5	5	0	1	1	0	1

of s ($Pred(s)$) blocks all back-door paths from s to the potential-outcome node node Y. Thus, by conditioning on $Pred(s)$, we may be able to further reduce confounding when estimating the causal effect of s. For example, Figure 25 shows the dynamic program dependence graph of `Proc2`; dotted edges represent data dependences and solid edges represent control dependences. Figure 26 shows the ICG of statement 7, where nodes 2, 3, and 5 of statement 7 are nodes in the dynamic program dependence graph of `Proc`. As the graph shows, there are three back-door paths in the graph: $7 \leftarrow 2 \rightarrow Y$, $7 \leftarrow 3 \rightarrow Y$, and $7 \leftarrow 5 \rightarrow Y$. These back-door paths must be blocked to accurately estimate the causal effect of statement 7 on Y.

6.3 Matching

Matching [46] is a technique that brings some of the benefits of randomized controlled experiments, in terms of reduced confounding bias, to observational studies. Matching involves mapping (if possible) each treatment unit from an observational study to

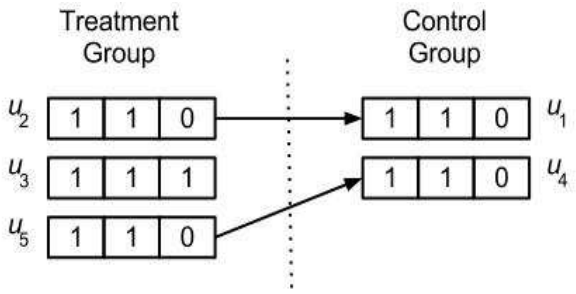


Figure 27: Units in treatment group and control group with their covariate values.

one or more control units that are similar to it, in such a way that *balance* is achieved between the resulting treatment group and control group with respect to covariate values. Matching may involve removing unmatched units or (in effect) duplicating certain units. We first describe a simple form of matching, called exact matching, before describing the more complex type of matching used with the program-dependence causal model.

In *exact matching*, each treatment unit is matched with one or more control units that have exactly the same covariate values as the treatment unit, if such control units exist. All matched treatment and control units are retained, and all unmatched units are discarded. The difference in the group means of the resulting treatment group and control group is an estimate of the treatment effect.

We now illustrate exact matching with reference to the procedure and data in Table 24 and Table 18, respectively. In the program-dependence causal model, the covariates for each statement s in procedure `Proc2` are the dynamic control and data dependence predecessors of s. For example, for statement 7, there are three predecessors: statement 2, which defines the value of `y` that may be used at statement 7; statement 3, which is statement 7's forward control dependence predecessor); and statement 5, which defines the value of `y` that may be used at statement 7. Table 18

shows the observational data gathered for statement 7 in `Proc2`. As the table shows, statement 7 has five units: three units in the treatment group because three program states (ps_2, ps_3, and ps_5) are covered by test cases (t_2, t_3, and t_5) and two units in the control group because program states ((ps_1 and ps_4) are covered by test cases (t_1 and t_4). Suppose the state of a covariate is one if it is covered by a test case and zero if it is not covered by a test case. Columns 4–6 in Table 18 shows a vector of covariates $\hat{X}$ for statement 7. The components of $\hat{X}$ are statements 2, 3, and 5. The vector of covariate values generated by the test cases is also shown for each program state (ps_i); a vector of the form [1, 1, 0] means that for that ps_i, statement 2 was covered, statement 3 was covered, and statement 5 was not covered, respectively.

Exacting matching works by first selecting a unit from the treatment group and then examines the control group to determine if there is a unit with a matching covariate vector. If such a unit exists in the control group it is matched to the unit from the treatment group and the two units are removed from consideration. Figure 27 shows the matches for the program states of statement 7. There is an arrow from each treatment unit to a matching control unit. Note that there is no match for t_3, because there is no unit in the control group to match with. Consequently, ps_3 is not used to estimate the causal effect of statement 7. Using Equation 17, the estimate for statement 7 is $(1 + 1)/2 - (0 + 0)/2 = 1.0$, which is the difference between the average outcome values for the treatment group and the average outcome values for the control group. The causal-effect estimates for the other statements are shown in Table 24, in the column labeled $\hat{\tau}_{match}$. (The NA for statement's 1, 2, and 3 reflects the fact that an estimate could not be computed for the statements because there were no control units for the statements.) The estimate for statement 5 is $0 - 1/1 = -1.0$ and the estimate for each of statements 4 and 6 is $(1 + 1 + 0)/3 - (0 + 0)/2 = 0.67$. It can be seen that the faulty statement, statement 7, has the highest estimate.

Unfortunately, as the number of covariates increases, it becomes more difficult

to find exact matches for treatment units. This difficulty results in more discarded units, which can increase bias (This form of bias is different from confounding bias. The bias is the difference in an estimator's expected value and the true value of the parameter being estimated.) when computing causal estimates. Therefore, other forms of matching are typically used.

6.3.1 Matching with Mahalanobis Distance

To obtain more flexibility in matching treatment units with control units, our technique uses the *Mahalanobis distance* metric [46], which is a measure of the similarity $d_M(\mathbf{a}, \mathbf{b})$ between two random vectors $\mathbf{a}$ and $\mathbf{b}$. Let S be the covariance matrix of $\mathbf{a}$ and $\mathbf{b}$ and let the superscript T denote matrix/vector transpose. Then

$$d_M(\mathbf{a}, \mathbf{b}) = \sqrt{(\mathbf{a} - \mathbf{b})^T S^{-1} (\mathbf{a} - \mathbf{b})} \tag{66}$$

The matrix S is the sample covariance matrix for the treatment and control units. Each treatment unit (test case covering statement s) with covariate vector $\mathbf{a}$ is matched with a control unit (test case not covering s) with covariate vector $\mathbf{b}$ for which $d_M(\mathbf{a}, \mathbf{b})$ is minimal with respect to a threshold. In matching *without replacement*, the two units are then removed from further consideration. In matching *with replacement*, the control unit is retained to be matched with another treatment unit. Replacement is normally used if the number of treatment units exceeds the number of control units; matching with replacement also reduces bias [65] in the causal estimate. A property of Mahalanobis-distance matching is that it regards all the component interactions of the covariate vector as equally important [65]. This property is important when matching on the predecessors of s because in the absence of any prior information about the relative importance of the predecessors, our technique treats all interactions of the predecessors of s as equally important.

```
   Input: Dynamic-PDG, Coverage-Info
   Output: Sorted τ̂
 1 foreach s ∈ Dynamic-PDG do
 2 |   τ̂(s) = -1.0;
 3 |   Model(s) = Compute causal model of s;
 4 |   Pred(s) = Compute predecessors of s from Pred(s);
 5 |   Mdata(s) = Match on Pred(s) in Coverage-Info;
 6 |   if Mdata(s) ≠ ∅ then
 7 |   |   Compute τ̂(s) from Mdata(s) using Equation (17);
 8 |   else
 9 |   |   Compute τ̂(s) using Equation (18);
10 |   end
11 end
12 Sort τ̂ in descending order;
```

Figure 28: The LocalizeFault-DPDG algorithm.

6.3.2 Matching based Causal Algorithm (`LocalizeFault-DPDG`)

Figure 28 shows the `LocalizeFault-DPDG` algorithm. The algorithm takes as input the dynamic program dependence graph (Dynamic-PDG) of a procedure and the coverage information (Coverage-Info) produced by executing the program containing the procedure on a set of test cases. `LocalizeFault-DPDG` processes each statement s in the Dynamic-PDG; it initializes the causal estimate of each statement s, ($\hat{\tau}(s)$), to -1.0 (line 2). `LocalizeFault-DPDG` computes the first major component of our technique, the causal model of s (Model(s)) (line 3). At line 4, the algorithm computes the predecessors of s (Pred(s)) from Model(s). After computing Pred(s), `LocalizeFault-DPDG` then computes the second major component of our technique by matching on the Pred(s) in the Coverage-Info to produce the matched data, Mdata(s) (line 5). If matching is successful, the $\hat{\tau}(s)$ is estimated using Equation (17) (line 7); if matching is not successful Equation (18) is used to estimate $\hat{\tau}(s)$ (line 9). The algorithm computes $\hat{\tau}(s)$ for every statement in the Dynamic-PDG. After the causal

estimate for each statement has been computed, `LocalizeFault-DPDG` sorts all the estimates in descending order at line 12. It then returns the sorted estimates at line 13 to the developer.

6.4 Empirical Evaluation

To evaluate the effectiveness of our technique for fault localization, we implemented it and performed empirical studies on a set of subjects. In this section, we first describe the set-up and then present the results of the studies.

6.4.1 Empirical Study Setup

For our studies, we used the same set of subjects in Table 12 in Section 5.8.1 except Gzip and Grep. We also omitted one version of Look because of the large amount of time required to gather all the execution data. We extended the implementation presented in Section 5.8.2 by computing dynamic data dependences and dynamic program dependences from dynamic control flow graphs. Recall that our technique uses dynamic control-flow graphs to compute dynamic dependences, because the former contain only statements that are actually executed by a set of test cases. We again implemented all fault-localization algorithms in *R* [58]. For matching on program dependences, we used the *R* package *Matching* [64]. We used matching with replacement and a default minimal distance of 0.00001. We also computed fault matrices that indicate, for each faulty program, which test cases pass and which test cases fail. We performed our studies on Mac OS X version 10.5.

6.4.2 Effectiveness Studies

To study the effectiveness of the program-dependence causal model for fault localization, we use the same cost metric (Cost) and procedure for comparing two fault-localization techniques presented in Section 5.8.3. Furthermore, to reduce the uncertainty in the causal effect computed for each statement because of the matching

technique, we computed a statement's causal effect 100 times and took the average of the 100 causal estimates. We found 100 to be an acceptable threshold for providing stable causal effects. Each computed causal effect has an associated standard error. We computed the average of the standard errors, and used it to construct a bound (one standard deviation) on the causal effect for each statement. For example, if the average of the causal effects of a statement on program failure is 0.3 and the average standard error is 0.1, then $[0.3 \pm 0.1]$ is the range of the causal effect over the 100 samples with 0.2 being the lower bound and 0.4 being the upper bound.

For brevity, we denote the program-dependence causal model for fault localization, which matches program states based on data and control dependences, as PD; we denote a variant of the technique that matches only on data dependences as DD, and we denote the control-dependence causal model [7], which considered only control dependences, as CD. We also use PDmin and PDmax to represent variants of PD computed using the lower bounds and upper bounds of the causal-effect estimates of PD, respectively. For example, if for a statement the average causal estimate obtained with PD is 0.3 and the average standard error is 0.1 then PDmin and PDmax yield causal estimates of 0.2 and 0.4, respectively.

Table 19 summarizes the results of comparing fault-localization techniques. The first column (Fault Loc. Tech.) shows the two fault-localization techniques (i.e., A vs B) that are being compared; the second technique in each pair (i.e., B) is the baseline. The second column (Positive (%)) shows the percentage of faulty versions for which A performed better than B, the third column (Negative (%)) shows the percentage of faulty versions for which A performed worse than B, and the fourth column (Neutral (%)) shows the percentage of faulty versions for which there was no improvement. For example, the first row, which compares PD to CD, shows that PD performed better than CD on 61.4% of the faulty versions, performed worse on 13.97% of the faulty versions, and performed identically on 24.63% of the faulty versions.

Table 19: Comparison of fault-localization models.

Fault Loc. Tech.	Positive (%)	Negative (%)	Neutral (%)
PD vs CD	61.40	13.97	24.63
PDmin vs CD	61.40	13.97	24.63
PDmax vs CD	60.29	14.71	25.00
PD vs DD	49.26	23.16	27.57
DD vs CD	43.75	25.74	30.51
PD vs Tarantula	72.06	7.72	20.22
PD vs Ochiai	44.12	21.32	34.56
CO vs Ochiai	53.68	10.29	36.03

Table 20: Distribution of positive improvements.

Fault Loc. Tech.	Minimum (%)	Median (%)	Maximum (%)	Mean (%)
PD vs CD	0.05	3.84	61.71	8.57
PDmin vs CD	0.05	4.03	61.71	8.75
PDmax vs CD	0.05	3.94	62.29	8.60
PD vs DD	0.03	3.43	45.71	6.11
DD vs CD	0.05	5.07	59.43	9.54
PD vs Tarantula	0.05	6.41	70.29	11.44
PD vs Ochiai	0.03	4.79	54.86	7.77
CO vs Ochiai	0.03	4.28	54.86	7.45

Table 20 shows the minimum, median, maximum, and mean values of A's improvement over B. Half the faulty versions with positive improvement values have improvements between the minimum and the median, and the other half have improvements between the median and the maximum. For example, the first row, which compares PD to CD, shows that half of the 61.4% of faulty versions with positive improvement values had improvements between 0.05% and 3.84% while the other half had improvements between 3.84% to 61.71%. The average positive improvement of PD over CD was 8.57%.

6.4.2.1 Study 1: New Technique vs Old Technique

The goal of this study is to compare the fault-localization effectiveness of our new technique PD to that of our previous causal technique CD, which considers control dependences but not data dependences. To do this, we compared the *Cost* values for

the two techniques.

Figure 29 shows the bar graph that summarizes the comparison of PD to CD over all program versions. The horizontal axis (baseline) represents the *Cost* of using our previous technique CD, the vertical axis represents the magnitude of improvement of PD over CD. The graph contains a vertical bar for each faulty version for which there was positive or negative improvement on the horizontal axis; the vertical bar shows the difference in *Costs*. The bars above the horizontal axis represent faulty versions for which PD performed better than CD (positive improvement) and the bars below the horizontal axis represent faulty versions for which PD performed worse than CD (negative improvement). For example, for faulty-version 3 on the graph, PD performed better by about 21.87%. Figure 29 shows that PD performed better than CD overall. As indicated in Table 19, PD performed better than CD on 61.4% of the faulty versions, worse on 13.97% of the faulty versions, and showed no improvement on 24.63% of the faulty versions. The first row of Table 20 characterizes the degree of positive improvement of PD over CD. As the table indicates, half the 61.4% of the faulty versions with positive improvement values had improvements between 0.05% and 3.84%, and the other half had improvements between 3.84% and 61.71%. The average positive improvement of PD over CD was 8.57%.

We also compared the *Cost*s of PDmin and PDmax to PD. As Table 19 shows, PDmin performed better than CD on 61.4% of the faulty versions, performed worse on 13.97% of the faulty versions, and performed identically on 24.63% of the faulty versions. The table also shows that PDmax performed better than CD on 60.29% of the faulty versions, performed worse on 14.71% of the faulty versions, and performed identically on 25% of the faulty versions. Table 20 shows that the performance of PD, PDmin, and PDmax are similar. The results of PDmin and PDmax provide evidence of the stability of the causal estimates computed with PD. Overall, the results indicate that PD improved the accuracy of fault localization over CD by further reducing

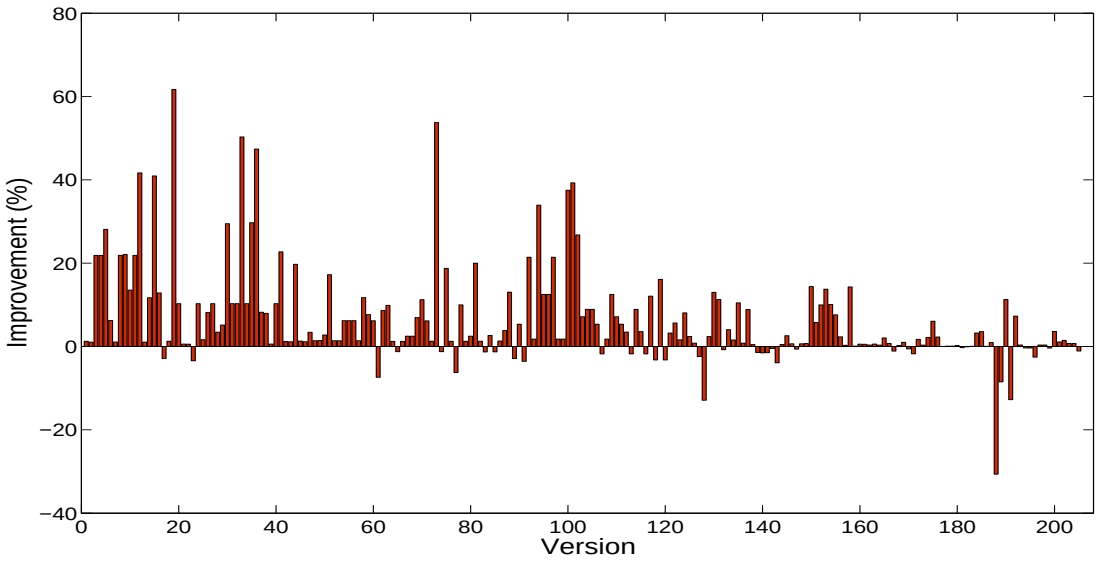
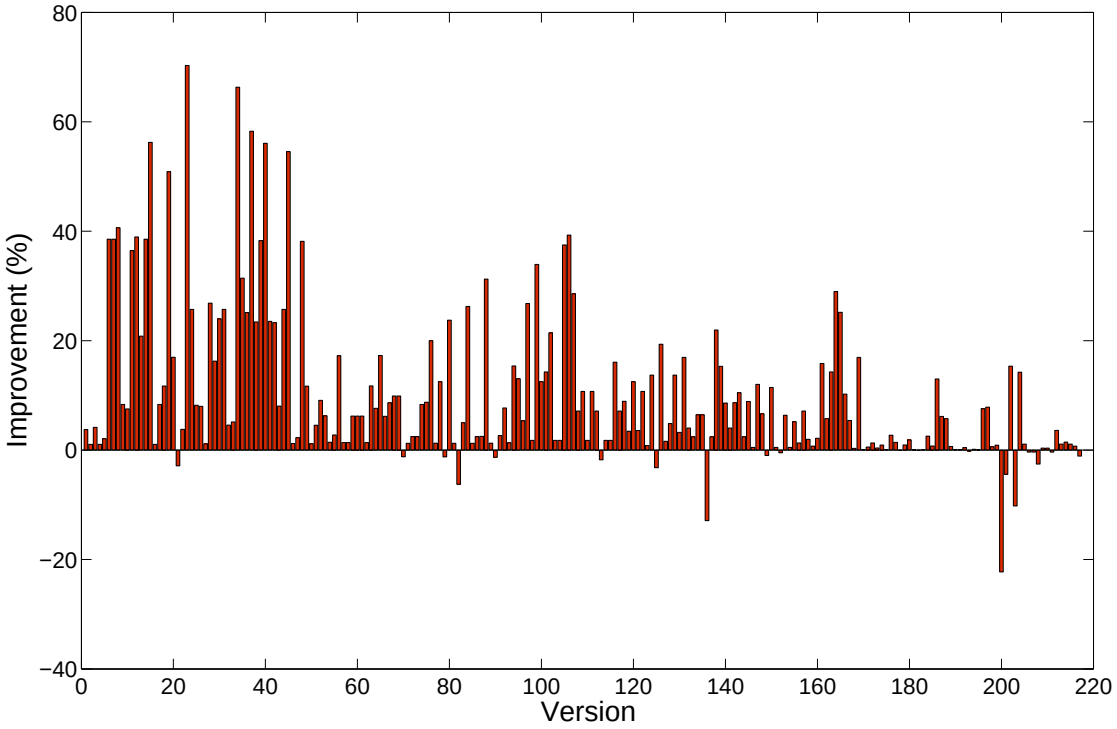


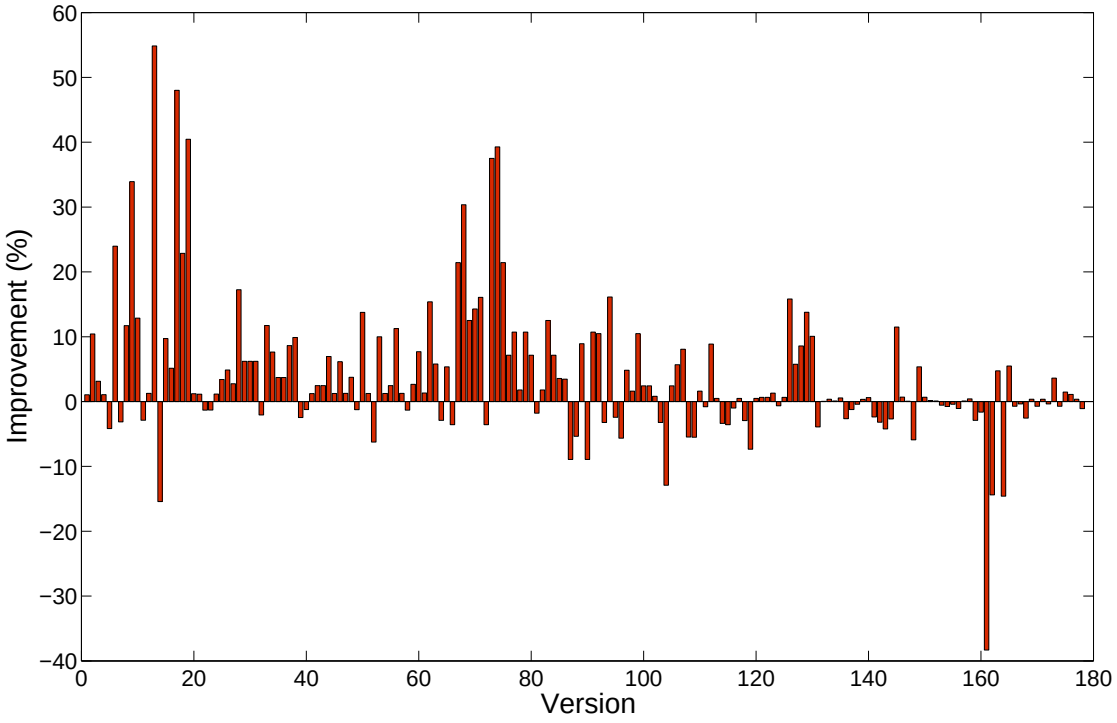
Figure 29: Comparison of new technique (Causal-PD) to old technique (Causal-CD).

confounding bias.

Although PD performed better overall, we wanted to see how much of the improvement resulted from data dependences. To do this, we compared DD to CD with respect to *Cost*. Table 19 indicates that DD had lower *Cost* than CD on 43.75% of the faulty versions and higher cost on only 25.74% of the versions. Thus most of the improvement seen with PD is due to considering data dependences. Considering both control and data dependences increases the accuracy of our technique by about 18%. Tables 19 and 20 indicate that causal models based on both data and control dependences are more effective than causal models based on either type of dependence alone. Overall, the results show that by blocking back-door paths created by program dependences in a faulty program, confounding bias can be reduced when estimating the failure-causing effect of a statement.



(a) Causal-PD vs Tarantula.



(b) Causal-PD vs Ochiai.

Figure 30: Comparison of new technique Causal-PD to Tarantula and Ochiai.

6.4.2.2 Study 2: New Technique vs Other Techniques

The goal of this study is to compare PD to two well-known statistical fault-localization techniques, Tarantula [38] and Ochiai [1], with respect to *Cost*. We used the *Costs* of Tarantula and Ochiai as baselines and subtracted the *Cost* of PD.

Figure 30(a) shows that PD performed better than Tarantula on most faulty versions. Table 19 indicates that PD performed better than Tarantula on 72.06% of the faulty versions, performed worse on 7.72% of the faulty versions, and performed identically on 20.22% of the faulty versions. Table 20 also shows that half the 72.06% with positive improvement values had improvements between 0.05% and 6.41% and that the other half had improvements between 6.41% and 70.29%. The average positive improvement of PD over Tarantula was 11.55%.

In our previous work [7], Ochiai performed better than our older technique (CD). However, Ochiai does not address confounding bias, so we incorporated CD into Ochiai (calling it Causal-Ochiai) and showed that Causal-Ochiai performed better than Ochiai. Here we show that PD performs better than Ochiai even when the techniques are not composed.

Figure 30(b) shows that PD performed better than Ochiai on most of the faulty versions. Table 19 indicates that PD performed better than Ochiai on 44.12% of the faulty versions, performed worse on 21.32% of the faulty versions, and performed identically on 34.56% of the faulty versions. Table 20 also shows that half the 44.12% of the faulty versions with positive improvement values had improvements between 0.03% and 4.79% and that the other half had improvements between 4.79% and 54.86%. The average positive improvement of PD over Ochiai was 7.77%. Tables 19 and 20 show that the performance of Ochiai was improved when PD was composed with it to produce Causal-Ochiai (CO). CO performed better on 53.68% of the faulty versions, performed worse on 10.29% of the faulty versions, and performed identically on 36.03% of the faulty versions. Also, half the 53.68% of faulty versions with positive

improvement values had improvements between 0.03% and 4.28% and the other half had improvements between 4.28% and 54.86%. The average positive improvement of CO over Ochiai was 7.45%.

Overall, the results indicate that PD performs better than both Tarantula and Ochiai, and improves Ochiai.

6.4.2.3 Computation Time

In this section, we present the average computation time required to compute all the fault-localization results for one version of some of the subjects. We found that the computation time was largely dependent on the size of the test suite of a program. The Matching package [64] takes considerable time to invert the large covariance matrices. For Tcas, which had 1608 test cases, it took on average 8 minutes. For Schedule, which had 2710 test cases, it took on average 32.73 minutes. For Printtokens2, which had 4115 test cases, it took on average 2.3 hours.

6.4.3 Threats to Validity

There are three main types of threats to validity that affect our studies: internal, external, and construct. Threats to internal validity concern factors that might affect dependent variables without the researcher's knowledge. The implementations of the algorithms we used in our studies could contain errors. The Matching package we used in our studies is open source and has been used by other researchers for experimentation, which provides confidence that the algorithms in the package are stable. To address potential errors when constructing the dynamic program-dependence graph, we compared manually generated dynamic program-dependence graphs of smaller subjects to graphs generated automatically by our technique.

Threats to external validity occur when the results of a study cannot be generalized. In this work, such threats are greatly alleviated because our work is based on established causal inference theory and methodology. However, more empirical

studies on additional subjects are needed to fully address this threat.

Threats to construct validity concern the appropriateness of the metrics used in our evaluation. We used the *Cost* metric to determine the effectiveness of our technique for fault localization. The *Cost* metric is a ranking metric that has been used to compare techniques in many fault-localization studies, though under a different name (*Score*). However, it is not established whether this metric is well suited for presenting fault-localization information to developers.

6.5 Discussion

We have presented a new technique that combines a program-dependence causal model, an instantiation of the causal framework, with covariate matching (a classical causal analysis technique). The program dependence causal model is combined with covariate matching to obtain more accurate (less biased) estimates of a given statement's effect on the occurrences of program failures. The use of both covariate matching and data-dependence information in our technique are innovations. The empirical results indicate that the new technique is more effective overall than other techniques—including the control-dependence causal-model technique, which does not consider data dependences.

Although the presented technique performed well overall, it performed relatively poorly on some faulty program versions, which indicates there is room for improvement. We mention three possible reasons for the technique's performance in some cases, and we suggest how it might be improved.

First, the results obtained with the new technique are subject to limitations of the test suite used for fault localization. Even if the test suite has desirable properties overall, it may be inadequate for accurately estimating the causal effect of certain statements. This will be the case for any statement that is covered by very few test cases or for which it is not possible to extract two reasonably well-matched

comparison groups of test cases that respectively cover and do not cover the statement. Suspiciousness scores computed for such statements cannot be trusted because they lack adequate support. To address this issue, we plan to investigate the automatic generation of test cases with the appropriate coverage characteristics.

Second, although matching using Mahalanobis distance was generally effective in our studies, it sometimes failed to yield balanced comparison groups. In the latter cases, the rankings produced by our technique are not consistent because they are based on valid causal estimates mixed with biased estimates. Using Mahalanobis distance for matching is also sometimes inefficient, because it is expensive to compute Mahalanobis distance when there are many program dependences. To address these issues, we plan to explore the use of other matching techniques, such as those based on propensity scores [46].

Finally, some faults cannot be effectively localized without considering the variable values carried by data dependences. For example, considering variables values may be the only way to determine that a check for an unusual condition is missing in a certain program location. Although the proposed technique considers data dependences it does not currently consider variable values.

CHAPTER 7

CONCLUSIONS

Current statistical fault-localization have resorted to borrowing, adapting, and applying metrics from other fields such as statistics to the fault-localization problem. There is nothing inherently wrong with these metrics except that the metrics were not developed to address the problem of causality. This dissertation presents a novel approach to finding the causes of program failures based on causal-inference theory.

We presented a unifying framework that combines statistical, program analysis, and causal analysis to enable effective reasoning (causal and probabilistic) over program behaviors. We showed in the dissertation that in some cases the causes of program failures can be identified from dynamic information gathered from programs. We also presented a discussion as to why in some cases the causes of program failures could not be reliably estimated. We unified the current statistical fault-localization metrics by showing analytically that they were all related; we also showed analytically that the metrics were biased and as such the metrics found associations between program elements and program failures instead of finding the program elements that caused the failure. We also presented empirical results on several software subjects with many faulty versions. We compared the causal models instantiated from our causal framework to current statistical fault-localization techniques. Our results showed that our causal models performed significantly better than the current techniques.

7.1 Merit

The dissertation research makes a number of contributions to the field of software engineering. First, the dissertation presents a novel program representation that combines statistical information with program-analysis information in a principled

manner to reason (probabilistically and causally) about program behaviors. We applied the representation known as the probabilistic program dependence graph to the problem of fault localization, and showed through numerous empirical studies the effectiveness of the technique.

The dissertation presents a theoretical causal framework for fault localization that combines program-dependence information with causal inference methodology for observational studies. We instantiated several causal models from our causal framework and demonstrated empirically the effectiveness of our approach over current fault-localization techniques. Our theoretically-based approach provides the foundation on which to build future fault-localization systems. Also, theoretical nature of our approach will ensure that future fault-localization systems built on our approach will not be susceptible to severe external validity problems. The increase in fault-localization accuracy will result in reduced time spent by developers during debugging, which, in turn, will lead to higher software quality.

Third, the dissertation presents a unification of the current statistical fault-localization metrics from a causal perspective. It presents an analysis of the behaviors of the metrics to assess the limitations inherent in the metrics from a causal perspective. Exposing the limitations of current statistical fault-localization metrics will direct future researchers to avoid pitfalls of the current metrics.

7.2 Future Work

This research provides many directions for possible future research. We discuss the future directions in four areas.

7.2.1 Causal Program Analysis

Program analysis is an approach to automatically reason about the behaviors of programs, for example, for program understanding or optimization. There are two main

types of program analysis: static analysis and dynamic analysis. Static analysis provides may/must information about program behaviors and dynamic analysis provides must information about a subset of program behaviors. However, the uncertainty inherent in many software-engineering problems requires an analysis that can handle this uncertainty. We intend to explore another kind of analysis, which we call causal program analysis. The goal of causal program analysis is to bridge the gap between static and dynamic analyses to handle the uncertainty in program behaviors. This analysis, which is observational and experimental in nature, will seek to combine static, dynamic, and statistical information in a principled way to reason causally about program behaviors. For example, causal program analysis can help fuse information from different sources such as developer knowledge, project history, and program analysis.

7.2.2 Software Debugging

This dissertation presents the foundation necessary for future research in the area of debugging. We view the dissertation work as a first step in addressing the larger fault-localization problem, that is to eventually answer the question "Why did my software fail?".

- One of the many important features of the causal framework is that the framework allows programmers to interact with models instantiated from the framework. This feature is useful because developers can provide the model with information that is not available through static or dynamic analysis. Another important feature is that the causal framework can determine whether any of its estimates are precise. One possible usage scenario is that the causal model initially presents the developer with some result, and the developer then analyzes the result and provides feedback to the model. The model then uses the feedback from the developer to refine the result.

- For the causal aspect of the dissertation, we focused on program dependences in our causal framework. However, for more effective analysis, we intend to incorporate different kinds of information such as variable values into the framework. For example, incorporating variable values into the framework will require the development of effective value-abstraction techniques.

- For fault localization to be useful to the developer, techniques should be able to generate explanations. This research makes it possible to generate causal explanations from dynamic information gathered from programs. These causal explanations can be in the form of causal graphs.

- In this dissertation, we only focused on single threaded programs. In the future we intend to extend our framework to concurrent programs. As a starting point, we intend to use concurrent program dependence graphs as the underlying representation from which we compute our causal causal graphs.

- In the future, we intend to focus on diverse kinds of large software systems such as database systems, e-commerce applications, and mobile platforms. In this dissertation we only focused on the C language, however, we intend to extend our approach to other languages such as C++, Java, Fortran, and Ada.

7.2.3 Software Testing

One of the important problems in software testing is regression testing. Regression testing is the testing of software systems to uncover faults after changes have been made to the system. One of the key insights provided by my current research is that software testing can be viewed as an experimental causal-analysis approach. The goal of testing then is to reveal how a program element(s) causally affect the outcome of the program. This causal insight potentially provides a way to determine and to construct test suites that reveal how program entities causally affect the outcome

of the software. Furthermore, the insight has the potential of addressing perennial problems in regression testing such as test-suite augmentation, selection, and prioritization. Another potential application is that several metrics developed for testing such as statement coverage or branch coverage, can potentially be unified from a causal perspective.

7.2.4 Change Impact Analysis

The goal of change impact analysis is to determine the parts of a software system that have been affected because of a change(s) to some other part(s) of the software. Our causal framework provides a unique opportunity to address novel problems posed by change impact analysis.

- A problem in change impact analysis is to quantify the importance of the affected parts of the software. Quantifying affected parts of the software will enable developers to focus their limited resources on testing the affected parts of the software that are deemed important according to some metric. Viewing the quantification problem from a causal perspective potentially provides a way to address this problem.

- Another problem is how to incorporate impacts of changes made to the software into the causal framework for effective reasoning. How will the causal graphs look like when changes are incorporated into the framework.

7.2.5 Usability Studies

The techniques that have been developed in this dissertation will ultimately be used by developers. Therefore there is the need to perform usability studies to determine the usability and effectiveness of the techniques in practice. An example of a study is to allow developers to interact with the causal framework by specifying *a priori* information they might have about a particular failure. The causal framework is

then supposed to use the information to generate more accurate results, which will be judged by the developers.

REFERENCES

[1] Abreu, R., Zoeteweij, P., and van Gemund, A. J., "On the Accuracy of Spectrum-Based Fault Localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, pp. 89–98, September 2007.

[2] Agrawal, H., *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, August 1991.

[3] Agrawal, H. and Horgan, J. R., "Dynamic Program Slicing," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pp. 246–256, ACM, 1990.

[4] Aho, A. V., Garey, M., and Ullman, J. D., "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, pp. 131–137, 1972.

[5] Angrist, J. D. and Pischke, J., *Mostly Harmless Econometrics: An Empiricist's Companion*. Princeton University Press, 2008.

[6] Baah, G. K., Podgurski, A., and Harrold, M. J., "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis," in *Proceedings of International Symposium for Software Testing and Analysis*, July 2008.

[7] Baah, G. K., Podgurski, A., and Harrold, M. J., "Causal Inference for Statistical Fault Localization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 73–84, July 2010.

[8] Banerjee, A., Roychoudhury, A., Harlie, J. A., and Liang, Z., "Golden implementation driven software debugging," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pp. 177–186, 2010.

[9] Bang-Jensen, J. and Gutin, G., *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2001.

[10] Bates, S. and Horwitz, S., "Incremental Program Testing using Program Dependence Graphs," in *Proceedings of Symposium on Principles of Programming Languages*, pp. 384–396, January 1993.

[11] Bishop, C. M., *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] Burger, M. and Zeller, A., "Minimizing reproduction of software failures," pp. 221–231, 2011.

[13] Burnell, L. and Horvitz, E., "Structure and Chance: Melding Logic and Probability for Software Debugging," *Communications of the ACM*, vol. 38, no. 3, pp. 31–41., 1995.

[14] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *In Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[15] Cheng, H., Lo, D., Zhou, Y., Wang, X., and Yan, X., "Identifying Bug Signatures Using Discriminative Graph Mining," in *Proceedings of the International Symposium on Software Testing and Analysis*, July 2009.

[16] Cleve, H. and Zeller, A., "Locating Causes of Program Failures," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, pp. 342–351, May 2005.

[17] Cochran, W. G., "The planning of observational studies of human populations," *Journal of Royal Statistical Society*, vol. 182, pp. 234–255, 1965.

[18] DeMillo, R. A., Pan, H., and Spafford, E. H., "Critical Slicing for Software Fault Localization," in *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 121–134, 1996.

[19] Do, H., Elbaum, S., and Rothermel, G., "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[20] Engler, D., Chelf, B., Chou, A., and Hallem, S., "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000.

[21] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B., "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[22] Ferrante, J., Ottenstein, K. J., and Warren, J. D., "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[23] Fisher, R. A., *The Design of Experiments*. 1935.

[24] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R., "Extended Static Checking for Java," in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2002.

[25] Galan, S., Aguado, F., F.J.Diez, and Mira, J., "NasoNet, Joining Bayesian Networks, and Time to Model Nasopharyngeal Cancer Spread," *Artificial Intelligence in Medicine*, vol. 2101/2001, pp. 207–216, 2001.

[26] Gallagher, K. B. and Lyle, J. R., "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17, pp. 751–761, 1991.

[27] Guo, S. and Fraser, M. W., *Propensity Score Analysis*. SAGE, 2010.

[28] Gyimóthy, T., Beszédes, A., and Forgács, I., "An Efficient Relevant Slicing Method for Debugging," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 303–321, 1999.

[29] Hangal, S. and Lam, M., "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *Proceedings of the 24th international conference on software engineering*, pp. 291–301, May 2002.

[30] Heckerman, D., Chickering, D. M., Meek, C., Rounthwaite, R., and Kadie, C. M., "Dependency Networks for Inference, Collaborative Filtering, and Data Visualization," *Journal of Machine Learning Research*, vol. 1, pp. 49–75, 2000.

[31] Holland, P. W., "Statistics and Causal Inference," *Journal of American Statistical Association*, vol. 81, pp. 945–970, 1986.

[32] Hovemeyer, D. and Pugh, W., "Finding Bugs is Easy," in *ACM SIGPLAN Notices*, 2004.

[33] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria," in *Proceedings of the International Conference on Software Engineering*, pp. 191–200, May 1994.

[34] Imbens, G. W., "Nonparametric Estimation of Average Treatment Effects Under Exogeneity: A Review," *Review of Economics and Statistics*, vol. 86, no. 1, pp. 4–29, 2004.

[35] Jeffrey, D., Gupta, N., and Gupta, R., "Fault Localization Using Value Replacement," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, (New York, NY, USA), pp. 167–178, ACM, 2008.

[36] Jiang, L. and Su, Z., "Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths," in *Proceedings of the International Conference on Automated Software Engineering*, pp. 184–193, November 2007.

[37] Jones, J. and Harrold, M. J., "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," in *Proceedings of the International Conference on Automated Software Engineering*, pp. 273–282, November 2005.

[38] Jones, J., Harrold, M. J., and Stasko, J., "Visualization of Test Information to Assist Fault Localization," in *Proceedings of the International Conference on Software Engineering*, pp. 467–477, May 2002.

[39] Jose, M. and Majumdar, R., "Cause Clue Clauses: Error Localization using Maximum Satisfiability," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pp. 437–446, 2011.

[40] Klamt, S., Flassig, R. J., and Sundmacher, K., "TRANSWESD: inferring cellular networks with transitive reduction," *Bioinformatics*, vol. 26, pp. 2160–2168, 2010.

[41] Laski, J. W. and Korel, B., "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 9, pp. 347–354, 1983.

[42] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable Statistical Bug Isolation," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 15–26, June 2005.

[43] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P., "SOBER: Statistical Model-based Bug Localization," in *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 286–295, September 2005.

[44] Lucia, Lo, D., Jiang, L., and Budi, A., "Comprehensive Evaluation of Association Measures for Fault Localization," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 2010.

[45] Manning, C. D., Prabhakar, and Schutze, H., *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[46] Morgan, S. L. and Winship, C., *Counterfactuals and Causal Inference: Methods and Principles of Social Research*. Cambridge University Press, 2007.

[47] Murphy, K., *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, 2002.

[48] Naish, L., Lee, H. J., and Ramamohanarao, K., "A Model for Spectra-based Software Diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 11:1–11:32, Aug. 2011.

[49] Neapolitan, R. E., *Learning Bayesian Networks*. Prentice Hall, 2003.

[50] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W., "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proceedings of the International Conference on Compiler Construction*, pp. 213–228, April 2002.

[51] Neyman, J. S., "On the Application of Probability Theory to Agricultural Experiments. Essay on Principles, Section 9," *Statistical Science*, vol. 5, pp. 465–480, 1923.

[52] Parnin, C. and Orso, A., "Are Automated Debugging Techniques Actually Helping Programmers?," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2011)*, (Toronto, Canada), pp. 199–209, July 2011.

[53] Pearl, J., *Causality: Models, Reasoning, and Inference*. San Francisco, CA, USA: Cambridge University Press, 2000.

[54] Pearl, J., "An Introduction to Causal Inference," *The International Journal of Biostatistics*, vol. 6: Iss. 2, Article 7, 2010.

[55] Pearl, J. and Verma, T., "A Theory of Inferred Causation," *In J.A Allen, R. Fikes, and E. Sandewall (Eds.), Principles of Knowledge Representation and Reasoning: Proceeding of the 2nd International Conference*, pp. 441–452, 1991.

[56] Podgurski, A. and Clarke, L. A., "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions on Software Engineering*, vol. 16, pp. 965–979, September 1990.

[57] Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K., "Darwin: An Approach for Debugging Evolving Programs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pp. 33–42, 2009.

[58] R Development Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[59] Rapps, S. and Weyuker, E., "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. SE-11, NO. 4, pp. 367–375, 1985.

[60] Renieris, M. and Reiss, S., "Fault Localization With Nearest Neighbor Queries," in *International Conference on Automated Software Engineering*, pp. 30–39, November 2003.

[61] Richardson, D. J. and Thompson, M. C., "An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection," *IEEE Transactions on Software Engineering*, vol. 19, no. 6, pp. 533–553, 1993.

[62] Rubin, D., "Estimating Causal Effects of Treatments in Randomized and Nonrandomized Studies," *Journal of Educational Psychology*, vol. 66, pp. 688–701, 1974.

[63] Rubin, D. B., "Teaching statistical inference for causal effects in experiments and observational studies," *Journal of Educational and Behavioral Statistics*, vol. 29, pp. 343–367, 2004.

[64] Sekhon, J. S., "Multivariate and Propensity Score Matching Software with Automated Balance Optimization: The Matching Package for R," *Forthcoming, Journal of Statistical Software*, 2008. Forthcoming.

[65] Stuart, E. A., "Matching Methods for Causal Inference: A Review and a Look Forward," *Statistical Science*, vol. 25, pp. 1–21, 2010.

[66] Tassey, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing," *National Institute of Standards and Technology, RTI Project Number 7007.0011*, 2002.

[67] Thrun, S., "Robotic Mapping: A Survey," *In Exploring Artificial Intelligence in the New Millennium*, pp. 1–35, 2002.

[68] Wasserman, L., *All of Statistics*. Springer, 2003.

[69] Weiser, M., "Program Slicing," in *Proceedings of the International Conference on Software Engineering*, pp. 439–449, March 1981.

[70] Wong, W. E., Debroy, V., and Choi, B., "A Family of Code Coverage-Based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, 2010.

[71] Yu, Y., Jones, J. A., and Harrold, M. J., "An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 201–210, 2008.

[72] Zeller, A., "Isolating Cause-Effect Chains from Computer Programs," in *Proceedings ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, November 2002.

[73] Zhang, X., Gupta, R., and Gupta, N., "Locating Faults through Automated Predicate Switching," in *Proceedings of the 28th International Conference on Software Engineering*, May 2006.

[74] Zhang, X., Gupta, N., and Gupta, R., "Pruning Dynamic Slices With Confidence," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 169–180, June 2006.

VITA

George was born in 1976 in Accra, Ghana. He emigrated to the United States of America in 1997. He attended Pace University in New York and graduated (Magna Cum Laude) in 2001 with a B.S. in Computer Science and a minor in Mathematics. He went to work in industry for a year and in 2003 he began his Ph.D studies at Georgia Institute of Technology. In 2005, he started working under the advisement of Mary Jean Harrold. He wil receive his Ph.D in Computer Science in June 2012.