Dynamic Impact Analysis:
Analyzing Error Propagation In Program Executions

Tarak Goradia

November 1993

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University

Approved: _____

Prof. Elaine Weyuker

Research Advisor

Acknowledgements

I am grateful to my advisor Prof. Elaine Weyuker for suggesting the validation experiment, for her careful reviews of this work and for patiently listening to the initial ideas about impact analysis. I am thankful to Prof. Allan Gottlieb and Prof. Ben Goldberg for their constructive comments on the thesis proposal. I am also thankful to Thomas Ostrand, Pat Vroom, and Thomas Murphy of Siemens Corporate Research, Inc. for supporting this research effort.

I am indebted to Michael Greenberg for providing the *Frame* system used for developing the prototype and for his help in resolving the problems related to Lisp and the Frame system. I am thankful to Michael Platoff for his advice on system interface issues, Monica Hutchins for being a good listener during the initial development of the impact analysis framework, Jean Hartmann for furnishing references from his personal library, Dilip Soni for technical discussions related to static dependencies, Bill Landi for his help in determining the correct probability and complexity expressions, Herb Foster and Maryam Shahraray for their insights into statistics and Amitava Datta for writing the control monitor program that led to an efficient use of the available computing resources for running the experiments.

I thank Leonor Abraído-Fandiño, Vivek Agrawal, Thomas Ostrand, and Jean Hartmann for their comments on an earlier version of this dissertation. I also thank my officemates Amitava Datta and Steve Masticola for listening to my dumb questions.

Abstract

Test adequacy criteria serve as rules to determine whether a test set adequately tests a program. The effectiveness of a test adequacy criterion is determined by its ability to detect faults. For a test case to detect a specific fault, it should execute the fault, cause the fault to generate an erroneous state and propagate the error to the output. Analysis of previously proposed code-based testing strategies suggests that satisfying the error propagation condition is both important and expensive. The technique of *dynamic impact analysis* is proposed for analyzing a program execution and estimating the error propagation behavior of various potential sources of errors in the execution. Impact graphs are introduced to provide an infrastructure supporting the analysis. A program impact graph modifies the notion of a program dependence graph proposed in the literature in order to capture some of the subtle impact relationships that exist in a program. An execution impact graph represents the dynamic impact relationships that are demonstrated during a program execution. The notion of *impact strength* is defined as a quantitative measure of the error sensitivity of an impact. A cost-effective algorithm for analyzing impact relationships in an execution and computing the impact strengths is presented. A research prototype implemented to demonstrate the feasibility of dynamic impact analysis is briefly described. The time complexity of dynamic impact analysis is shown to be linear with respect to the original execution time, and experimental measurements indicate that the constant of proportionality is a small number.

The experiments undertaken to validate the computation of impact strengths are presented. An experience study relating impact strengths to error propagation in faulty programs is also presented. The empirical results provide evidence indicating a strong positive correlation between impact strength and error propagation. The results also emphasize the need for better heuristics to improve the accuracy of the error propagation estimates. Potential applications of dynamic impact analysis to mutation testing, syntactic coverage-based testing and dynamic program slicing are discussed.

Contents

List of Figures

Chapter 1

Introduction

One of the important problems in software testing is deciding when a program has been tested "enough". The cost of fixing a bug in a program increases rapidly as the program passes through the different phases of software development and maintenance. Software development managers often have to strike a balance between the cost of software testing during development and the cost of bug fixing during maintenance. They turn to the software testing research community for providing *test adequacy criteria* that enable them to decide whether a software has been adequately tested.

Test adequacy criteria proposed in the literature require the use of information available at various stages of software development, such as specification, design or coding. Based on the development stage which provides the information, testing approaches are categorized as being specification-based, design-based or code-based. Researchers argue that each approach has its unique advantages and none of these approaches can replace any of the others [66, 22]. Also, testing strategies using the same approach may differ in the nature and amount of information used. For example, in the specification-based approach, random testing methods use only the input domain specification for selecting test cases, while the category partition method [55] uses information from the functional

specification to partition the input domain into equivalence classes and selects test cases from each of the classes.

The research described in this dissertation focuses on test adequacy criteria using the code-based testing approach. The primary motivation behind this research was to address some of the problems related to *effectiveness* and *computational feasibility* in previously proposed code-based testing strategies. The former is determined by the fault detection capability of the associated adequacy criterion. The latter refers to the feasibility with respect to the processor and memory resources consumed for the execution and analysis of test cases used by the tester while following the strategy. In reality, the feasibility of a testing strategy also depends on the human effort required in satisfactorily resolving the undecidable problems associated with the strategy. This research does not address such feasibility issues related to undecidable problems.

This chapter summarizes the research described in this dissertation. We first present a brief overview of the problem and informally describe our approach. Then we summarize our contributions and describe the organization of this document.

1.1 Problem Overview

The effectiveness of a testing strategy is determined by its ability to detect faults. We use a model of fault detection based on similar models defined by Morell [45], Richardson and Thompson [67] and Offutt [53]. According to this model, for a test case execution to detect a specific fault, it should meet three conditions: the *fault execution condition*, which causes the fault to be executed, the *error creation condition*, which causes the fault execution to generate an incorrect intermediate state, and the *error propagation condition*, which enables the errors in the incorrect intermediate state to propagate and cause an incorrect output. This model assumes that it is possible to identify an intermediate state as being incorrect. The *errors* in an incorrect state refer to the values

associated with the incorrect state variables.[1] Test adequacy criteria based on syntactic coverage do not explicitly enforce the above conditions for any faults. Instead, they hope that the execution of key syntactic components of a program will lead to the detection of the faults associated with those components. On the other hand, the adequacy criteria based on strong mutation analysis [10, 13] require that for each specified fault there be at least one test case which meets all of the above conditions for detecting that fault. However, the cost of strong mutation analysis is extremely high. In order to reduce the cost, weak mutation testing [27] was proposed. It does not require the satisfaction of the error propagation condition and instead relies on the hope that the error propagation condition will be satisfied when the other two conditions are satisfied. In this thesis, we investigate the problem of satisfying the error propagation condition.

1.2 Problem Analysis and Our Solution Approach

An error in an incorrect state may fail to propagate to the output of the execution if each of the computations using the incorrect state either masks the error or does not affect the output. A computation exhibiting the error masking behavior is typically an implementation of some many-to-one function. Voas and Miller [73] defined a metric called the *domain/range ratio* (DRR) to estimate the error propagation characteristics of many-to-one functions. The main drawback of this metric is its static nature. It fails to account for the fact that different invocations of a function could have different error propagation characteristics. In order to study the dynamic characteristics of error propagation, we designed a technique to analyze propagation of errors during program executions.

Consider an execution $\mathcal{T}$ of a program $\mathcal{P}$. A fault in the program may introduce

[1] The definitions of *state* and *state variables* can be found in section 2.1. Also, we acknowledge that the term "error" has several different meanings and refer the reader to section 2.2 for a discussion.

errors in several instances of the state variables during the execution. Therefore, each instance of a state variable can be looked upon as a potential source of error. Let S be a state in the execution, and x be a potential source of error in that state. Assume that we are interested in examining the propagation behavior of $\varepsilon(x)$, a set of *plausible errors* for x. We say that an error is plausible if it could occur as a result of a likely fault in the program. If we randomly choose an error from $\varepsilon(x)$ and modify the execution $\mathcal{T}$ by introducing the error in x, what is the chance that the error will propagate to the output of the modified execution? This probability, p_x, gives a measure of the sensitivity of the output to $\varepsilon(x)$, the set of plausible errors for x. When p_x is high (near 1.0), it means that the output is very sensitive to an error in x. And when p_x is low (near 0.0), it means that the output is very insensitive to an error in x. We can relate this with the correctness of the value of x as follows.

- If the output is sensitive to an error in x, and if the output is correct, then the value of x is likely to be correct and any fault that would have caused an error in x is likely to be absent.
- However, if the output is insensitive to an error in x, we cannot say anything about the correctness of the value of x.

Note that p_x may depend on, among other things, the values of some of the variable instances in the state S. A fault may actually create related errors in two or more variable instances in the state. Therefore, in order to truly understand the propagation behavior of errors caused by real faults, one should consider combined error sets for various groups of state variable instances. However, the number of such groups to be considered could be as large as the size of the power set of the state variable instances. Given the magnitude of this problem, we focus only on the simpler problem of computing the probability p_x for every possible source of errors x in the execution. There are three major issues in addressing this simpler problem. First, it is not clear how to identify

the set of plausible errors for x. Second, the computation of p_x could be very expensive. Third, the number of such x's in an execution could be very large. The framework of dynamic impact analysis presented in this thesis attempts to address these issues. The notions of *program impact* and *impact paths* are defined to capture the different ways in which a potential source of error can affect the output. A pragmatic method for approximating the sets of plausible errors is described. The notion of *impact strength* of x is defined as an approximation of the probability p_x and a cost-effective algorithm is designed to compute the impact strengths of x.

1.3 Summary of Contributions

New Concepts

A *program impact graph* modifies the notion of a program dependence graph proposed in the literature in order to capture some of the subtle impact relationships that exist in a program. An *execution impact graph* represents the *dynamic* impact relationships that are demonstrated during a program execution. The notion of *impact Strength* is defined as a quantitative measure of the error sensitivity of an impact. Dynamic impact analysis is proposed as a cost-effective technique to analyze the impact relationships in an execution and compute the impact strengths.

Prototype Tool

A prototype tool, DIAna, generates the impact graph of a program, carries out the impact analysis of a program execution and provides an infrastructure for conducting the validation experiments.

RESULTS

Empirical studies provide evidence indicating a strong positive correlation between impact strength and error propagation. The time complexity of dynamic impact analysis is shown to be linear with respect to the original execution time. Experimental measurements indicate that the impact analysis of an execution is about 2.5 to 14.5 times slower than the original execution. Mutation testing, syntactic coverage-based testing and dynamic program slicing are identified as potential applications for dynamic impact analysis.

1.4 Organization of Dissertation

Chapter 2 describes some of the basic terms and concepts related to software testing at large. Chapter 3 describes previously proposed code-based testing strategies and motivates the problem of error propagation. The problem is analyzed and our approach is outlined. Chapter 4 introduces and describes the notions of a program impact graph and an execution impact graph which provides the infrastructure needed for carrying out dynamic impact analysis. A discussion of related work in the area of program dependence graphs is also presented. Chapter 5 describes the framework of dynamic impact analysis. It defines the notion of impact strength and describes the algorithm to analyze impact relationships in an execution and compute impact strengths. A discussion of related work in the areas of execution analyses and error propagation is presented. Chapter 6 presents an overview of DIANA, a prototype implementation of dynamic impact analysis. It also discusses major pragmatic issues encountered during the implementation. Chapter 7 describes the experiments undertaken to validate the computation of impact strengths and presents the empirical results. The subject programs used in the experiments are described in Appendix A. An experience study was

undertaken to relate impact strengths to error propagation in the executions of faulty programs. The study involved analyzing thirty faulty programs. Chapter 8 describes the study and summarizes the results of the analyses. Details of the analyses are presented in Appendix B. Chapter 9 discusses potential applications of dynamic impact analysis to mutation testing, syntactic coverage-based testing and dynamic program slicing. Chapter 10 presents the conclusion and directions for further research. Two indices at the end of the dissertation provide ease of cross-referencing.

Chapter 2

Basic Terms & Concepts

This chapter explains some basic terms and concepts in the areas of program representation, debugging and software testing. The terms and concepts specific to our research are described later in chapters 4 and 5.

2.1 Program Execution State

From the programmer's point of view, at any point of time during execution, the program execution state consists of the following:

- Control state, which includes the next operation to be performed,
- Data state, which includes the contents of all storage locations and registers used by the program,
- Input state, the current position of the read heads on the input files (or streams)[1],
- Output state, the current position of the write heads on the output files *and* the contents of output files, and
- Resource usage state, the information about execution time, memory usage, etc.

[1] We will assume that there are files representing terminal I/O.

For convenience, *state variables* refer to the atomic units representing the state information described above and an execution state is said to be composed of instances of these state variables. The *execution history* of a program execution refers to the sequence of transitions from an initial execution state to a final execution state.

2.2 Fault Detection

A *test case* for a program is an input in the domain of the specification. A *test set* or *test suite* is a set of test cases. A *failure* occurs when a program computes an incorrect output for a test case [29]. A *programming error* or a *design error* is a mental mistake by a programmer or a designer, respectively.[2] A programming error or a design error may result in a textual problem with the code called a *fault* [29]. Without the information about the programming or design error that caused a fault, it is often difficult to associate a fault with specific text in the program. For example, consider the following faulty code that prints 1 to `N-1` instead of 1 to `N`.

```
for(i=1;i<N;i++) print(i);
```

The fault in the above code may be identified either as a problem with the loop exit condition (`i<N` instead of `i<=N`) or as a combination of two problems: one with loop initialization (`i=1` instead of `i=0`) and one with the argument to `print` (`i` instead of `i+1`). Thus, the perception of a fault depends on which correct version of the faulty code is considered or which programming error is presumed to cause the fault.

When a program fails on a test case, we say that at least one fault is present in the program. A fault is *detected* when it causes a failure. In what way does a fault lead to a failure? Several researchers, including Ostrand and Weyuker [56], Morell [45, 46, 47], Richardson and Thompson [67, 68], and Offutt [53], have independently investigated

[2] This is based on the definition of *error* in [29].

this problem. As mentioned in the introduction, we consider a model of fault detection which is based on similar models defined by Morell, Richardson and Thompson, and Offutt. According to this model, for a test case execution to detect a specific fault, it should meet three conditions:

- the *fault execution condition*, which causes the fault to be executed,
- the *error creation condition*, which causes the fault execution to generate an incorrect intermediate state, and
- the *error propagation condition*, which enables the errors in the incorrect intermediate state to propagate and cause an incorrect output.

The *errors* in the incorrect state refers to the values associated with the incorrect state variables. We acknowledge that the term *error* is already overloaded with several different meanings. For example, in numeric computations, it refers to the difference between the expected and the actual value of a quantity. As mentioned earlier, it also refers to a mental mistake by a programmer or a designer. Nevertheless, for the lack of a better alternative, we decided to use the term error to denote an incorrect value in an execution state.

The primary difficulty in applying this model to real faults is the problem of identifying the *first* incorrect intermediate state produced by a fault during an execution. Richardson and Thompson [67] assume the existence of *the* correct execution, and assess the incorrectness of a state with respect to the corresponding state in the correct execution. In reality, there may be several correct executions and a program state is considered correct or not depending on the definition of the set of corresponding correct or acceptable states. Alternatively, a state may be considered incorrect because the values of the state variables are inconsistent with respect to some implicit state invariants. Therefore, instead of requiring *the* correct execution, we assume that it is possible to identify whether a state is correct or incorrect. As we shall see in section 8.2.1, there

are certain kinds of faults for which this assumption does not hold. Nevertheless, given that this assumption holds for a large variety of faults, and for the lack of a better alternative, we decided to use the above model of fault detection.

2.3 Flow Graph Model of Program Structure

A program is often represented as a control flow graph for the purpose of simplifying various analyses of the program structure. A *control flow graph* is a directed graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. A *node* v_i represents either a single operation or a *basic-block*, where a basic-block is a single-entry single-exit sequence of code that is always executed together. An *edge* (v_i, v_j) represents a possible transfer of control from node v_i to node v_j. If $(v_i, v_j) \in E$, node v_i is a predecessor of node v_j and v_j is a successor of v_i. Without loss of generality, we assume that G has a unique node with no predecessor (the *source node*) and a unique node with no successor (the *sink node*). A *path* from v_i to v_j is a sequence of nodes starting with v_i and terminating in v_j. A test case for the program corresponds to a source to sink path in G which represents the execution sequence exercised by the test case. A *loop-free path* is one in which all nodes are distinct. A *simple path* is one in which all nodes, except the first and the last, are all distinct [65]. A *complete path* is one whose initial node is the source node and final node is the sink node.

Assuming that a variable x is bound to a memory location, an occurrence of x in which a value is stored in the memory location is termed a *definition of* x, and an occurrence of x in which a value is retrieved from the memory location is termed a *use of* x. When the association between variable name x and its memory location is voided (due to scope exit, language definition, etc.), it is termed an *undefinition of* x. Note that after an undefinition of x, the name x may very well be bound to another memory location in a different scope, but then it is a "different" x.

A *definition-clear path* with respect to a variable x is a path such the first node on the path possibly contains a definition of x, and any other node on the path does not define or undefine x. A definition of x at node v_i *reaches* a node v_j if there exists a path from v_i to v_j that is definition-clear with respect to x. In this case, if node v_j contains a use of x, there is said to be a *definition-use association* between nodes v_i and v_j with respect to x. A definition-use association is a special case of *definition-use chain*, defined as a sequence of nodes such that each node in the sequence contains a definition which reaches a use in the next node in the sequence, except for the last node which may or may not contain a definition.

2.4 Coincidental Correctness

The phenomenon of *coincidental correctness* is defined differently by various researchers. White, Cohen and Zeil [80, page 103] describe it as "when a specific test point follows an incorrect path, and yet the output variables coincidentally are the same as if the test point were to follow the correct path". According to Jeng[31], coincidental correctness occurs when a wrong function is executed, but due to some coincidence, the output is the same as if it were computed by the correct function. Richardson and Thompson [68, page 533] give a definition similar to that of Jeng: "Coincidental correctness occurs when no failure is detected even though a fault has been executed".

In this thesis, we are interested in the phenomenon of coincidental correctness as defined by White, Cohen and Zeil. To avoid ambiguity, we will refer to such a phenomenon by saying that the test execution or the result is *tolerant to errors in control paths*. The degree of such tolerance may vary depending on the program and the test case executed.

2.5 Test Oracle

All forms of software testing depend on the availability of an *oracle*. Given the behavior of a program during a test run, the oracle determines whether it is correct or not. The oracle may be a program specification, a table of examples, or simply the programmer's knowledge of how a program should operate. The observed behavior of the program may be available at various levels. Depending on the kinds of program behavior examined by oracles, Howden [28] classifies them as: *input-output oracles*, *trace oracles*, and *interface oracles*. An input-output oracle is capable of determining if the produced output is correct for that input. The availability of an oracle of this kind is a standard assumption in almost all of testing. A trace oracle is capable of determining if the produced trace of pre-specified events represents correct behavior or not. It is often enough to determine whether a pair of events on an object meet its interface specification or not. An interface oracle is used for this purpose.

2.6 Undecidability Results about Testing

There are two major undecidable problems that constrain the potential of software testing. Given two programs with infinite input domains, there is no algorithm to decide whether they compute the same function. Also, given a program with infinite input domain, there is no algorithm to decide whether there exists an input which causes a specific program component (e.g. statement, branch or path) to be executed. Several other undecidable problems in testing are directly or indirectly related to these two problems. If we assume that the input domains are finite, these problems can be solved by running the program(s) over all points in the input domain, which is computationally intractable.

2.7 Terms Related to Mutation Testing

Let $\mathcal{P}$ be the program under test. A single syntactic change to $\mathcal{P}$ is called a *mutation*. Typically, these mutations represent deletion or substitution of operators, variables, constants and statements in the program [13, 10]. When a mutation is applied to the program $\mathcal{P}$, the resulting program $\mathcal{P}'$ is called a *mutant*. A mutant $\mathcal{P}'$ is said to be *killed* by a test case if it produces a different output than the program $\mathcal{P}$ for the test case. An *equivalent mutant* is one that can not be killed by any test case in the domain of specification of the program.

2.8 Dynamic Program Slicing

Korel and Laski [35] introduced the notion of a *dynamic program slice* as "an executable part of a program whose behavior is identical to that of the original program with respect to a subset of variables of interest" during a specific execution. Agrawal and Horgan [2] proposed a simpler notion of a dynamic program slice. They define a dynamic program slice as "a collection of statements that affected the values of variables of interest" [70, page 108] during a specific execution. Computing a dynamic program slice using either of the above definitions requires processing of an execution history backwards and involves following the data and control dependencies from the operations that affect the values of specified variables. Venkatesh [70] uses the term *backward dynamic slice* to refer to both of the above notions of a dynamic slice.

Chapter 3

Problem Discussion

In this chapter, we discuss problems related to the effectiveness and computational feasibility of previously proposed code-based testing strategies. We first discuss what we mean by effectiveness and computational feasibility. Then we summarize the two main code-based testing approaches: the syntactic coverage-based approach and the fault-based approach. Testing strategies representing each of these approaches are briefly described and evaluated. Finally, we analyze the problem and motivate our solution approach.

3.1 Effectiveness

Weyuker, Weiss and Hamlet [79, page 4] describe effectiveness as follows: "Thus, the *effectiveness* of a criterion is the extent to which it enables us to uncover all of a program's failures. Note that effectiveness of a criterion is not linked to how many failures are *found*, but rather, to how many *remain*. Otherwise, the effectiveness of a criterion would depend on how buggy the program was in the first place." It is clear from this description that the effectiveness of a testing strategy is determined by its ability to detect faults.

3.2 Computational Feasibility

The cost of using a testing strategy should ideally take into account the computational resources and human effort consumed in (1) selecting test cases, (2) executing test cases, (3) validating the output for each of the test cases, and (4) carrying out the adequacy analysis. A major difficulty arises while evaluating the cost of a testing strategy because, in most strategies, human testers are involved in test case selection, validation of the output and in resolving the undecidable problems associated with the adequacy analysis. There does not exist yet an objective measure for quantifying this human effort. Therefore, we will use *only* the computational resources used by a testing strategy as a yardstick for measuring the associated cost. The computational resources consumed in the last three tasks directly depend on the size of the test suite used. Several studies [52, 77, 78] use only the size of test suite as a measure of the cost involved in using a test adequacy criterion. Weiss [75] argues that it is incorrect to assume that each test case consumes the same amount of resources since the test cases may differ in the amount of execution time and the sizes of input and output. Taking this argument into account, we define the computational resources used by a criterion as the processor and memory resources consumed for the execution and analysis of the test cases used by the tester in satisfying the criterion. We will use the term *computational feasibility* to refer to the feasibility with respect to the usage of the above mentioned computational resources.

3.3 Code-based Testing Approaches

Most of the research on code-based testing has concentrated on two main approaches: the syntactic coverage-based approach and the fault-based approach. For each of these approaches, we briefly describe the major testing strategies representing the approach and evaluate the approach with respect to effectiveness and computational feasibility.

3.3.1 The Syntactic Coverage-based Approach

Syntactic coverage based strategies make use of the observation that if a certain *syntactic component* (e.g. a statement, a branch, a definition-use association) in a program is not executed even once, it may contain a potential fault. A program is not considered adequately tested until certain syntactic components within the code are exercised ('covered'). Thus, a coverage-based test adequacy criterion specifies the syntactic components in the program that need to be exercised for adequate testing. Typically, a testing tool monitors the execution of each test case and keeps track of the syntactic components exercised by the test case and gives feedback to the tester about the components yet unexercised. The tester continues to provide more test cases until he/she has exercised all components specified by the criterion or has determined that the remaining components are not feasible.

Most of the *syntactic components* required to be exercised by common coverage-based strategies are derived from either the control flow characteristics of the program or the dataflow characteristics of the program or a combination of the two. Controlflow-based strategies are relatively simple to understand. The *statement coverage* criterion requires that every statement in the program be executed at least once during some test case execution. This idea is further extended in the *branch coverage* criterion, which requires that every branch in the code be taken at least once during some test case execution. Similarly, the *all-paths* criterion requires that every possible path in the program be executed at least once by some test case. However, since a program with loops can potentially contain an infinite number of paths, the all-paths criterion is not practical. Given this practical limitation, several coverage-based testing strategies propose path selection criteria to limit the number of paths or subpaths to be exercised.

Dataflow-based testing strategies [64, 17, 39, 51, 69] define path selection criteria

derived primarily from the dataflow characteristics of the program. The syntactic relationships to be exercised in this approach are either *definition-use chains* or *simple paths* connecting a variable definition to its use. For example, Rapps and Weyuker [64] propose the *All-Uses* criterion which requires that every definition-use association be exercised at least once. Ural and Yang [69] propose the *All simple OI-paths* criterion which requires exercising simple paths from input variables to the outputs influenced by the input. The rationale is that the association between an input variable and the output variable that is influenced by this input variable is critical and must be examined during testing.

Coverage-based testing has intuitive appeal for the tester. For adequately testing a module, it seems necessary that each of its important syntactic components (such as statements, branches or definition-use associations) be exercised. Another advantage of coverage-based testing is its low cost. Empirical studies [77, 78, 7] demonstrate that even for the most demanding of the Rapps and Weyuker family of dataflow coverage criteria [64], the average number of test cases required is fewer than the number of decision statements in the module.

One of the major drawbacks of coverage-based testing criteria is the lack of direct evidence concerning their fault detection ability. The requirement of exercising a syntactic component is satisfied by executing the component during an arbitrary test case execution. The computation accompanying the execution of that syntactic component and its impact on the overall program behavior are not considered.

A coverage-based test adequacy criterion relies on static program analysis to generate the syntactic components that it requires a test set to exercise. Some of these statically determined syntactic components in the program may not be executable. Determining whether or not there exists an input which will exercise a given syntactic component or relationship in the program is sometimes difficult or impossible. A closely related

problem is that of selecting an input that will cause a specific program feature to be executed. In general, both of these problems are undecidable, and are typically left for the human tester to solve. These problems are common to all code-based testing strategies (including the fault-based testing strategies described in the next section) and adversely affect the usability of these strategies. The research reported in this thesis does not address these problems.

3.3.2 The Fault-based Approach

Fault-based testing strategies concentrate on detecting specific faults or classes of faults. On the one hand, some fault-based testing strategies offer a guarantee regarding the absence of specific faults, provided certain unrealistic assumptions are satisfied. Such strategies usually have very high cost associated with them. For the purpose of this discussion we refer to them as *strong fault-based strategies*. On the other hand, in order to avoid the high cost, some fault-based strategies give up the guarantee regarding the absence of specific faults. We will refer to these strategies as *weak fault-based strategies*. Below, we discuss some of the strong and weak fault-based strategies.

Strong fault-based strategies

In this section, we describe three examples of strong fault-based strategies: mutation analysis [13, 10], symbolic testing [47], and Relay testing [67].

Mutation analysis is based on the *competent programmer hypothesis* – the assumption that the program, if incorrect, differs from a correct program by at most a few minor faults[12]. The first step in mutation analysis is the construction of a collection of *mutants* (§2.7) of the program. Given a test set, each test case is run on the *live mutants*, those that are not killed by any of the previous test cases. At the end, a mutant remains alive for one of the two reasons: (1) the test data is inadequate or (2) the

mutant is *equivalent* to the original program. The adequacy of a test set is measured by a mutation score representing the ratio of number of mutants killed to the number of non-equivalent mutants. Determining whether a mutant is equivalent to the original program is in general an undecidable problem and theoretically limits the usability of mutation testing. In practice, human testers are frequently able to detect equivalent mutants. However, when a large number of mutants are involved, this may be very time consuming and therefore expensive. Another key assumption involved in mutation analysis is the *coupling effect hypothesis* which "states that a test set that distinguishes all non-equivalent mutants with simple faults is so sensitive that it will also distinguish mutants with more complex faults"[54, page 132]. This assumption is needed for the following reason. When a mutant is killed by a test suite, it means that the test suite is good at detecting the isolated simple fault represented by the killed mutant. However, if the fault were present as a part of a complex fault involving several syntactic changes to the program, it may or may not be detected by the test suite. Offut [54] provides experimental evidence suggesting that a test set good at killing a set of mutants containing single mutations is also good at detecting mutants containing double mutations. No such empirical evidence has been reported for faults not represented by single or double mutations.

Symbolic testing [46, 47] is an application of Morell's theory on fault-based testing and is based on symbolic execution [11]. Symbolic execution's ability to model a class of executions by a single symbolic execution is generalized by providing the ability to simultaneously execute infinitely many alternate programs. This is achieved by representing infinitely many alternatives by a single *symbolic alternative*. For example, in the expression $x * y + 3$, 3 could be replaced by a symbolic alternative, say F, giving $x * y + F$. This transformed expression represents infinitely many alternatives of the original expression – one for each constant that can replace F. Typically, a symbolic

alternative represents a class of potential faults ("incorrect constant" in this example). The original program and the symbolic alternative are symbolically executed on a test case, and their output expressions are compared. The resulting equation describes those alternatives that are not distinguished from the original program by this test case. The symbolic testing techniques described above can be applied to demonstrate the absence of compound faults, however, at prohibitive cost. Determining the symbolic alternatives representing a fault class is non-trivial and has not been adequately addressed.

Richardson and Thompson [67] defined the RELAY model of the process by which a potential fault in the program results in a failure. Based on this model, they develop *revealing conditions* that are necessary and sufficient to guarantee detection for a pre-specified class of faults. That is, any test case that satisfies these conditions would be successful in detecting a fault from the specified class, if present. These conditions are computed using symbolic execution. The RELAY model provides a framework not only for determining what faults have been eliminated, but also for providing the necessary and sufficient revealing conditions to help test data selection.

The strong fault-based testing strategies typically have better understood fault detection capabilities than coverage-based testing strategies. Each of the above described strategies rigorously pursue the goal of meeting the fault execution, error creation and error propagation conditions (§2.2) for specific faults. After executing a test set, a strong fault-based testing strategy can give a guarantee regarding the absence of prespecified faults under the unrealistic assumption that a prespecified fault does not interact with other faults in the program. Although specific faults are eliminated, nothing can be said about the classes of faults not modeled by a given fault-based strategy. Moreover, there is no evidence that all fault classes can be described by a suitable fault classification scheme such that their absence can be guaranteed by the fault-based approach.

Another major problem with a strong fault-based strategy is its high cost. For

example, the analysis of a 29 line program using a mutation testing tool (Mothra [12]) generated 1067 mutants, and it took 52 test cases and 22,192 executions to reach a mutation score of 92%. The applicability of other prominent fault-based strategies such as symbolic testing and RELAY testing depend heavily on the feasibility of symbolic execution which has a very high cost even for small programs.

WEAK FAULT-BASED STRATEGIES

Hamlet [23] originally proposed the idea of testing a program expression by distinguishing it from a set of alternate expressions. In his approach[1], several alternate expressions are considered for each of the designated expressions. When a designated expression is encountered during a test case execution, its alternatives are evaluated and compared with the value of the original expression. Those that evaluate differently from the original are marked. After all the test data has been executed, the unmarked alternate expressions are printed by the compiler as warning messages, indicating that either the test data are inadequate, or equivalent simpler expressions have been found. The idea of *testing a program component by distinguishing it from its alternatives* is an important characteristic of all fault-based testing strategies.

In order to avoid the complexity of mutation testing arising out of independent executions of a large number of mutants, Howden [27] proposed weak mutation testing which simplifies the requirements for killing a mutant. Specifically, weak mutation testing requires that the code containing the mutation be executed at least once and that at least one such execution yield a different execution state immediately following the execution than that produced by the unaltered code.

Zeil proposed *perturbation testing* [81] and the EQUATE testing strategy [82] which are similar to Hamlet's compiler-based testing and Howden's weak mutation testing

[1] Most of our description of Hamlet's approach has been adapted from Morell's description in [46].

described above. These strategies proposed by Zeil use different approaches for determining the alternative expressions. For example, in perturbation testing, the perturbing functions used to arrive at alternate expressions have some sort of global dependence on the code since it uses information about the variable names used and types of calculations performed in the entire module.

Each of these strategies rigorously pursue the goal of meeting the fault execution and error creation conditions for specific faults, and rely on the *weak mutation hypothesis* for meeting the error propagation condition (§2.2). The weak mutation hypothesis states that a test case that satisfies the fault execution and error creation conditions with respect to a specific fault will also satisfy the corresponding error propagation condition [42]. This assumption reduces the cost, but gives up the guarantee regarding the absence of specific faults.

3.4 Problem Analysis

Based on the above evaluation of code-based testing strategies, we make the following observations.

- The notion of *exercising* a syntactic component in the coverage-based approach is weak in that it only requires an arbitrary execution of the syntactic component. In order to detect potential faults associated with a syntactic component, it is important that the corresponding error creation and error propagation conditions are satisfied as well.
- In their attempt to guarantee the absence of specific faults under certain assumptions, strong fault-based strategies require that the fault execution, error creation and error propagation conditions be satisfied for the specified faults. However, these strategies are computationally very expensive. In comparison, the weak

fault-based strategies do not enforce the error propagation condition and their computational costs are considerably lower than their strong counterparts.

The first observation suggests the importance of satisfying the error propagation condition, and the second observation implies that the cost for guaranteeing the satisfaction of the error propagation condition may be inherently very high. This motivated us to examine the problem of error propagation and to design a cost-effective technique to estimate the likelihood of error propagation from various potential sources of errors in a program execution.

In order to better understand the difficulties involved in satisfying the error propagation condition, we first have to understand the reasons for possible failures of error propagation. An error in a state variable instance may fail to propagate to the output of the execution if one of the following holds:

- no computation using the erroneous state variable instance affects the output, or
- there is at least one computation that uses the erroneous state variable instance and affects the output, however, in all such computations, the error is eventually masked out.

In the first case, we say that the erroneous state does not impact the output or that the erroneous state has *no impact*. For example, consider a tax computation program that incorrectly loads the data about itemized deductions. However, in a specific test scenario, the tester chooses the standard deduction method. Therefore, the incorrect state variables associated with itemized deductions do not impact the output.

The second case is more interesting, and we discuss it in detail in the following paragraphs. For simplicity, consider the computation represented by a binary function $f(w, y)$ that masks a specific error in w. Let w' denote the corresponding erroneous value of w. The function f is said to mask the error in w if $f(w', y) = f(w, y)$. Alternatively, it may be the case that y also has an error. Let y' denote the erroneous value of y.

Then, the errors in w and y are masked if $f(w', y') = f(w, y)$.

It is possible that individually, the errors in w and y are not masked, but they are masked when present together. That is, $f(w', y) \neq f(w, y)$, $f(w, y') \neq f(w, y)$, but $f(w', y') = f(w, y)$. In such cases, we say that w' and y' are *canceling errors*. For example, in $w * y$, if the values of w and y both have incorrect signs, the result will have the correct sign. Similarly, in $w + y$, if the signs of the errors in w and y are different but with same magnitudes, the result will still be correct.

An error masking function is essentially a many-to-one function. Voas, Miller and Payne [73, 71, 74] introduced the notions of "internal state collapse" and "implicit information loss" while referring to the behavior of many-to-one functions and attempted to quantify these notions by introducing a metric called *domain/range ratio* (DRR) [73]. "The *domain/range ratio* (DRR) of a specification is the ratio between the cardinality of the domain of the specification to the cardinality of the range of the specification. A DRR is denoted by $\alpha : \beta$, where α is the cardinality of the domain and β is the cardinality of the range" [71, page 238]. A high DRR of a function indicates that the function is less likely to propagate an error and a low DRR indicates that the function will readily propagate an error. They acknowledge that the DRR of a function only "partially suggests" the likelihood of propagating errors to the result of that function and that it is not clear how to usefully extend the definition of DRR for a function with n-dimensional input space. In the following examples, we illustrate some of the limitations of the DRR metric while providing more insight into the phenomenon of error masking.

Example 1

Consider the function `string-match` that compares two strings and returns `true` or `false` depending on whether the two strings are equal or not. If α is the cardinality of the set of all strings, the DRR of the `string-match` function with respect to either

argument is $\alpha : 2$, which is clearly very high. This suggests that the likelihood of propagating an error in an argument would be very low in an implementation of the `string-match` function. This is certainly the case when the result of the match is `false` since it is less likely that an error in one or both of the argument strings will make them equal. However, when the result of the match is `true`, the function is extremely sensitive to errors in the input strings. This is so, since most errors in two equal strings will make them unequal. This example suggests that the error sensitivity of a many-to-one function may be different for different result values.

Example 2

Consider the multiplication operation $a * b$. Given infinite integer domains, the DRR for this operation with respect to a is 1. This indicates that an error in a will always propagate to the result of the operation. However, in a specific instance of the operation, when $b = 0$, an error in a does not propagate to the result of the operation. This example suggests that the error sensitivity of a many-to-one function with respect to a specific argument may depend on the values of other arguments.

Example 3

Consider the relational expression $a > b$ with the result `true`. Given that a is any integer, since the cardinality of the range of the expression is 2, it is clear that the DRR of this operation with respect to a is very high. Hence one should expect very poor propagation for errors in a. Since the result is `true`, the positive errors in the value of a will not propagate to the result. However, depending on the magnitude $|a - b|$, the `true` result can be very sensitive to negative errors in the value of a. This example suggests that the error sensitivity of a many-to-one function may depend on the nature of errors under consideration.

Example 4

Consider an array reference $a[w]$ used as an operand in some operation. It is straightforward to see that the DRR of the indexing operation is 1 since every index refers to a different array element. Therefore, one would expect that an error in w will be readily propagated. However, even though each index corresponds to a different array element, several array elements can have the same value. In such cases, the likelihood of error propagation could be really small. This example illustrates the possibility of failure of error propagation in referencing operations such as array indexing or pointer dereferencing. Such operations are largely ignored in the literature discussing error propagation [42, 46, 53, 67, 68, 71, 72].

Thus, in a specific execution of a function, the propagation of an error in an argument w to the result r may vary with the types of operations involved in the computation, the result value r, the values of other arguments, and the type of the error. From these observations, we argue that in order to better understand the error propagation behavior in a computation, we ought to consider a *dynamic* metric rather than a *static* metric such as the DRR. Such a dynamic metric should represent, in a specific execution, the likelihood of propagation of errors from the source of error (w) in the computation to the result of the computation (r). In other words, we want to measure the *error sensitivity* of the result r with respect to the source of error w. Generalizing this, in an execution, we want to measure the error sensitivities of the output of the execution to several potential sources of errors in the execution. In order to be useful, such a measurement should be cost-effective. The following paragraphs present the high level issues associated with measuring error sensitivity and introduce our approach.

Consider a specific execution $\mathcal{T}$ of a program. If W represents the set of potential sources of errors of interest in $\mathcal{T}$, and r represents the output of $\mathcal{T}$, we want to measure the error sensitivity of the output r with respect to each $w \in W$. We refer to this measure

as the strength of w's impact on r, or simply the *impact strength* of w. Let $E(w)$ denote the set of plausible errors for w with respect to which we want to measure the impact strength of w. A simple approach for computing the impact strength of w could be as follows: for every w' in $E(w)$, run an alternate execution $\mathcal{T}'$ obtained by substituting w' for w in $\mathcal{T}$, and examine whether the error propagates to the output. This would yield a frequency estimate of the desired measure of error sensitivity. There are four potential problems in this simple approach. First, it is not clear how to determine $E(w)$, the set of errors for w. Second, in an alternate execution, the error may take an incorrect path and possibly take much longer than the the original execution or may never terminate. Third, $E(w)$ could be a very large set and it may be prohibitively expensive to run alternate executions for each member of $E(w)$. Fourth, since we want to measure the impact strengths of a large number of potential sources of errors in an execution, this adds a new dimension to the cost of the analysis.

The framework of dynamic impact analysis presented in this thesis attempts to address these problems. The notions of *program impact* and *impact paths* are defined to capture the different ways in which a potential source of error can affect the output. To avoid the high cost associated with executing alternate control paths, we analyze only the control path of the execution and define heuristics to estimate the impact due to avoiding alternate paths. Using this and other cost-saving approximations, an algorithm is designed to compute the desired impact strengths. We then conduct experiments to validate the computation of impact strengths and to understand the consequences of the various approximations.

Chapter 4

Impact Graphs

Our notion of an impact graph has been strongly influenced by the need to understand the roles played by various syntactic entities during program execution and the kinds of impact they have on the program behavior during the execution. As we shall see in the following chapter, the notion of impact defined here forms the basis for measuring the quality of impact of various program entities on the output of an execution.

We use C as the reference programming language for describing the impact graphs and related concepts. We first define the notion of a *program impact graph* which attempts to capture the *static* impact relationships among various program entities. Then we define the notion of an *execution impact graph* which attempts to capture the *dynamic* impact relationships among the instances of various program entities during execution. This is followed by a description of several terms and notations related to impact graphs. Finally, we compare the notion of a program impact graph with the notions of program dependence graphs used in the literature.

4.1 Basic Terms & Notations

In this section, we present some of the terms and notations used in describing the program impact graph and the execution impact graph.

A *decision predicate* is the expression used in a control flow construct, whose evaluation at runtime determines which branch is taken. Each potential branch that can be taken as a result of a decision predicate evaluation is called a *decision branch* of that predicate. A *decision arm* is the code segment contained in the textual scope of a decision branch. It is possible that the code segment corresponding to a decision arm is empty. The decision arms of a decision predicate are *siblings* of each other.

Unconditional control transfer constructs such as `goto`, `break`, `continue`, function call and `return` correspond to specific branches in an interprocedural control flow graph [59] of a program. Such a branch will be termed an *unconditional branch* to distinguish it from a decision branch. Consider the following C statement: `if (condition) goto Label1;`. There are three distinct branches here. The conditional predicate in the `if` statement corresponds to two decision branches: a true branch and a false branch. The `goto` statement corresponds to an unconditional branch.

In the C language, the `break` statement is used in two distinct contexts: for breaking out of a loop and for breaking out of the fall-through `case` statement within a `switch` statement. The former use of the `break` statement will be referred to as a *loop exit*.

Program Entities and Entity Instances

During impact analysis, we will be investigating the impact relationships among the following program components: variable definitions, variable uses, constant uses, decision predicates, decision arms, function definitions, operators and function calls. We will use the word *entity* to refer to a syntactic component listed above. As needed, we will add more program components to this list of entities.

During a program execution, an expression may be executed zero or more times and the various entities contained in the expression may also be "executed" zero or more times. We will use the term *entity instance* to refer to *an* "execution" of an entity. We illustrate this in the following example.

Program	Variable Def/Use Entities	Entity Instances
A = 1;	A_1	a_1
B = 2;	B_1	b_1
while (A < 5)	A_2	${}_1a_2,\ {}_2a_2,\ {}_3a_2$
A = A * B;	$A_3,\ A_4,\ B_2$	${}_1a_3,\ {}_1a_4,\ {}_1b_2,\ {}_2a_3,\ {}_2a_4,\ {}_2b_2$
C = A + 5;	$C,\ A_5$	$c,\ a_5$

As a convention, we will always use identifiers beginning with an upper case letter to specify entity names and identifiers beginning with a lower case letter to specify entity instances. As shown in the above example, the various entities corresponding to the same variable are distinguished using a subscript and the various entity instances corresponding to the same entity are distinguished using a prefix-subscript. For example, ${}_1a_4$ is the first instance of the entity A_4, ${}_2a_4$ is the second instance of A_4, and so on. When there is no ambiguity, the subscripts are omitted. For example, if P denotes the predicate entity (A < 5), the three instances of P would be denoted as ${}_1p$, ${}_2p$ and ${}_3p$.

Kinds of Operations

Each instruction in an intermediate code representation [5, pages 13-14] of a program will be referred to as an *operation*. We will assume that the time for executing an operation is bounded by a constant. The operations in an intermediate code representation

of a C program can be broadly classified into four categories: control transfer operations, data transfer operations, referencing operations and computation operations. The computation operations can be further subdivided into address and data computations.

The *control transfer* operations include the operations involving transfer of control due to language control constructs such as if, while, do while, switch, goto, break, continue, function call, return and conditional expression. The *data transfer* operations include various forms of assignment operations and implicit data transfer operation such as copying actuals to formals.

The *referencing* operations involve accessing a storage location for reading or writing given the location address. The location address is either implicitly known to the compiler or is explicitly computed at execution time using specific operations provided in the language. The *address computation* operations explicitly compute location addresses at execution time. For example, when a variable is referenced by name, the compiler implicitly knows its address. In contrast, the dereferencing operators (`*`, `->`) and the array indexing operator (`[ ]`) involve address computation at execution time.[1] Each of these operators involve two operations: an address computation operation and a referencing operation. A memory allocation routine such as `malloc()` could also be looked upon as an address computation operation.

All other operations are grouped together as *data computation* operations. They involve manipulation of various kinds of "data", including integer, floating point and pointer data. The C language supports pointer arithmetic, whereby a pointer is manipulated like other arithmetic data, with some restrictions. Thus pointer data can take part in data computation (e.g. pointer `p` in `p = p + 2`), and likewise, integer data can take part in an address computation operation (e.g. index `i` in `a[i]`). Therefore, we will treat both data and address computation operations as *computation* operations.

[1] The '.' operator in C may or may not result in address computation at execution time.

Note that a single statement might consist of several different categories of operations defined above. For example, consider the following statement:

```
c = cond ?  a[i] * b :  p->f ;
```

Assume that `a` has been declared as `int a[10]` and `p` is a pointer to a structure with `f` as a field. This statement includes one control transfer operation, two explicit address computation operations (`[]` and `->`), one data computation operation (`*`), one data transfer operation (`=`) and several referencing operations. Of course, not all of them get executed when this statement is executed.

Observable Program Behavior and Output

In order to determine whether a program execution meets its specification, the test oracle "observes" the initial state and a set of one or more partial execution states from the corresponding execution history. We will refer to such a set as the *observable program behavior*. Typically, for an input-output oracle, this set consists of only the output state at the end of the execution. However, there may be oracles which look at intermediate execution states as well. Often, resource usage is also examined by the oracle. Specifically, when the execution does not terminate, the oracle may examine the accumulated execution time to reach its verdict.

Out of the various components of an execution state described above, some components can be easily observed externally, while some are relatively difficult to observe. For example, the output state can be easily examined, while the complete data state at a specific point in the program is relatively difficult to examine. For the purpose of the discussion of impact graphs and dynamic impact analysis, we will assume that the *observable program behavior* consists of the following:

- the output state, and

- the *accumulated execution time*, which is the length of the time interval from the start of the execution to the time of the observation.

For convenience, we will often use the terms *output*, *program behavior* or *observable behavior* to mean observable program behavior.

4.2 Notion of Program Impact

Our goal is to define a notion of *program impact* that can be used to describe and quantify the kind of impact a program entity has on the observable program behavior in a specific execution. The notion of *program impact* can be looked upon as the inverse of the notion of *program dependence* as defined in the literature [16, 2, 63]. When a program entity B depends on entity A, we can say that A impacts B. However, as we shall show in section 4.6, the various definitions of program dependence graphs proposed in the literature were not suitable for use in dynamic impact analysis. Therefore, we developed the notion of a program impact graph based on the operand/operator level program dependence graph suggested by Ferrante, Ottenstein and Warren [16]. In comparison to such a program dependence graph, the program impact graph represents detailed dataflow and new kinds of dependencies. The most distinguishing feature of the program impact graph is the notion of *potential control impact* which is motivated below.

In order to get an intuitive understanding of the potential control impact, consider the following question.

> How does a decision arm (the code segment corresponding to a decision branch) impact the observable program behavior?

A decision arm can impact the observable program behavior in one or more of the following ways:

1. by executing an observable state-transition event e.g. writing output onto a file.

2. by modifying the data state (e.g. defining a variable), which then impacts the observable program behavior.

3. by modifying the control state (e.g. executing another control transfer operation), which then impacts the observable program behavior.

4. by avoiding one or more of the above which could have impacted the observable program behavior; e.g. avoiding output, avoiding a variable definition, avoiding indefinite iteration, and avoiding indefinite recursion.

The first item represents *direct impact* on entities that contribute to the observable program behavior. For example, arguments of output statements determine the output produced. In the next two items, the impact of the decision arm on the observable program behavior is indirectly expressed in terms of the impact of the entities that it directly impacts. This implies that the notion of program impact is *transitive*. The last item represents *potential control impact*. As an example of such behavior, consider the following code segment:

$$\begin{gathered}\text{Def1}(v);\\ \text{if } (p)\ \text{Def2}(v);\\ \text{Use1}(v);\end{gathered}$$

Now consider an execution of the above code in which the `false` branch of the conditional statement is taken. What role does the conditional statement play in such an execution? It avoids a state change which would have killed Def1(v). At a later point, if Use1(v) gives an incorrect value of v, it could be either because Def1(v) was incorrect or because the predicate p was incorrect. Thus, it is clear that an incorrect value of p could cause an incorrect value of v at Use1(v) and hence p should be included in the

backward dynamic slice (§2.8) with respect to v. However, the backward dynamic slice [2, 35, 70] computed by following the conventional data and control dependencies among the executed statements will not contain the predicate p. In the context of fault localization, Korel and Laski [36] introduced the notion of *potential influence* in an attempt to consider the potential control impact of predicate p on the Use1(v) (see section 4.6 for a discussion). We generalize this notion to include the potential impact of p on the output state by avoiding an output statement or on the control state by avoiding a loop exit or a function return. In the program impact graph, we would like to explicitly represent such potential control impact. In summary, a decision arm can impact the observable program behavior either by causing a state modification or by avoiding a state modification.

From the above discussion, the following requirements of program impact graph become apparent.

> The program impact graph should capture the direct and potential impact relationships that exist among the program entities and it should support computation of transitive impact relationships.

The following section describes the notion of a program impact graph designed to meet these requirements.

4.3 Program Impact Graph

A program impact graph is a directed graph with nodes representing the program entities and the arcs representing direct impact relationships among the entities. The potential impact relationships, where necessary, are represented as attributes on the nodes. In order to capture the subtle differences among various kinds of impacts, we use additional attributes. For example, each node has a *node kind* and each arc has an *impact kind*.

Throughout our description of the program impact graph, we will use the following example to illustrate its features. Consider the following conditional expression statement in a C program:

```
c = cond ?  a[i] * b :  p->f ;
```

Assume that `a` has been declared as `int a[10]` and `p` is a pointer to a structure with `f` as a field. The basic entities involved in the above expression and their direct impact relationships are shown in Figure 4.1. The nodes in the figure have been numbered for ease of reference. While referring to the figure, the node numbered i will be referred to as n_i and an arc from node i to node j will be referred to as $\langle n_i, n_j \rangle$.

Nodes of an Impact Graph

The different kinds of nodes that constitute an impact graph are described below.

- *variable definition* node, represents an assignment to a variable. For example, n_{33} represents an assignment to the variable `c`. The variable definition nodes n_1 through n_6 are not a part of the conditional expression. They represent variable definitions that occur elsewhere in the program.
- *variable use* node, represents a use of the value stored in a variable. For example, n_{18} is a use of an element of the array `a` and n_{20} is a use of the variable `b`.
- *constant use* node, represents a use of a constant value. e.g. n_{12}, n_{26}.
- *temporary definition* and *temporary use* nodes, represent the definition and immediate use of a temporary variable. The implicit assignment and use of a temporary is treated like an assignment and use of a regular variable. Usually, temporary definition and use nodes occur in pairs. Hence for pictorial representation, we represent the pair as a single node. Examples include n_{16}, n_{21}, and n_{27}. However,

Figure 4.1: Fragment of a Program Impact Graph

when a temporary variable is conditionally defined, as in the conditional expression of C, we separately represent the definition and use nodes of a temporary variable as illustrated by n_{22}, n_{29}, and n_{32}.

- *operator* node, represents the operation being performed. This includes both built-in operators and function calls. For example, n_{13} and n_{19} represent the indexing and multiplication operators respectively.
- *decision predicate* node, represents the result after evaluating a decision predicate. For example, n_9 represents the result of the predicate in the conditional expression.
- *code segment* node, represents a source code segment of importance. For example, in a decision arm is represented as a code segment node (e.g. n_{10}, n_{11}) and so is a function definition. To aid visualization, the diagrammatic representation of a program impact graph shows a code segment node corresponding to the code segment C as encapsulating the nodes corresponding to the syntactic entities within C. The nodes encapsulated by a code segment node N are said to be the *members* of N. It is possible for a code segment node to be a member of other code segment nodes. A node which is a member of code segment node N but not a member of any member of N is called an *immediate member* of N. In other words, a node is an immediate member of the innermost encapsulating code segment node. For example, nodes n_{23} through n_{29} are immediate members of the code segment node n_{11}.

Representing Specific Program Entities as Nodes

The fragment of program impact graph in Figure 4.1 does not contain examples of all possible program entities that can be modeled using the kinds of nodes described above. Below we discuss some interesting situations.

With each *function definition* we associate a code segment node, and if the function

returns a value, we associate a unique *function result* entity represented by a temporary definition node. The *return expression* entity in a return statement is associated with a temporary use node or a variable use node, as appropriate.

A function call involves the use of a *function reference*, which is either a constant use node or a variable use node depending on whether a function name is used or a function pointer is used. The actual parameters involved in the call are either variable use nodes or temporary use nodes. The formal parameters of a function definition are variable definition nodes. Also, with each function call, when its return value is used in some expression, there is a temporary definition node and a temporary use node corresponding to the definition of *function result* and the use of the *function return value*.

Library functions with a variable number of arguments are also supported in this framework. We will defer their discussion until chapter 6.

Focusing mainly on the program entities that influence dynamic impact behavior, we do not currently model the impact due to syntactic entities such as structure definitions, type definitions, and C preprocessor constructs.

Arcs of an Impact Graph

The nodes in the impact graph are connected by directed arcs which reflect the direct impact relationships among the entities represented by the nodes. We distinguish among four different kinds of impacts: data impact, reference impact, operator impact and control impact. Each of these is separately discussed below. (Note that the references to impact arcs of the kind $\langle n_i, n_j \rangle$ refer to the arc from node n_i to node n_i in Figure 4.1).

Data Impact

Data impact arcs capture the following impact relationships.

- *operand-result* — connect each of the operands of a computation operation to the node corresponding to the result of that operation. For example, $\langle n_{14}, n_{16}\rangle$ and $\langle n_{20}, n_{21}\rangle$ are operand-result arcs.
- *data transfer* — connect the source to the destination in a data transfer operation. This includes the implicit assignment operations to temporary variables without any explicit source level assignment operator. For example, $\langle n_{21}, n_{22}\rangle$ and $\langle n_{22}, n_{32}\rangle$ represents data transfer operation among temporary variables. These also represent the data transfer from the actual parameters of a function call to the formal parameters of the function definition or from the function result to the function return value.
- *def-use* — connect a variable definition node to its use nodes. For example, $\langle n_3, n_{20}\rangle$ and $\langle n_5, n_{24}\rangle$ are def-use arcs.

A data impact arc from node A to node B signifies that in an execution, the data value of an instance of B may directly depend on the data value of an instance of A. Note that the first two items above represent impacts that are local within an operation while the last item represents *def-use arcs* which represent interactions among various computation operations within the program. These def-use arcs represent both intra-procedural and inter-procedural definition-use associations.

Reference Impact

Reference impact arcs connect a node representing a constant or computed location address value to the node representing the referenced entity. For example, the arc $\langle n_{15}, n_{14}\rangle$ models reference impact due to the node n_{15} representing a constant address and the arc $\langle n_{16}, n_{18}\rangle$ models reference impact due to the node n_{16} representing the address computed by the indexing operation.

Operator Impact

Operator impact arcs connect an operator to the result of an operation involving that operator. For example, $\langle n_{19}, n_{21} \rangle$ and $\langle n_{13}, n_{16} \rangle$ are operator impact arcs.

Control Impact

Control impact is more difficult to express than data impact, reference impact or operator impact. Recall from the discussion in section 4.2 that a decision arm can have direct or potential control impact on other program entities. We use control impact arcs to represent the direct control impact relationships and attributes to represent the potential control impact relationships.

Control impact arcs represent the following direct impact relationships:

- *decision branch* — connect a decision predicate node to each of the code segment nodes corresponding to the decision arms of the decision predicate. For example, $\langle n_9, n_{10} \rangle$ and $\langle n_9, n_{11} \rangle$ represent decision branches.

- *implicit control* — implicitly connect a code segment node to the nodes among its immediate members that represent either data state modification or control state modification. Examples of such nodes are: function reference node, variable definition node, and decision predicate node.

  Additionally, there are cases for which a decision arm computes a value and makes it available outside the scope of the decision arm. For example, such is the case in a conditional expression or a short circuit boolean operation. In such cases, the code segment node corresponding to a decision arm is connected to the temporary definition node representing the value computed in the decision arm. For example, each of $\langle n_{10}, n_{22} \rangle$ and $\langle n_{11}, n_{29} \rangle$ represents the control impact of a decision arm on the computed value.

- *function call* — connect a function reference node to the code segment node corresponding to the called function's definition.

In case of decision branches and implicit control, a control impact arc from node A to node B signifies that in an execution, B is instantiated only if A is instantiated. However, this is not true in the case of a function call, since the function definition may be instantiated by several call sites.

Potential Control Impact

Currently, the impact graph attempts to approximate three kinds of potential impact relationships: avoiding modification of the output state, avoiding modification of the data state (i.e. avoiding a variable definition), and avoiding indefinite iteration or recursion.

Each code segment node supports the following attributes:

- *controls-exit* – a boolean flag that is true if a loop exit or a return statement will *always* be executed when this node is executed and is false otherwise.
- *controls-output* – a boolean flag that is true if an output statement will *always* be executed when this node is executed and is false otherwise.
- *list-of-defined-variables* – a list of the variables that will *always* be defined when this node is executed.

The first two attributes are easy to compute using simple control flow analysis. The last attribute requires complex dataflow analysis. In fact, in the presence of arrays and pointers, it is not possible to get a precise value for this attribute [59, 60]. Also, it is possible that all of the variable definitions within a code segment are conditional, and therefore may or may not be executed even if the code segment is executed. These problems cause approximations in the values of this and other derived attributes.

Each decision predicate node also supports the above three attributes, with appropriately modified definitions. A decision predicate node has the controls-exit/controls-output attribute true if *all* of its decision arms has the corresponding attribute true. Similarly, the list-of-defined-variables for a decision predicate node is the intersection of the corresponding lists from *all* of its decision arms.

Each code segment node representing a decision arm additionally supports the following attributes:

- *potentially-controls-exit* – a boolean flag that is true when *all* of the sibling decision arms have the controls-exit attribute true and is false otherwise.
- *potentially-controls-output* – a boolean flag that is true when *all* of the sibling decision arms have the controls-output attribute true and is false otherwise.
- *list-of-potentially-impacted-variables* – a list of the variables that will *always* be defined by *each* of the sibling decision arms. That is, it is a list of the variables whose redefinitions are definitely avoided by executing this decision arm.

These attributes attempt to capture the potential impact relationships mentioned above for various commonly occurring situations. Not all situations can always be represented using only these attributes. For example, in the `switch` statement of C, it is possible to have an *empty* decision arm for which all the above attributes are either false or empty lists. In that case, no impact is indicated for the decision arm. More research is necessary to investigate the impact relationships of code segments which are not covered by the above attributes.

Current limitations

Currently, we do not model the impact due to the `goto` statement and the 'fall-through' semantics of the `case` statements within a `switch` statement. The implementation described in Chapter 6 permits these constructs to appear in a program, but does not attach any impact relationships to them. Also, as mentioned above, we cannot always

represent the impact due to a code segment and approximations are introduced due to inherent inaccuracy in dataflow analysis.

4.4 Execution Impact Graph

As we saw in the previous section, the nodes of a program impact graph correspond to program entities and the arcs correspond to the (static) direct impact relationships among these entities. An *execution impact graph* for a specific program execution is similar to a program impact graph, except that the nodes in the execution impact graph represent the entity instances that actually occurred during the execution and the arcs represent the direct impact relationships that were demonstrated during the execution. For example, Figure 4.2 shows a fragment of the execution impact graph corresponding to the program impact graph fragment of Figure 4.1.

Observations

- An execution impact graph is always a directed acyclic graph. That is, an entity instance cannot impact itself directly or indirectly.
- A decision node in the execution impact graph has exactly one control impact arc emerging from it.
- The arcs correspond to impact relationships that were actually demonstrated during the execution. On the other hand, arcs in a program impact graph represent possible impact relationships.
- Every arc in the execution impact graph has a corresponding arc in the program impact graph. More than one arc in the execution impact graph may correspond to the same arc in the program impact graph.
- As we will see in the next section, every node (entity instance) in the execution impact graph has a value associated with it.

Figure 4.2: Fragment of an Execution Impact Graph

4.5 More Terms & Notation

In this section, we describe some more terms and notation related to impact graphs. These terms will be used in describing the framework for dynamic impact analysis in chapter 5. A cursory reading of this section should help the reader in understanding the aspects of an impact graph that are important in dynamic impact analysis. To avoid the need for frequent cross-referencing, we will give short explanations accompanying future references to these terms.

$\mathcal{P}$ denotes a program or its program impact graph.

$, (\mathcal{P})$ the set of all possible executions $\{\mathcal{T}_1, \mathcal{T}_2, \cdots\}$ of the program $\mathcal{P}$.

$\mathcal{T}$ denotes a specific execution or its execution impact graph.

X, Y upper case letters denote entities within the program $\mathcal{P}$.

x, y lower case letters denote entity instances in the execution $\mathcal{T}$.

$E(X, \mathcal{T})$ or $\{{}_1x,\ {}_2x, \cdots\}$ denotes the set of all instances of entity X in the execution $\mathcal{T}$.

$E^{-1}(x)$ denotes the entity corresponding to the entity instance x.

Entity Instance Value

$V(x)$ denotes the value of the entity instance x. Most entity instances have values associated with them naturally. For example, a variable use instance has the value of the variable retrieved from the memory, a decision predicate instance has the value which is used to determine what branch to take, and so on. However, some entity instances such as a code segment instance usually do not have any natural values. To ensure uniform treatment of all entity instances, we define the following values.

μ is the value used to indicate the existence of entity instances that do not naturally have an associated value.

ϕ is the value used to indicate a hypothetical absence of an entity instance.

$W(X)$ denotes the set of valid values allowed by the entity X (using the broader notion of *value* defined above).

Impact Successor and Predecessor Sets

Let y be an entity instance in an execution impact graph $\mathcal{T}$.

$I_{pred}(y)$ denotes the *impact predecessor set* of y, and is defined as the set of all entity instances x such that there is an impact arc $\langle x, y \rangle$ in $\mathcal{T}$.

$I_{succ}(y)$ denotes the *impact successor set* of y and is defined as the set of all entity instances z such that there is an impact arc $\langle y, z \rangle$ in $\mathcal{T}$.

The above two relations are similarly defined for an entity Y in a program impact graph $\mathcal{P}$.

$I_{pred}(Y)$ denotes the impact predecessor set of Y, and is defined as the set of all entities X such that there is an impact arc $\langle X, Y \rangle$ in $\mathcal{P}$.

$I_{succ}(Y)$ denotes the impact successor set of Y and is defined as the set of all entities Z such that there is an impact arc $\langle Y, Z \rangle$ in $\mathcal{P}$.

Impact Paths

Recall that a simple path in a directed graph is one in which all nodes, except possibly the first and last, are distinct [17]. An *impact path* is a simple path in an impact graph. Recall that an execution impact graph is a directed acyclic graph, hence all of its paths are simple.

The notion of an impact path should not be confused with the notion of a control path in a control flow graph.

Let $Paths(i, j)$ denote the set of impact paths from node i to node j in an impact graph.

Impact kind attribute

As we described earlier, with each arc in an impact graph, we associate an *impact kind* attribute which takes one of the four values: *control*, *data*, *reference* and *operator*. These are the four *basic impact kinds* in our framework. The impact kind of an impact path is defined in terms of the impact kinds of its constituent impact arcs as shown below. In the following, we use 1^+ to mean 1 or more and 0^+ to mean 0 or more.

Impact-kind of an impact path	Impact-kinds of constituent impact arcs
data	1^+ *data*
control	0^+ *data* and 1^+ *control*
operator	0^+ *data* and 1^+ *operator*
reference	0^+ *data* and 1^+ *reference*
ctrl-ref	0^+ *data* and 1^+ *control* and 1^+ *reference*
op-ctrl	0^+ *data* and 1^+ *operator* and 1^+ *control*
op-ref	0^+ *data* and 1^+ *operator* and 1^+ *reference*
mixed	all others

Examples

In Figure 4.2, the impact path from the definition of `b` at n_3 to the definition of `c` at n_{33} is a *data* impact path. The impact path from the definition of `i` at n_2 to the definition of `c` at n_{33} is a *reference* impact path, while that from n_{13} to n_{33} is an *op-ref* impact path. Similarly, the impact path from the definition of `cond` at n_4 to the definition of `c` at n_{33} is a *control* impact path.

The set of impact paths from node i to node j, denoted $Paths(i, j)$, can be partitioned into the subsets $Paths(i, j, \kappa)$, where $\kappa \in \{$*control*, *data*, *operator*, *reference*, *ctrl-ref*, *op-ctrl*, *op-ref*, *mixed*$\}$, such that all the paths in a subset have the same impact kind.

Observable Nodes

Recall that the observable program behavior consists of the output state and the accumulated execution time. The accumulated execution time is used by the test oracle in determining whether the program execution is in an indefinite iteration or indefinite recursion. Given this, an entity instance x in an execution impact graph is *observable* in the following cases:

- x's value appears in the output state, or
- x's existence has potential control impact by avoiding an output state modification, or
- x's existence has control impact by exiting a loop or returning from a function, or
- x's existence has potential control impact by avoiding a loop exit or a function return.

In the first case, x is the argument to an output function such that x's value is reproduced in the output. The other three cases represent the control impact of a code segment node, and correspond to the attributes potentially-controls-output, controls-exit and potentially-controls-exit, respectively. These represent useful roles played by decision branches. And since the corresponding decision branches may or may not be associated with data state modifications affecting observable behavior, it is important to capture these roles explicitly. However, in each of the three cases, the corresponding event is not strictly observable. For example, it is likely that even if a loop exit is missed, the loop may eventually terminate without affecting the observable behavior.

An entity node in a program impact graph is *observable* if during some execution, one of its instances is observable. For example, the arguments of an output function are observable.

4.6 Related Work on Program Dependence Graphs

The concept of the program impact graph was developed by modifying the notion of a program dependence graph in order to clearly represent the impact-relationships that exist among program entities. In this section, we first present a brief overview of the research work in the area of program dependence graphs. Then we discuss the reasons why it was necessary to modify the notion of a program dependence graph in order to support dynamic impact analysis.

A program dependence graph representation of a program is used in several applications: code optimization, parallelization, debugging and software testing. Dependence graphs were introduced by Kuck, Muraoka and Chen [38] as an intermediate representation suited for performing optimizations. Ferrante, Ottenstein and Warren [16] introduced the program dependence graph in order to provide a "unifying framework in which previous work in program optimization may be applied" (page 319). They describe a program dependence graph in which the nodes are statements and predicate expressions, and the edges represent program dependencies. They provide suggestions as to how one could construct a fine grained program dependence graph in which the nodes are operators and operands. Each dependence in a program is classified as being either a control dependence or a data dependence. Data dependencies are further classified as *flow dependence* [25], *output dependence* [37], or *antidependence* [37]. A flow dependence represents a definition-use association, an output dependence represents the sequencing of multiple definitions of the same object and an antidependence represents sequencing between a use and a definition of a variable. The data dependencies are

further characterized as *loop-carried* or *loop-independent*. The former captures the dependencies that arise due to multiple iterations through a loop, while the latter captures the dependencies that occur because of execution order, regardless of loop iteration.

Horwitz, Prins and Reps [25] evaluated the adequacy of program dependence graphs for representing programs. They introduced the notion of a *def-order dependence* [24]. A def-order dependence establishes a partial order among the definitions of a variable that reach a specific use of the variable. In order to define an algorithm for interprocedural slicing, Horwitz, Reps and Binkley [26] proposed the notion of a system dependence graph that extends the notion of a program dependence graph to include procedure calls.

Ottenstein and Ellcey [58] discuss the practical difficulties encountered while implementing a program dependence graph as defined in [16] and describe their approach in resolving those difficulties. They suggest ways to represent array references, procedure calls, input-output and case statements.

Podgurski and Clarke [62] introduce the notions of *weak control dependence* and *strong control dependence*. "The essential difference between weak and strong control dependence is that weak control dependence reflects a dependence between an exit condition of a loop and a statement outside the loop that may be executed after the loop is exited, while strong control dependence does not" [62, page 968]. They refer to data and control dependencies as *syntactic dependencies* and introduce the concept of *semantic dependence*. Informally, a program statement s is semantically dependent on statement s' if the function computed by s' affects the execution behavior of s in some execution. The results of their study indicate that although syntactic dependence is a necessary condition for semantic dependence, it is by no means sufficient. In principle, we agree with the importance of semantic dependencies advocated by Podgurski and Clarke. In fact, roughly speaking, the goal of dynamic impact analysis is to examine and

quantify the semantic dependencies between various program entities and the output. However, the definition of semantic dependence proposed in [62] is not concrete enough for effective use in dynamic impact analysis. In particular, it does not specify how to determine whether a semantic dependence has been demonstrated or not.

The program impact graph proposed in this chapter is conceptually similar to the fine grained program dependence graph suggested by Ferranti, Ottenstein and Warren [16]. This follows since the impact relation can be looked upon as an inverse of the dependence relation. That is, when an entity B depends on entity A, we can say that A can potentially have an impact on B. However a program dependence graph was not considered adequate for providing the infrastructure required to support dynamic impact analysis. We discuss below the three main reasons for modifying the notion of a program dependence graph in order to support dynamic impact analysis.

In the program impact graph, besides data and control dependencies, we also represent reference dependence and operator dependence. For example, in the expression $a[i] + b$, the value of i is used in an address computation. An incorrect value of i would cause an incorrect memory location to be accessed, and depending on the data stored in that location, the result of the addition would either be correct or incorrect. On the other hand, if the value of b is incorrect, the result of the addition operation will also be incorrect. Our experience has been that address computation errors behave differently from data computation errors. Hence it was important to represent address computations and their impact on the program behavior. Also, in order to capture the effect of operator faults, it was necessary to represent the impact due to an operator.

In a program dependence graph, the flow dependence edges traditionally represent dataflow between a definition and a use of a variable. In order to understand the impact of a variable use on the output, we need a more detailed dataflow representation. Therefore, the program impact graph explicitly represents the dataflow from an operand to the

result of an operation or from a function result to the temporary variable representing a function return value at the call-site, and so on.

Often, a decision branch affects program behavior by avoiding a state change. For example, in an `if` statement without an `else` part, the *false* branch may affect the program behavior by *avoiding* a variable definition. In a program impact graph, we have the ability to represent the impact due to a decision branch which affects program behavior by *avoiding* one or more of the following kinds of state-changes: an output, a variable definition, indefinite iteration, and indefinite recursion. A program dependence graph does not provide means for representing such impact. Korel and Laski [36] introduced the notion of a *potential influence* in the context of fault localization. In a very recent paper, Agrawal, Horgan and Krauser [3] revised this notion to a *potential dependence* while defining the concept of a *relevant slice*. According to the revised definition, "the use of a variable v, at a location, l, in a given execution history is said to be *potentially dependent* on an earlier occurrence, p, of a predicate in the execution history if (1) v is never defined between p and l but there exists another path from p to l along which v is defined, and (2) *changing* the evaluation of p may cause this untraversed path to be traversed" [3, page 353]. This definition considers only the potential impact of a predicate on the data state. Our notion of potential control impact also includes the potential impact of a predicate on the output state and the control state.

The reasons mentioned above will become clearer in our use of the program impact graph while carrying out dynamic impact analysis, as presented in chapter 5.

Chapter 5

Dynamic Impact Analysis

5.1 Introduction

The objective of dynamic impact analysis is to measure the error sensitivity of the output of an execution to the potential sources of errors in the execution. A transition from one execution state to another could introduce an error in the execution. Similarly, an incorrect operand or operator could introduce an error in the execution. Therefore, we treat all entity instances as potential sources of errors and for each entity instance x, we would like to estimate the sensitivity of the output to errors in x. Usually, there is no probabilistic element in error propagation; a specific error in x either propagates or does not propagate to the output. Therefore, when speaking of the sensitivity of the output to errors in x we are implicitly referring to the probability that an error, *randomly chosen* from a set of plausible errors for x, when introduced in x, will propagate to the output. Clearly, this probability depends on the set of errors for x and the probability distribution used for randomly choosing an error from the set. So the first problem is to determine the sets of plausible errors for each entity instance in an execution and the relative importance of the errors in each error set. For the time being, let us assume

that we have this information and we compute the above probability (say p_x) for the entity instance x. How should we interpret the value of p_x? When p_x is high (near 1.0), it means that the output is very sensitive to an error in x. When p_x is low (near 0.0), it means that the output is very insensitive to an error in x. We can relate this to the correctness of the value of x as follows.

- If the output is sensitive to an error in x, and if the output is correct, then the value of x is likely to be correct.
- However, if the output is insensitive to an error in x, we cannot say anything about the correctness of the value of x.

Thus, if we can compute p_x for every entity instance x in an execution impact graph, that would meet our objective of measuring the error sensitivity of the output to potential sources of errors in the execution. However, as discussed in section 3.4, the computation of these probabilities is extremely expensive. Therefore, we define heuristics to *estimate* p_x. The estimate is referred to as the *impact strength* of x, and it serves as a measure of the sensitivity of the output to errors in x. In this chapter, we describe these heuristics and the resulting algorithm for carrying out dynamic impact analysis. The motivation behind the design of the heuristics was to achieve low computational complexity, preferably proportional to the complexity of the original execution with a small constant of proportionality. In order to achieve this complexity goal, we made several compromises in our design of the framework for dynamic impact analysis. Each of these compromises are clearly identified throughout this chapter.

5.2 New Concepts

5.2.1 Acceptable Value Set

Consider an expected output value v. If v is an integer value, we say that a value v' is erroneous if $v' \neq v$. However, in the context of floating point numbers, when we say that v' is erroneous, we mean that v' is not in the acceptable range of v. To capture this notion in our formal framework, we define the acceptable value set of entity instance x, denoted by $A(x)$, as the set of all the values of x that will be acceptable to the test oracle. In a numeric program, the acceptable value set for an output value may be specified as $\{v \pm \delta\}$, where δ is the acceptable error. Usually, for entity instances with integral or enumerated values, $A(x)$ is a singleton set.

5.2.2 Error Set and Error Distribution Function

We are interested in measuring the sensitivity of the output to a set of plausible errors in an entity instance x. Therefore, we associate the abstract concept of an *error set* with each entity instance. Ideally, the error set of x should represent the kinds of errors possible in x during an execution due to various plausible faults in the program. However, in the absence of well-accepted fault models for software, we believe that it is not possible to accurately model this ideal situation and therefore it is necessary to consider approximate models.

Taking a naive approach, one could define the error set of x independent of the effects of plausible faults as follows.

$$\varepsilon(x) = \{w \mid w \notin A(x) \wedge w \in W(E^{-1}(x))\}$$

Recall that $E^{-1}(x)$ is the entity corresponding to x, $W(X)$ is the set of all valid values allowed by the entity X, and $A(x)$ is the set of acceptable values for x. $\varepsilon(x)$ is a fault independent error set since it considers all possible allowable error values for the entity

instance x. For example, let x be a reference to a variable of type integer with the value 7. The fault independent error set, $\varepsilon(x)$ is given by $\{v|v \neq 7, v \in I\}$, where I is the set of machine-representable integers.

A problem with this model is that it gives undue importance to error values that are not plausible for x. In reality, some error values are more likely than others. In order to model this, one could choose the error sets based on the information about the effects of plausible faults. A fault dependent error set of x, denoted by $\varepsilon(x, \mathcal{F})$, represents the alternate values of x that could result when a fault represented by $\mathcal{F}$ is present in the program. Note that $\varepsilon(x, \mathcal{F}) \subseteq \varepsilon(x)$. There are two problems with this approach. First, there are no well-accepted fault classifications for software. Second, even if we were targeting a well-defined fault class $\mathcal{F}$, this approach does not specify how to compute the effect of the faults represented by $\mathcal{F}$ on the values of various entity instances in the program. For example, let x be an entity instance with the value 7. Suppose we were interested in the class of faults represented by off-by-one faults. One might be inclined to define the fault dependent error set for this fault, $\varepsilon(x, \mathcal{F})$ to be be $\{6, 8\}$. However, this error set does not capture the effect of off-by-one faults at other locations in the program.

Given these intrinsically difficult problems, we take a pragmatic approach to approximate the ideal situation described earlier. Rather than representing all possible errors in x as a single class of errors, as in the fault independent approach, we consider different commonly observed error classes. An error in an entity instance x could either represent an error propagated from elsewhere or be an error caused by a fault directly associated with x. Presently, we consider three classes of errors to capture the effect of a fault on the value of entity instances *immediately* affected by the fault: *delta errors*, *arbitrary errors* and *reference errors*. The *arbitrary errors* of x are selected by uniformly sampling $W(X)$, the set of allowable values for X. The *delta errors* of x represent errors

caused by slight modifications to the value of x. The *reference errors* of x represent the errors caused by referencing an entity other than X. The details regarding the actual computation of these errors are deferred until section 6.2.1. In addition to these three error classes, we also consider the error class representing *propagated errors*. The *propagated errors* of x are those transmitted from x's impact predecessors. Once again, the details of computation of the propagated errors are deferred until section 6.2.1. The error sets of x are denoted as $\varepsilon(x,\mathcal{C})$ where $\mathcal{C}$ refers to one of the four error classes: *delta*, *arbitrary*, *reference*, and *propagated*.

In order to randomly choose an error from an error set, we need to specify a probability distribution function over the range of values in the error set. One could use such a distribution function to define the relative importance of various kinds of error classes. However, for the purpose of this thesis, we will assume a uniform distribution unless otherwise specified.

Note that the notion of an error corresponds to an entity instance. This is different from the notion of a mutant corresponding to an entity. For example, when a variable reference X is replaced by Y in the program $\mathcal{P}$, it is referred to as creating a mutant. However, when the value of entity instance x is modified in the execution $\mathcal{T}$, it is referred to as introducing an error which results in a *slightly altered execution* $\mathcal{T}'$.

Compromise: Our definition of error sets approximates the ideal situation in which the error set of x would represent the effect of all plausible faults. The classes of errors considered in the above model do not represent all kinds of errors. For example, one could consider an error class that represents periodic repetition of the value of x or one that represents the special values for the type of X. Our position is that the basic error classes can be refined and new error classes can be added once we demonstrate the usefulness of this approach.

5.3 Impact Strength

The execution impact graph captures the dynamic impact relationships among various entity instances that were demonstrated during the execution. Not all impact relationships are alike. Not only do they differ in the kind of impact, the individual "strength" of the impact may also vary. Intuitively, x's impact on y is strong if any error in the value of x is reflected as an error in the value of y. The impact is weak if an error in the value of x does not always result in an error in the value of y. Thus, the strength of x's impact on y should indicate the sensitivity of y to errors in the value of x. In this section, we formalize this intuitive notion of *impact strength* for dynamic impact relationships.

Let $\mathcal{T}$ be an execution impact graph and x and y be two entity instances within the impact graph. As mentioned in the previous section, when speaking of the sensitivity of y to errors in x, the following two parameters are important: the set of errors for x and the probability distribution used for randomly choosing an error from the set. Given these two parameters, we define the strength of x's impact on y as the probability that an error, randomly chosen from a set of plausible errors for x, when introduced in x, will propagate to y. However, as mentioned before, the computation of this probability is extremely expensive, hence we define heuristics to estimate it.

There are various connectivity situations between x and y that need to be investigated:

- there is an impact arc $\langle x, y \rangle$ and that is the only impact path from x to y, or
- there is only one impact path of length > 1 from x to y, or
- there are multiple impact paths from x to y.

The following subsections deal with each of these situations.

5.3.1 Strength of an Impact arc

Each impact arc can be looked upon as a filter which may or may not allow an error to be propagated. Intuitively, an impact arc which propagates all the errors represented by an error set is said to have a strong impact with respect to that error set. On the other hand, an impact arc which does not propagate any error represented by an error set is said to have no impact with respect to that error set. The following definitions of impact strength attempt to capture this intuition.

In a specific execution $\mathcal{T}$, consider entity instances x and y and the impact arc $\langle x, y\rangle$ connecting them. Recall that $V(x)$ denotes the value of the entity instance x and $\varepsilon(x, \mathcal{C})$ denotes the error set of x representing the error class $\mathcal{C}$. By definition of $\varepsilon(x, \mathcal{C})$, $V(x) \notin \varepsilon(x, \mathcal{C})$.

Consider a hypothetical situation, in which the value of x is changed from $V(x)$ to $V'(x)$ while the values of all other entity instances in the impact predecessor set $I_{pred}(y)$ remain unchanged. Assume that the new value of x to cause a change (if any) in the value of y only via the impact arc $\langle x, y\rangle$. That is, if there is another impact path from x to y, the error is assumed not to propagate via that path. Let $V'(y)$ be the resulting new value of y. Recall that $A(y)$ is the acceptable value set of y. The following definition of impact strength applies to this hypothetical situation.

Definition 1 *Given that $V'(x)$ is randomly chosen from the error set $\varepsilon(x, \mathcal{C})$, the impact strength of an impact arc $\langle x, y\rangle$, denoted by $ImpS(\langle x, y\rangle, \mathcal{C})$, is defined as $P[V'(y) \notin A(y)]$. This assumes that the error in $V(x)$ can propagate only along the impact arc $\langle x, y\rangle$.*

Thus, the impact strength $ImpS(\langle x, y\rangle, \mathcal{C})$ represents the probability of transmitting an error representing the error class $\mathcal{C}$ from x to y along the impact arc $\langle x, y\rangle$. For an impact arc contained in computation operations, this impact strength can be computed

Example:
Consider an addition operation, `h + f` with result `r`. Let the error set of `+` representing delta errors be the set of all other binary arithmetic operators of the C language and the two special operators $\lhd$ and $\rhd$.[a] That is, $\varepsilon(\texttt{+}, delta) = \{\texttt{*}, \texttt{-}, \texttt{/}, \texttt{\%}, \lhd, \rhd\}$. Let $\{.2, .4, .2, .1, .05, .05\}$ be the corresponding error distribution function.[b] That is, 0.2 is the probability of incorrectly using `*` instead of `+`, 0.4 for using `-`, and so forth. In the following table, we compute the impact strength for the impact arc $\langle \texttt{+}, r \rangle$ for three different combinations of operand values.

a + b $\Rightarrow$ r	Operators resulting in error	Impact strength for $\langle \texttt{+}, r \rangle$	
3 + 0 $\Rightarrow$ 3	{*, /,%, $\rhd$}	.55	$(.2 + .2 + .1 + .05)$
2 + 2 $\Rightarrow$ 4	{-, /,%, $\lhd$, $\rhd$}	.8	$(.4 + .2 + .1 + .05 + .05)$
3 + 1 $\Rightarrow$ 4	{*, -, /,%, $\lhd$, $\rhd$}	1.0	$(.2 + .4 + .2 + .1 + .05 + .05)$

Note how the impact strength varies for various instances of the same operation.

[a] The special operators $\lhd$ and $\rhd$ simulate alternate simplified expressions: $a \lhd b \equiv a$ and $a \rhd b \equiv b$.

[b] This distribution was selected for illustrative purposes only. In our implementation, we use a uniform distribution.

Figure 5.1: Strength of an Impact arc

by actually executing the operation involving the impact arc for each error in the error set and obtaining a frequency estimate for the probability. In case the error set is infinite or very large, the frequency estimate obtained from a small representative sample of the error set may suffice. The example in Figure 5.1 illustrates the computation of the strength of an impact arc.

Compromise: In order to avoid a combinatorial explosion, the interaction among the errors in the operands of an operation is not considered in defining the strength of an impact arc. As a consequence, the failure of error propagation due to cancelling errors (§3.4, page 25) is not accounted for in the impact strength computation.

5.3.2 Cumulative Impact Strength

Now we consider the other two connectivity situations between entity instances x and y.

- there is only one impact path of length > 1 from x to y, or
- there are multiple impact paths from x to y.

Strength of an Impact Path from x to y

Consider, once again, a situation in which the value of x is changed from $V(x)$ to $V'(x)$. Assume that the new value of x causes a change (if any) in the value of y only via the impact path $\{x, e_1, e_2, \cdots, e_{m-1}, y\}$. That is, if there is another impact path from x to y, the error is assumed not to propagate via that path. Let $V'(y)$ be the resulting new value of y. The following two definitions of impact strength apply to this hypothetical situation.

Definition 2 *Let p denote the impact path $\{e_0, e_1, e_2, \cdots, e_{m-1}, e_m\}$, where $e_0 = x$ and $e_m = y$. Given that $V'(x)$ is randomly chosen from the error set $\varepsilon(x, \mathcal{C})$, the impact strength of the impact path p, denoted by $ImpS(p, \mathcal{C})$, is defined as $P[V'(y) \notin A(y)]$. This assumes that the error in $V(x)$ can propagate only along the impact path p.*

Thus, the impact strength $ImpS(p,\mathcal{C})$ represents the probability of transmitting an error representing the error class $\mathcal{C}$ from x to y along the impact path p. The approach described for computing the strength of an impact arc could also be used here, but it is not practical for the following reasons. First, propagating an error along an impact path may require the execution of an alternate control path. The alternate execution may take very long or may never terminate. Second, it may not be possible to prevent the error in x from propagating to y via impact paths other than p. Considering these problems, we next propose a more practical definition which gives an estimate of the impact strength that satisfies the above definition.

Definition 3 *Let p denote the impact path $\{e_0, e_1, e_2, \cdots, e_{m-1}, e_m\}$, where $e_0 = x$ and $e_m = y$. Given that $V'(x)$ is chosen from the error set $\varepsilon(x,\mathcal{C})$, the estimated impact strength of the impact path p, denoted by $EstImpS(p,\mathcal{C})$, is defined as follows.*

$$EstImpS(p,\mathcal{C}) = min(\ ImpS(\langle x, e_1\rangle,\mathcal{C}),\ \min_{i=1}^{m-1} ImpS(\langle e_i, e_{i+1}\rangle, propagated)\ )$$

Informally, this definition says that the impact strength of an impact path is limited by the strength of its *weakest* constituent arc, where the impact strength of $\langle x, e_1\rangle$ is computed with respect to the error set $\varepsilon(x,\mathcal{C})$ while the impact strength of each of the other arcs is computed with respect to the corresponding error set representing propagated errors. One could also define the strength of an impact path as the product of the strengths of the constituent arcs, but that requires the questionable assumption that the impacts of individual arcs were statistically independent.

Compromise: In order to reduce the computational complexity, the above definition makes several approximations. First, the propagated error set used for computing the strength of the weakest arc is assumed to proportionately represent the effects of the errors in the original error set $\varepsilon(x,\mathcal{C})$. Second, it may be the case that the errors propagated by one impact arc result in the errors that are not propagated by a subsequent

impact arc. In such a case, ideally, the combined strength should be zero even though both impact arcs have non-zero strengths. However, the above definition would compute the combined strength as the minimum of the individual impact strengths. Third, it may be the case that propagating an error results in an impact path different from that used for computing the impact strength. This is possible when there is a control or a reference impact arc in the original impact path. As we shall see later, this last approximation is the primary cause for inaccuracy in the computed impact strengths.

Combining Strengths of Multiple Impact Paths from x to y

We now discuss the general case. Recall that $Paths(x, y, \kappa)$ denotes the set of paths from entity instance x to entity instance y such that each path has impact kind κ, where κ is one of $\{$*control*, *data*, *operator*, *reference*, *ctrl-ref*, *op-ctrl*, *op-ref*, *mixed*$\}$.

Consider a situation in which the value of x is changed from $V(x)$ to $V'(x)$. Assume that the new value of x causes a change (if any) in the value of y via an impact path in $Paths(x, y, \kappa)$. Let $V'(y)$ be the resulting new value of y. The following definitions of impact strength apply to this hypothetical situation.

Definition 4 *Let (x, y) denote a pair of entity instances. Given that $V'(x)$ is chosen from the error set $\mathcal{E}(x, \mathcal{C})$, the impact strength of x's impact on y via paths with impact kind κ, denoted by $ImpS((x, y), \mathcal{C}, \kappa)$, is defined as $P[V'(y) \notin A(y)]$.*

Thus, the dynamic impact strength $ImpS((x, y), \mathcal{C}, \kappa)$ represents the probability of transmitting an error from x to y along the paths with impact kind κ.

Once again, it is not practical to compute the strength of x's impact on y using the above definition for roughly the same reasons mentioned for definition 3 computing the strength of an impact path. Below, we propose a more practical definition which gives an estimate of the impact strength that satisfies the above definition.

Definition 5 *Let (x, y) denote a pair of entity instances. Let $Paths(x, y, \kappa) = \{p_1, p_2, p_3, \cdots, p_m\}$. Given that $V'(x)$ is chosen from the error set $\varepsilon(x, \mathcal{C})$, the estimated impact strength of x's impact on y via paths with impact kind κ, denoted by $EstImpS((x, y), \mathcal{C}, \kappa)$, is defined as follows.*

$$EstImpS((x, y), \mathcal{C}, \kappa) = \max_{i=1}^{m} EstImpS(p_i, \mathcal{C})$$

Informally, the cumulative impact strength of a set of impact paths from x to y is defined as the strength of the *most* sensitive impact path in the set.

Compromise: In general, one would expect the likelihood of propagating an error via a set of impact paths to be greater than that via any single impact path from the set. From this argument, it may seem reasonable to consider the impact paths in a set as statistically independent and use the appropriate probability rules to combine the strengths of individual paths to obtain an even greater estimate for the combined impact strength. However, in our experience, impact paths are rarely independent. Moreover, the errors propagating along two impact paths may cancel one another. Hence, we decided to choose the above definition as a middle ground between the two opposing tendencies.

5.3.3 Impact on the Observable Program Behavior

So far we have defined the impact strength of an entity instance x on an entity instance y. We are interested in computing the impact strength of every entity instance in an execution with respect to the output of the execution. Let O_1, O_2, etc. denote the output entities of the program and ${}_jo_i$ denote the jth instance of the output entity O_i. We would like to combine the impact strengths of x on individual output entity instances, to compute the impact strength of x on the entire output. Once again, we cannot assume that the impacts are statistically independent, hence we choose the following definition:

Definition 6 *Let x denote an entity instance. Let $Paths(x, y, \kappa) = \{p_1, p_2, p_3, \cdots, p_m\}$. Given that $V'(x)$ is chosen from the error set $\varepsilon(x, \mathcal{C})$, the estimated impact strength of x's impact on y via paths with impact kind κ, denoted by $EstImpS((x, y), \mathcal{C}, \kappa)$, is defined as follows.*

$$EstImpS(x, \mathcal{C}, \kappa) = \max_{i,j} EstImpS((x, {}_j o_i), \mathcal{C}, \kappa)$$

Thus, the impact strength of an entity instance x is defined as the maximum of the strengths of its impacts on all of the output entity instances.

Compromise: All of the output entity instances are treated the same by the above definition. In reality, towards the purpose of detecting a failure by observing the output, some output entities are not as useful as others. For example, in the output of a database query, a constant header string in the output is less useful for detecting a failure than the values representing the result of a query. Also, as mentioned during the discussion on observable nodes (§4.5, page 50), not all of the considered output entity instances are strictly observable.

5.3.4 Impact Strength of a Mutation

In order to study the applicability of dynamic impact analysis for strong mutation testing [13, 10], we also define the impact strength of a mutation. Recall from section 2.7 that a mutation is a single syntactic change to the program and the resulting alternate program is called the corresponding mutant. In our framework, we consider only those mutations that require substitution of one program entity in an operation by an alternate program entity such that the program remains *syntactically correct*.[1] Consider a mutation M of the entity X in a program. In an execution of the corresponding mutant, the operation containing M may be executed zero or more times resulting in various instances of the mutation M. An instance of the mutation is said to be *weak-killed*

[1] A program is syntactically correct when it compiles without errors.

if the result of the original operation is different from that of the altered operation containing the mutation instance. For example, consider the operation `2 * 2` which yields the result `4`, and the two operator mutations: one replacing "`*`" by "`/`" and the other replacing "`*`" by "`+`". The first mutation is marked weak-killed since the altered operation produces a different result than the original operation. The second mutation is not weak-killed since the altered and the original operations produce the same result.

For each of the weak-killed mutation instances m, its impact strength is defined as the impact strength of the corresponding original entity instance x. If a mutation instance is not weak-killed, its impact strength is 0. That is,

$$EstImpS(m, \mathcal{C}, \kappa) = \begin{cases} EstImpS(x, \mathcal{C}, \kappa) & \text{if } m \text{ is weak-killed} \\ 0 & \text{otherwise} \end{cases}$$

Clearly, the impact of the mutation M on the output depends on the impact of the weak-killed instances of the mutation. Therefore, we need to select an appropriate function for combining the impact strengths of the weak-killed instances. Once again, the question is, can we assume statistical independence among the instances of a mutation? The answer is yes for some mutations and no for others. It depends on the nature and quality of interaction among the different instances of a mutation. Initially, we did not assume statistical independence and defined the impact strength of a mutation as the maximum of the impact strengths of its weak-killed instances. However, from preliminary experiments, we found that the likelihood of error propagation gradually increased with the number of weak-killed instances of a mutation. In order to capture this gradual increase, we modified our decision and used the following method for combining the impact strengths based on the assumption of statistical independence.

Definition 7 *Let M denote a mutation and ${}_1m, {}_2m, \cdots, {}_nm$ denote the instances of M. Let s_i denote $EstImpS({}_im, \mathcal{C}, \kappa)$, the impact strength of mutation instance ${}_im$. For convenience, let $s_0 = 0$. The estimated strength of M's impact on the output via paths with*

impact kind κ, denoted by $EstImpS(M, \mathcal{C}, \kappa)$, is defined as follows.

$$EstImpS(M, \mathcal{C}, \kappa) = \sum_{i=1}^{n} s_i \prod_{j=0}^{i-1} (1 - s_j)$$

The right hand side of the above equation uses the standard rule for computing the combined probability of success, given individual probabilities of success for n independent trials.

5.3.5 Combining Impacts of Different Kinds

An entity instance x can have different kinds of impact on the output. For example, in the expression `i * a[i-1]`, the variable `i` has both data and reference impact on the result of the expression. The question is: what is the relative importance of the different kinds of impact? Initially, we did not have a clear answer to this question, hence we defined the combined strength of different kinds of impacts simply by taking an average, as shown below.

Definition 8 *Let x denote either an entity instance or a mutation. Let $EstImpS(x, \mathcal{C}, \kappa)$ denote the strength of x's impact on the output via paths with impact kind κ. Let $K(x, y)$ denote the set of different impact kinds represented in the impact paths from x to the output and κ denote a member of this set. The estimated average impact strength of x's impact on the output via all impact paths is defined as follows.*

$$EstImpS(x, \mathcal{C}) = \underset{\kappa}{avg}\ EstImpS(x, \mathcal{C}, \kappa)$$

Later, during our initial validation experiments, we observed that the impact strengths involving only data impact were much more accurate than those involving control and reference impacts. Also from our experience, we know that a substantial fraction of entity instances do not have data impact on the output. Hence we defined the following method for combining different impact kinds so as to give dominance to data impact when present.

Definition 9 *Let x denote either an entity instance or a mutation and $EstImpS(x, \mathcal{C}, \kappa)$ denote the strength of x's impact on the output via paths with impact kind κ. Let $K(x, y)$ denote the set of different impact kinds represented in the impact paths from x to the output and κ' denote a member of $K(x, y) - \{data\}$. The estimated average impact strength of x's impact on y via all impact paths, $EstImpS(x, \mathcal{C})$, is defined as follows.*

$$\begin{cases} avg\ (\ EstImpS(x, \mathcal{C}, data),\ \underset{\kappa'}{avg}\ EstImpS(x, \mathcal{C}, \kappa')\ ) & \text{if } K(x, y) \supset \{data\} \\ EstImpS(x, \mathcal{C}, data) & \text{if } K(x, y) = \{data\} \\ \underset{\kappa'}{avg}\ EstImpS(x, \mathcal{C}, \kappa') & \text{otherwise} \end{cases}$$

Compromise: In the above definition, all impact kinds other than data impact are treated the same. In theory, one could assign different weights to each of the impact kinds based on their relative importance in error propagation. However, with our limited experience with dynamic impact analysis, we could not come up with a justifiable scheme for assigning such weights. As we gain more experience, we hope to address this issue in future.

5.3.6 Summarizing Impact Strengths

$ImpS(\langle x, y\rangle, \mathcal{C})$, the strength of impact arc $\langle x, y\rangle$ with respect to the error set $\varepsilon(x, \mathcal{C})$, represents the probability of transmitting an error representing an error class $\mathcal{C}$ along the impact arc. The presence of the error class parameter $\mathcal{C}$ emphasizes the importance of the choice of error sets while computing impact strengths.

$EstImpS(p, \mathcal{C})$, the estimated strength of the impact path p is defined as the minimum of the impact strengths of the arcs constituting the impact path.

$EstImpS((x, y), \mathcal{C}, \kappa)$, the cumulative impact strength of a set of paths of impact kind κ from x to y, is defined as the strength of the *most* sensitive impact path in the set.

The impact strength of an entity instance x, $EstImpS(x, \mathcal{C}, \kappa)$, is defined as the maximum of the strengths of its impacts on all of the output entity instances.

The impact strength of a mutation M, $EstImpS(\mathrm{M}, \mathcal{C}, \kappa)$, is defined in terms of the impact strengths (s_i) of its instances as $\sum_{i=1}^{n} s_i \prod_{j=0}^{i-1}(1 - s_j)$.

For an entity instance or a mutation, the strengths of different kinds of impacts on the output are combined by taking a data dominant average of the impact strengths.

5.4 Computing Impact Strengths

Having defined the theoretical framework and rationale behind impact strengths, we now describe our algorithm for analyzing an execution to compute the strength of the impact of each entity instance on the observable program behavior. The computational complexity of the algorithm is also discussed. A prototype implementation of dynamic impact analysis is described in chapter 6.

5.4.1 Algorithm Overview

The algorithm for carrying out the impact analysis of a specific execution has two phases.

In the first phase, the program is executed. While executing the program, the strengths of individual impact arcs are computed, and this information is saved in an execution trace. Also, the observable entity instances (those that directly affected the observable program behavior) are marked in the trace. Thus, the execution trace essentially contains the entity instances and the impact strengths of the impact arcs.

In the second phase, the execution trace is processed in reverse. This amounts to traversing backwards in the execution impact graph ensuring that an entity instance is reached only after all its impact successors are processed. When an entity instance y is encountered during this traversal, it is processed as follows.

`if` y is observable,

assign 1.0 impact strength to y

`else`

assimilate all pending impact strengths propagated from y's impact successors

and compute the impact strength of y.

`for each` impact predecessor x of y,

propagate y's impact strength to x via $\langle x, y \rangle$

and append it to the list of pending impact strengths of x.

`end for each`

`end if`.

It is important to note that only a small portion of the execution impact graph needs to be present in the primary random access memory at any point of time during the algorithm. We will discuss this further while presenting the complexity analysis in section 5.4.6.

In the above overview of the algorithm, three components need elaboration: computing the strength of an impact arc, assimilating the pending impact strengths of an entity instance and propagating the impact strength via the arc $\langle x, y \rangle$. These are described in subsequent sections.

5.4.2 Computing Strength of an Impact Arc

Recall definition 1 (page 61) for the impact strength of an impact arc $\langle x, y \rangle$.

> Given that $V'(x)$ is randomly chosen from the error set $\varepsilon(x, \mathcal{C})$, the impact strength of an impact arc $\langle x, y \rangle$, denoted by $ImpS(\langle x, y \rangle, \mathcal{C})$, is defined as $P[V'(y) \notin A(y)]$.

In order to estimate this probability, we first need the error set $\varepsilon(x, \mathcal{C})$, the error

distribution function used for randomly choosing an error from the error set, and the acceptable value set $A(y)$. Section 6.2.1 discusses how we compute error sets in our prototype implementation. We use a uniform distribution for choosing an error from the error set. Also, we use a singleton $A(y)$ with the original value of y as the only member. In the subsection 5.4.5, we discuss how a non-singleton acceptable value set can be incorporated in this algorithm.

For operations such as data or control transfer operations, the impact strength is always 1.0 because any error in x is directly reflected as an error in y. For computation operations and referencing operations, we can estimate the above probability by two alternate approaches. One approach is to select a random sample of alternate values of x from the error set, perform the operation with each of these alternate values, count the number of times the resulting value of y has an error, and thus compute a frequency estimate for the above probability. Another approach is to use a rule-based scheme. That is, for each operation kind, use a set of rules which examine the operands and estimate the above probability. In fact, one could use a combination of the two approaches: use the first approach for some kinds of operations and the second approach for the rest. In our prototype implementation (§6), we used only the first approach. An important point to note is that this estimate should be performed in time bounded by a constant, as explained in the section 5.4.6 on complexity analysis. Hence, we require that the size of the sample of alternate values be bounded by a constant.

5.4.3 Propagating Impact Strengths

While back-propagating an impact strength along an impact arc $\langle x, y \rangle$, we combine the strength of the impact arc with the strength of entity instance y by taking the *minimum* of the two strengths in accordance with the definition 3 on page 64. The impact strength thus combined represents the impact strength of the set of impact paths from x to the

output that include the impact arc $\langle x, y \rangle$.

5.4.4 Assimilating Impact Strengths

We process an entity instance y when its turn comes while processing the execution history in reverse. Since the execution impact graph is a directed acyclic graph, this ensures that all of the impact successors of y have been processed before processing y and have propagated their strengths to y. So we first determine y's strength by assimilating the various strengths propagated from its impact successors. Conceptually, this is accomplished by taking the *maximum* of all of the pending impact strengths at y in accordance with the definition 5 on page 65. Actually, in order to keep a constant bound on the number of pending impact strengths, assimilation is carried on the fly as new impact strengths are added to the list of pending impact strengths.

5.4.5 Other Details and Possible Extensions

In order to keep the above description of the algorithm simple, we purposely omitted the following items: handling of the potential control impact, handling of the impact kind, and determining the acceptable value set for floating point values.

Handling of Potential Control Impact

Recall that a code segment node representing a decision arm has three attributes representing potential control impact: *potentially-controls-exit*, *potentially-controls-output*, and *list-of-potentially-impacted-variables*. When a decision arm instance is encountered in phase two, before assimilating its pending impact strengths, these attributes are processed as follows. If the decision arm potentially controls an exit or potentially controls an output, an impact strength of 1.0 is added to the list of pending impact strengths. If the list of potentially impacted variables is non nil, the decision arm could have impacted

those variables by avoiding their redefinitions. We capture this impact as follows. During phase one, when a decision arm is instantiated, the nodes corresponding to the last definition of each of the potentially impacted variables are recorded in the trace record for this decision arm instance. During phase two, our processing strategy ensures that the decision arm instance will be processed before any such variable definition instances. The pending impact strengths, if any, of each such definition instance are added to the pending impact strengths of the decision arm instance. In order to ensure that the time for processing the potential control impact on variables is bounded by a constant, it is important that the number of potentially impacted variables processed per decision arm be bounded by a fixed constant. Therefore, when the list of potentially impacted variables is very large, only a fixed size random sample is processed.

Handling Impact kind

When the impact strength of 1.0 is introduced at an observable entity instance in the second phase of the algorithm, we associate with it an impact kind. If the value of entity instance appears in the output, the associated impact kind is *data*, otherwise it is *control*. The impact strengths for various impact kinds are propagated in parallel. Recall that each impact arc has an impact kind attribute. While combining the strength of an impact arc with a propagated impact strength, the impact kind of the resulting strength is determined by the rules on page 49 in section 4.5. While processing the pending impact strengths at an entity instance, the impact strengths of different impact kinds are assimilated separately. At the end, the strengths of different kinds of impact are combined by computing the data-dominant average as per the definition 9 on page 70.

Determining Acceptable Value Set

The approach of taking a singleton acceptable value set is not always desirable for floating point values. In fact, the notion of acceptable value set was invented primarily for taking care of the fact that in numeric programs using floating point values, usually

a range of values is acceptable as an output value. In such cases, one would need to determine the acceptable value set for various entity instances with floating point values. Suppose that an output value has a tolerance of $\pm\delta$. This does not necessarily mean that all intermediate floating point values in the computation have the same tolerance. In order to determine the acceptable tolerance at various intermediate points in the computation we propose two preprocessing phases, phase-i and phase-ii. In phase-i, we simply execute the program and produce the execution history. In phase-ii, we start with the output tolerance and propagate it backwards through the entire computation as follows. Consider a floating point operation where y is the result of the operation and x is one of the operands of the operation, such that there is an impact arc $\langle x, y\rangle$. Let $\partial y/\partial x$ denote the partial derivative of y with respect to x. Given the acceptable tolerance Δy for y, the acceptable tolerance Δx for x is given by $\Delta y/(\partial y/\partial x)$. This would work for all continuous functions. For piecewise continuous functions one would have to take a one sided partial derivative. This approach is applicable to all floating point arithmetic operators in the C language and commonly used mathematical library functions. However, due to our limited knowledge about numerical programs, we do not claim that this approach is applicable in general.

5.4.6 Complexity Analysis

In a typical application, the dynamic impact analysis algorithm would be run for every test case execution of the program. Therefore, its performance is crucial to the feasibility of the proposed approach. In this section, we analyze the computational complexity of our algorithm. The notations used in the complexity analysis are given below.

- t Number of operations executed during the program execution
- n Number of nodes (entity instances) in the execution impact graph $\mathcal{T}$

e Number of arcs in the execution impact graph $\mathcal{T}$

d Maximum nesting depth of control constructs in the program $\mathcal{P}$

R Maximum number of storage locations used by the program execution at any point of time during the execution. This includes the variable locations allocated on stack, heap or global data space, and the locations allocated for temporaries and function return address.

S Maximum amount of sequential file storage space required by the program execution.

Lemma 1 *The number of nodes n and the number of arcs e in the execution impact graph are each $O(t)$.*

Proof:

The maximum number of operands in an operation is bounded by a constant. When an operation is executed, each of its operands is instantiated at most once. Therefore, the number of entity instances corresponding to each executed operation is bounded by a constant. Hence $n = O(t)$.

The number of arcs in an execution impact graph can be divided into two classes: those that are local to an operation and those that connect entity instances across operations. The maximum number of local impact relationships within an operation is bounded by a constant. There are only two kinds of arcs that are not local to an operation: def-use impact arcs and implicit control impact arcs. Since a node can have at most one incident def-use arc and at most one incident implicit control arc, the number of such non-local arcs is $O(n)$. Hence $e = O(t) + O(n) = O(t)$. □

During the second phase of dynamic impact analysis, when an impact arc $\langle x, y \rangle$ is processed, the impact strength from y is propagated to x, and the entity instance

x becomes *pending*. It will be processed when the operation containing that node is processed. The following lemma gives an upper bound on the number of such pending entity instances at any time during the second phase.

Lemma 2 *The maximum number of pending entity instances at any time during the second phase of dynamic impact analysis is $O(R)$, where R is the maximum number of storage locations used by the program execution at any time during execution.*

Proof:

For impact arcs that are local within an operation, the impacting entity instance gets processed almost immediately after the impacted entity instance. There are only two kinds of arcs that are not local to an operation: def-use impact arcs and implicit control impact arcs. When a variable use node is processed, some variable definition node receives the propagated strength from the use node and becomes pending. However, at any point of time while processing the execution history in reverse, the number of pending variable definition nodes cannot exceed R. When an impact strength is added to a list of pending strengths, it is actually assimilated into the list, thus ensuring a constant bound on the size of the list.

Recall that a node is an immediate member of the innermost encapsulating code segment node and the implicit control impact arc connects a code segment to an immediate member of that code segment. When an implicit control impact arc is processed, a code segment node receives the propagated strength from an immediate member and becomes pending. The number of such pending code segment nodes is bounded by the expression ($current\text{-}stack\text{-}depth * d$). Note that d, the maximum nesting depth of control constructs, is a small constant. Note also that each stack frame has one return address location associated with it, which is already included in R. This proves the lemma. □

The following discussion shows that the dynamic impact analysis algorithm has the following space-time complexities:

- time complexity of $O(t)$,
- random-access space complexity of $O(R)$, and
- sequential-access space complexity of $O(S) + O(t)$.

We first show how one can compute the impact strength of an impact arc in constant time and space. Then we show that the algorithm indeed has the above space-time complexities.

Computing Strength of an Impact arc in Constant Time and Space

The storage requirement for computing the strength of an impact arc consists of the space for the error set and the space for keeping track of the results of computation.

For an impact arc contained in a computation operation, the calculation of its strength requires executing the operation as many times as the cardinality of the error set. In order to ensure that this calculation takes constant time and space, the size of the error set should be bounded by a constant. Therefore, if the actual error set is very large or unbounded (e.g. an interval set), we select a constant size random sample from that set. This approximation may cause inaccuracies in our computation if the sample does not faithfully represent the entire population, however, it solves the problem of a potentially unbounded computation.

Space Complexity

During phase one of dynamic impact analysis, there are two main sources of random-access storage requirements: the space for program variables and the space for computing impact strengths. The former is clearly $O(R)$. As shown above, the space required for each computation of impact arc strength is constant and it can be deallocated once the computation is complete. Thus the random-access space complexity of the first phase is $O(R)$.

During phase one, the sequential-access storage requirements for saving the execution history and the strengths of impact arcs is clearly proportional to the number of nodes n and the number of arcs e in the execution impact graph. From lemma 1, each of these is $O(t)$. Finally, our algorithm does not affect S, the sequential-access storage requirements of the actual program execution. Thus the sequential-access space complexity of phase one is $O(S) + O(t)$.

During phase two, there are two main sources of storage requirements: the set of pending entity instances (those that have received propagated strengths from one or more of their respective impact successors but are yet unprocessed), and the impact strength attributes on those entity instances. Lemma 2 shows that the maximum number of pending entity instances at a give time is bounded by R. With each entity instance, the maximum number of different kinds of associated strengths is bounded by the total number of impact kinds, which is a constant (in our framework, we have eight different impact kinds). Thus, the random-access space complexity of the second phase is also $O(R)$.

Time Complexity

The time taken for phase one consists of two components: the time for executing the operations and that for computing the impact strength for every impact arc. The former is clearly $O(t)$ since each operation takes a constant time. The latter is also $O(t)$, because from lemma 1, the number of arcs e in the execution impact graph is $O(t)$ and we have shown that computing the strength for an impact arc takes constant time.

The time taken for phase two is directly proportional to the number of nodes and edges in the execution impact graph and the number of potential impact attributes processed in a decision arm. In lemma 1, we have already shown that the number of nodes and edges is $O(t)$. By ensuring a constant bound on the number of potentially

impacted variables processed, the time for processing a decision arm instance is kept within a constant bound. Thus, the time taken for phase two is also $O(t)$.

5.5 Related Work

In this section, we discuss several research efforts involving execution analysis that are related to dynamic impact analysis.

Dynamic Program Slicing

Recall from section 2.8 that computing a dynamic program slice [2, 35] requires processing of an execution history backwards and involves following the data and control dependencies from the operations that affect the values of specified variables. A similar approach is used in the second phase of our algorithm to compute impact strengths. In our algorithm, we start from the observable entity instances in an execution and follow the impact arcs in reverse to reach all of the entity instances that had impact on the output. Thus we traverse the dynamic program slice with respect to the output. Since our goal is to compute impact strengths, we combine and propagate impact strengths while traversing the program slice with respect to the output. In section 9.3 we discuss how impact strengths can be used to compute a more accurate dynamic program slice.

Impact on the Output

Ural and Yang [69] proposed the *All simple OI-paths* criterion which requires exercising simple control-flow paths from input variables to the outputs influenced by the input. They argue that the association between an input variable and an influenced output variable is critical and must be examined during testing. However, no attempt is made to ascertain dynamically whether an input variable actually has influence on the output.

In a recent paper[14], Duesterwald, Gupta and Soffa propose a refinement of the *all-uses* criterion [64]. They consider a definition-use association to be exercised only when some execution of the association has *influence* on at least one output value. Their basic approach is as follows. Let G_s denote the static (program) dependence graph. During an execution, a *dynamic dependence graph* G_d is built such that $G_d \subseteq G_s$. The edges of the dynamic dependence graph represent the dependencies that were demonstrated at least once during the execution. An *output slice* is obtained by starting from the nodes representing output operations and taking the transitive closure using control and data dependencies represented in the dynamic dependence graph. The definition-use associations that are represented in the output slice are said to have *output influence*. This is similar in principle to a sub-goal of dynamic impact analysis, since our algorithm finds the entity instances that had some impact on the program output in a specific execution. There are two important differences between their approach and ours:

- Their approach detects only the no impact case, while our approach also detects the cases of zero impact or weak impact. One of the important thrusts of our research is the notion of impact strength — which attempts to give a quantitative measure of the impact (or "influence"). The validation results described in chapter 7 support our claim that the impact strength reflects error sensitivity with respect to the output.

- Since $G_d \subseteq G_s$, the dynamic dependence graph G_d does not distinguish between different execution instances of a definition-use association. As a consequence of this approximation, their algorithm may incorrectly indicate that an association has influence on the output.

Their approach has the same time complexity as dynamic impact analysis. The space complexities of the two approaches differ mainly in the treatment of the execution trace. In their approach, because of the approximation mentioned above, the execution trace

needs to be processed only in the forward direction. Therefore, one could use a buffer in primary storage to hold parts of the execution trace before they are processed. In our approach, we also need to process the trace in reverse to achieve greater accuracy. Therefore, given limited primary memory, if an execution trace is very large, it will have to be buffered in secondary storage.

Alternate Expression Analysis

In section 5.4.2, we outlined the procedure to compute the strength of an impact arc which is reminiscent of the idea of testing a program expression by distinguishing it from a set of alternate expressions originally proposed independently by Hamlet [23] and Demillo, Lipton and Sayward [13]. Howden [27] used the same technique in his proposal for *weak mutation testing* which was designed to reduce the cost of mutation analysis [13]. Each of these research efforts has been summarized in section 3.3.2.

Sensitivity Analysis

Voas [72] proposed the technique of *sensitivity analysis* with the goal of ranking program locations "based on their ability to impact the program's computation". The term *program location* refers to a program instruction or an operation. From [72], the *sensitivity* of a program location l is an estimate of the minimum probability that a fault in l will result in an output error under a specific input distribution. One factor in the computation of sensitivity is an estimate of the *propagation probability* – the probability (with respect to an input distribution) that an error in the data state at a location causes an output error. On the surface, it *appears* that the notions of impact strength and propagation probability are identical since they both deal with the probability that an error is propagated to the output. However, closer inspection reveals that they are different quantities obtained by different approaches. Voas' propagation probability refers to the likelihood that an error introduced in that location during execution is propagated to

the output for a given input distribution. The program is executed for a large number of inputs to estimate the propagation probability of a single location [72]. When a location has a high propagation probability, it means that for most inputs, an error in the execution of the location will be detected as an output error. In contrast, the impact strength of an entity instance refers to the likelihood that in a *specific* execution, an error injected in an entity instance during the execution is propagated to the output. Using dynamic impact analysis, the impact strengths of *all* the entity instances in an execution are estimated in a time proportional to the execution time. *It is possible that a location has a very low propagation probability with respect to an input distribution, but has a high impact strength for a specific input.* Thus, based on the propagation probability with respect to an input distribution, it is possible that a location is said to have poor testability even though in specific test cases the likelihood of propagation is very high for errors in that location. In contrast, dynamic impact analysis would insist that an entity is said to have poor testability only when it is not possible to find a test case in which an instance of that entity has high impact strength.

Domain/range Ratio

Voas, Miller and Payne [73, 71, 74] introduced the notions of "internal state collapse" and "implicit information loss" while referring to the error propagation behavior of many-to-one functions and attempted to quantify these notions by introducing a metric called *domain/range ratio* (DRR) [73]. The DRR metric and the limitations due to its static nature have been discussed in section 3.4.

Dynamic Error Flow Analysis

Murrill and Morell [50, 48, 49] introduced the technique of dynamic error flow analysis (DEFA) for analyzing error propagation on a path by path basis and for identifying the paths that are more desirable for testing. An execution trace of a presumed correct

program is compared with that of a syntactically close faulty program to study the error creation and propagation behaviors. The objective is to compute various metrics such as the sizes of erroneous data states, the length of the control path between error creation and error masking or output error, the error propagation ability of individual control paths and so on. Although the title and the general objective of this research effort are very similar to ours, the approaches are very different. The impact strength metric estimates the error sensitivity of an impact path rather than a control path. The impact paths associated with a specific instance of a control path may have significantly different error sensitivities. Also, the ability of a control path to propagate errors may depend on the actual data. For these reasons, we investigate error propagation behavior of specific test executions rather than studying the average error propagation behaviors of several test executions that follow the same control path.

Chapter 6

Prototype Implementation

This chapter presents an overview of DIANA, our prototype implementation. The prototype generates the impact graph of a program, carries out the impact analysis of a program execution and provides the infrastructure for conducting the validation experiments described in chapter 7. The purpose of this overview is two-fold:

- to describe the capabilities of the prototype that made it possible to carry out the validation experiments, and
- to demonstrate the feasibility of dynamic impact analysis.

The design of the prototype and the experience gained in developing the prototype using an object-oriented approach may be of interest to some readers. However, in order to focus on the primary objective of our research, we shall not discuss these aspects.

6.1 System Overview

The prototype system consists of four major components:

- a front-end,
- a dataflow analyzer,

- a program representation generator, and
- a flexible program execution environment.

Figure 6.1 gives a logical overview of how these components interact within the prototype. In the figure, oval boxes represent computational modules, plain rectangular boxes/cubes and triangles represent primary memory data structures, and stacked rectangular boxes represent secondary storage data structures. The arrows in the figure represent the information flow within the system. A dotted arrow indicates that there is an implicit underlying computational component which derives the target data structure from the source data structure.

Parsing and Dataflow Analysis

The front-end parses the program under test to produce an attributed syntax tree and a symbol table. The dataflow analysis module processes these and computes definition-use associations.

Generating Program Representation

The program representation generator uses the syntax-tree, symbol-table and def-use associations to generate an intermediate representation that has three different facets:

- an executable interprocedural control flow graph (*icfg*), with nodes representing individual operations, and arcs describing the control flow,
- a program impact graph, with nodes representing program entities, and arcs representing direct impact relationships, and
- program mutations, representing the syntactic alternatives for selected program entities such as variable references, constants and operators.

The program representation is implemented as a relational network using the *Frame system* [20]. The Frame system supports persistence of a relational network. That is,

Figure 6.1: System Overview

the program representation can be saved on the disk and reloaded on demand.

Program Execution Environment

The prototype provides a flexible program execution environment which interprets the executable interprocedural control flow graph and permits arbitrary control over program execution. It supports four modes of program execution, as described below.

- In the *standard* mode, the program is executed without any special control.
- In the *dynamic impact analysis* mode, the program execution is analyzed using the algorithm described in section 5.4. The first phase of the algorithm computes individual impact strengths and produces an execution trace. The second phase of the algorithm processes the execution trace in reverse and computes impact strengths.
- In the *state error execution* mode, the program is run as in the standard mode, except that a specified error is injected into an entity instance during the execution. If the observable program behavior of this slightly altered execution is different from that of the original execution, then we say that the specified state error is detected.

  During the experiments, state errors are produced as follows. The execution trace contains information about data state modifications. Specifically, it contains entity instances corresponding to variable definitions and the data that was written into the corresponding memory locations. A state error is created by randomly picking an entity instance (say x) corresponding to a variable definition and associating with it a data error. The data error (say e) is picked randomly from $W(X)$, the set of allowable values for the entity corresponding to x, with the constraint that e is not equal to $V(x)$, the value of x in the original execution.

- In the *mutant execution* mode, a mutant is created by replacing a program entity by one of its mutations, and the mutant is run as in the standard mode. If the observable program behavior of this execution is different from that of the original standard execution, then we say that the mutant is killed (§2.7).

The validation experiments reported in chapter 7 use the computed impact strengths and the information about the detected state errors and killed mutants. It should be emphasized that the state error execution mode and the mutant execution mode are provided *only* for carrying out validation experiments. These modes are not needed for performing impact analysis.

Development Environment

The front-end and the dataflow analysis components were implemented using the C language as a part of another project at Siemens Corporate Research, Incorporated[61, 59]. We implemented the program representation generator and the execution environment using the common-lisp object system (CLOS[32]). Together, these two modules amount to about 16,000 lines of Lisp code. The Frame system developed by Greenberg [20] provided an excellent environment for implementing and debugging the program representation generator. A Sun-IPC workstation was used to develop the prototype.

6.2 Pragmatic Issues

While implementing dynamic impact analysis, several pragmatic issues had to be resolved. Below, we briefly discuss how the prototype addressed these issues.

6.2.1 Computing Error Sets

The choice of an appropriate strategy to determine the error sets for various entity instances was a very important issue while implementing dynamic impact analysis. Recall

from section 5.1 that when an entity instance x has high impact strength, it means that the output is sensitive to the errors in x. Thus the impact strength may vary depending on what values we include in the error set of x and in the error sets of the nodes along the impact paths from x to the output. In section 5.2.2, we defined the notion of error set and described our approach for computing it. Specifically, we considered four classes of errors: propagated errors, arbitrary errors, delta errors and reference errors. In the following paragraphs, we describe the details for actually computing these errors.

Propagated Errors

The *propagated errors* of x are transmitted from x's impact predecessors. Consider an operation $x \leftarrow y\ op\ z$, with operands y and z, operator op and the result x. Let $V(x)$ denote the value of result x. There are three impact arcs associated with this operation, $\langle z, x\rangle$, $\langle op, x\rangle$ and $\langle y, x\rangle$. Given the error sets of y, z and op, we determine the propagated errors of x as follows. Recall from section 5.4.2 that while computing the impact strength of the above impact arcs, we repeatedly execute the operation with either an operand error or an operator error. Every time the result of the computation is different from $V(x)$, we include the result in the propagated errors of x. In our implementation, this procedure is generalized to operations with any number of operands.

Arbitrary Errors

The *arbitrary errors* of x are selected by uniformly sampling $W(X)$, the set of allowable values for X. If X is of a numeric or the character type, the set of allowable values is obvious. If the value of x is a pointer, the set of allowable values is taken to be the set of all locations with the same type as the location referenced by the value of x.

Delta Errors

The *delta errors* of x represent errors caused by slight modifications in the value of x. The function used to create delta errors depends on the type of x. Let v be the value of x. For integer v, the set of delta errors is selected by sampling uniformly the set $\{v \pm 1, v \pm 2\} \cup [\lfloor v - 10\% \rfloor, \lceil v + 10\% \rceil]$[1] For floating-point v, the set of delta errors is selected by sampling uniformly the set $\{v \pm e_1, v \pm e_2\} \cup [v - 5\%, v + 5\%]$, where e_1 and e_2 are small floating-point numbers. For character v, the set of delta errors consists of the four characters nearest to v in the ordered character set. If the character value is alphabetic, we also include the character obtained by changing its case from lower case to upper case or vice versa. For a pointer value v, the set of delta errors is empty unless the pointer value references an array element (say e). In that case, the set of delta errors consists of the pointer values that reference the four nearest array element locations surrounding the array element e.

Reference Errors

The *reference errors* of x represent the errors caused by referencing a wrong entity instead of X. Suppose that we have the set of mutations of the variable use entity X. (Selection of mutations will be discussed in section 6.2.2). Further suppose that the entity X is instantiated as x at time t during an execution. Then, the set of reference errors for x is the set of values of the variables corresponding to the mutations of X at time t. Reference errors of entity instances of constant references and operators are defined in the same way.

The error set of operator entities consists of only reference errors and every instance of an operator entity has the same error set.

[1] The notation $[a, b]$ refers to the set of values in the interval from a to b, both inclusive.

6.2.2 Selecting Mutations

Recall from section 5.3.4 that a mutation in our framework involves substituting a program entity by an alternate entity such that the program remains syntactically correct. In our prototype, such mutations serve two purposes: for determining reference errors described above and for performing the validation experiments to be described in chapter 7. The following mutation types were used in the prototype.

- operator mutations — replacing an operator by another operator.
- variable reference mutations — replacing a variable reference by another variable reference or a constant reference.
- constant reference mutations — replacing a constant reference by another constant reference or a variable reference.
- structure field reference mutations — replacing a structure field reference by another structure field reference.

While selecting the mutations for an entity X, we first determined the set of all possible applicable mutations such that after replacement, the program will remain syntactically correct. The mutations for X are determined by sampling the set of all possible mutations. Note that the mutations are determined at the time of generating the program representation and are attached as node attributes in the program impact graph.

6.2.3 Handling Library Functions

There are two issues regarding library functions: representation of library functions in the program impact graph, and computation of the associated impact strengths.

In order to capture the impact relationships due to calls to library functions, we

Example 1: scanf(format-string, read-item-ptr1, read-item-ptr2, ···)

```
        Function name:  scanf
          Return type:  integer
        Call-specific:  true
  Required Arguments:
                        arg-name:                        format-string
                        arg-type:                        (pointer char-type)
                        has-impact-on-side-effects:      true
                        kind-of-impact-on-side-effects:  control impact
  Variable Arguments:
                        arg name:          read-item-pointer
                        is-deref-modified: true
```

Example 2: printf(format-string, print-item-1, print-item-2, ···)

```
        Function name:  printf
          Return type:  integer
        Call-specific:  true
  Required Arguments:
                        arg-name:          format-string
                        arg-type:          (pointer char-type)
                        is-deref-output:   true
  Variable Arguments:
                        arg name:   print-item
                        is-output:  true
```

Example 3: atof(number-string)

```
        Function name:  atof
          Return type:  double-float
        Call-specific:  false
  Required Arguments:
                        arg-name:                  number-string
                        arg-type:                  (pointer char-type)
                        has-impact-on-result:      true
                        kind-of-impact-on-result:  reference impact
```

Figure 6.2: Examples illustrating the impact-related aspects of library functions

specified the impact-related characteristics of each library function in a machine readable form. Figure 6.2 gives a few examples in a human readable form to illustrate how we specified the impact-related aspects of library functions. Most of the attributes mentioned in the figure are self explanatory. A library function with a *variable* number of arguments has the *call-specific* attribute true, indicating that its impact relationships depend on the call-site. For such a function, we create a separate function entry operation for each call to the function. With each such function entry operation, we associate as many formals as the number of actuals in the corresponding call. This enables us to establish a one-to-one mapping between the actuals and the formals regardless of the number of arguments.

For the purpose of computing impact strengths, we classify library functions into two categories: those that have side effects and those that do not. While computing impact strengths of impact arcs associated with library functions that do not have side effects, such as the `atof` function in figure 6.2, we treat them the same as the built-in language operators. On the other hand, for a library function that has side effects, such as the `scanf` function in figure 6.2, we assign an impact strength of 1.0 to each impact arc associated with the function. That is, we assume that any error in the argument of a call to a libary function with side effects will result in an error in at least one of the side effects.

6.3 Limitations of the Prototype

This section summarizes some of the limitations of the current prototype.

- It does not handle function pointers, graphical interfaces and structure initializations.

- It uses an abstract memory model in which all variable locations are in the *heap*. Arrays and pointer arithmetic are logically supported, however, the program call stack is not viewed as an array. Therefore programs that use the call stack as an array cannot be handled.
- The prototype is extravagant in its use of space, and so is the Frame system that implements the program representation. Also, the entire program representation is required to be resident in the addressable virtual memory. Hence the size of the program that can be processed by the prototype was limited by the available swap space which has to be shared with the Common lisp environment. Therefore, we have not yet attempted to process subject programs with more than 750 lines.

Chapter 7

Validation

This chapter describes the experiments undertaken to validate the computation of impact strengths and presents the results of the experiments. We first describe the validation approach emphasizing the goals of this empirical study. We then describe the characteristics of the subject programs chosen for the study and the method used in selecting test suites for the subject programs. This is followed by a detailed description of the experimental procedure applied to each subject program. A majority of this chapter is devoted to presenting and discussing the results of these experiments.

7.1 Validation Approach

Our goal is to examine whether the impact strengths computed by dynamic impact analysis reflect the error sensitivity with respect to the output. Specifically, we want to answer the following questions.

1. What is the relationship between the impact strength of an entity instance x in an execution and the likelihood that an error in the value of x will propagate to the output? The answer to this question will indicate the accuracy with which the

impact strength of an entity instance x in the original execution predicts the likelihood of error propagation in a slightly altered execution created by introducing an error in x.

2. What is the relationship between the impact strength of a mutation M in a test case execution and the likelihood that the corresponding mutant will be killed by that test case? The answer to this question will indicate the accuracy with which the impact strength of a mutation in a test case execution of the original program predicts the likelihood of the test case killing the corresponding mutant.

In order to answer the first question, we need to design an experiment that examines the relationship between the impact strength of an entity instance x and the observed propagation of an error in x. One difficulty in designing such an experiment is that the impact strength is a continuous variable in the range [0.0,1.0] while the observed error detection is a boolean variable. Therefore it is unclear how to quantitatively study their relationship. One could create an almost continuous variable corresponding to error detection by considering the frequency of success of error detection among the members of a random set of errors in x. However, this approach is not feasible because often the set of all possible error values of an entity instance is very small. For example, a boolean predicate instance p has only two values, one correct and one incorrect, and hence the size of an error set of p cannot be greater than 1.

Similarly, in order to answer the second question, we need to design an experiment that examines the relationship between the impact strength of a mutation M in a test case and the observation whether the test case kills the mutant corresponding to M. Once again, the former is a continuous variable and the latter is a boolean variable and it is unclear how to quantitatively study their relationship.

Given this problem, we take a pragmatic approach for creating an almost continuous variable from a set of observations corresponding to the boolean variable as outlined

below. The actual experimental procedure is described in section 7.4.

In the first experiment, we are interested in the accuracy with which the impact strength of an entity instance predicts the propagation of error in that entity instance. To answer this question for a given subject program, we use the following method. First, we take a very large sample of entity instances randomly chosen from several different test case executions of the program. Consider an entity instance x in this sample. Let $\mathcal{T}$ be the corresponding test case execution and s be the impact strength of x obtained by analyzing $\mathcal{T}$. We randomly choose an error for x and run an alternate execution $\mathcal{T}'$ obtained by introducing the error in x while running $\mathcal{T}$. The boolean variable p records whether the error is detected at the output or not (1 if detected, 0 otherwise). This gives us a data point (s, d) where s is the impact strength and d is the corresponding boolean value for the error detection. This is repeated for each entity instance in the sample so as to obtain a large number of data points like (s, d). Then we consider all n data points that have the same impact strength s. Let m out of the n data points have error detection as 1. If n is large enough, the error detection ratio m/n is statistically significant[1] and indicates the likelihood of error propagation among entity instances with impact strength s. And since the error detection ratio m/n is a (reasonably) continuous variable, we could now study the relationship between m/n and s.

In the second experiment, we are interested in the accuracy with which the impact strength of a mutation in a test case predicts the killing of the corresponding mutant by the test case. To answer this question for a given program, we use a strategy similar to the one used in the first experiment. We consider the impact strengths of all of the generated mutations of the program obtained by analyzing different test case executions. Let M be a mutation with impact strength s in the execution of a specific test case T. The test case T is run on the mutant corresponding to M and the boolean variable k

[1] If n is small, slight statistical variations in the value of m can result in large variations in the ratio m/n. Typically, $n >= 30$ is considered large enough.

records whether the mutant is killed or not (1 if killed, 0 otherwise). This gives us a data point (s, k). This is repeated for every (mutation, test case) combination, so as to obtain a large number of data points like (s, k). As in the first experiment, we compute the ratio m/n. In this case, m/n denotes the mutant kill ratio and indicates the likelihood of killing the mutants corresponding to the mutations with impact strength s. We then study the relationship between m/n and s.

The actual experimental procedure based on the above approach is described in section 7.4. The experimental procedure was performed separately for each of the subject programs described below.

7.2 Subject Programs

The choice of subject programs is always a difficult issue in conducting an experiment in software engineering primarily due to the lack of a set of benchmark programs representative of the universe of programs. The choice was made more difficult for us because of the following considerations.

- The experimental procedure executes a program thousands of times. Therefore, the average execution time of the subject program could not be larger than one minute.
- As mentioned in section 6.3, our prototype cannot handle large programs. Also, the prototype does not support graphical user interfaces, function pointers, and the array view of the call-stack.

Within these constraints, we tried our best to select programs representing diverse application domains such as as database processing, numeric algorithms, string processing, accounting, graph algorithms, statistical computation etc. We also attempted to ensure that the subject programs represented the various aspects of computer programming

Ref. No.	Program Name (alphabetic order)	Lines of Code (excl. blanks & comments)	(#Operators + #Operands)*	Test Suite Size
		29 to 556	86 to 2381	12 to 51
1	accounting	200	2255	50
2	aggregate-db-reln	556	2117	13
3	altitude-separation	124	448	50
4	bank-promotion	150	765	50
5	biconnectivity	157	406	50
6	binary-search	39	176	34
7	chi-square-test	218	1010	37
8	depth-first-search	121	388	50
9	determinant	52	670	50
10	eval-expr	69	200	48
11	func-zero	117	642	17
12	info-measure	217	574	37
13	join-db-reln	503	2282	12
14	keywords	79	427	17
15	list-ops	173	671	51
16	mortgage	49	207	41
17	pattern-replace	513	2381	25
18	poly-calculus	89	339	50
19	prio-schedule	285	986	50
20	select-db-reln	516	2294	21
21	square-root	36	167	38
22	strong-connectivity	151	654	50
23	tax-form	133	908	50
24	triangle-type	57	239	39
25	word-count	52	262	15
26	x_power_y	29	86	35

* This metric called *program length* was defined by Halstead[21].

Table 7.1: Subject Programs

such as the nature of computation (iterative, recursive, or simple), data structuring facility (basic types, arrays, structures, or pointers), allocation of variables (heap or stack) and input method (command line, file or interactive). Table 7.1 gives the names and reference numbers for each of the 26 subject programs used in the study. In addition, the table gives the test suite size and two metrics of program size for each subject program. Appendix A gives relevant information about each of the programs and refers the reader to an anonymous ftp site for the source code of the programs.

7.3 Selecting Test Suites

For 19 subject programs, test suites were generated in the following way. We used a test specification language (TSL [55, 6]) to specify the choices for input variables of the program. An attempt was made to cover the functional specification of the program, error situations and some extreme values for the input variables. The TSL-tool then generated a test suite from the test specification. From this test suite, we removed duplicate test cases to yield the test suite that was used in the experiments. For the remaining 7 programs, the TSL language was not powerful enough to capture the dependencies among the inputs. Hence for these programs, test cases were hand generated following the same guidelines mentioned above. The source of the programs and the test suites are made available at an anonymous ftp site (see appendix A).

7.4 Experimental Procedure

This section describes the experimental procedure in detail. The procedure takes two inputs: a program and a test suite for the program. In the following description of the procedure, the `accounting` program and the corresponding test suite are used for the purpose of illustration. The steps involved in the procedure are described below.

Step 1: Generating Program Representation

The internal program representation is generated for the subject program. This includes generating the interprocedural control flow graph, the program impact graph and program mutations. The program mutations are selected as described in section 6.2.2. For the `accounting` program, there were a total of 549 mutations including 122 operator reference mutations, 136 constant reference mutations and 291 variable reference mutations.

Step 2: Executing and Killing Mutants

First, the program is run in the standard execution mode for each test case in the test suite and the output is saved. Then the program is run in the mutant execution mode for each (mutation, test case) combination. For each mutant execution, the output is compared with the output of the standard execution. If the outputs are different, the mutation is marked *strong-killed* for that test case. For each test case, the information about which mutations are strong-killed by that test case is saved in a database. For the `accounting` program, each of the 34 test cases were run on each of the 549 mutants, totaling to 18,666 executions.

Step 3: Carrying out Dynamic Impact Analysis

The program is now run in the dynamic impact analysis mode for each test case in the test suite. During the first phase of the algorithm, besides computing and recording the impact strengths of individual impact arcs, we also record the following information in the execution trace:

- *weak-killed* mutations (see §5.3.4),
- entity instances corresponding to data state modifications and the data values stored in the modified locations.

During the second phase of the algorithm, the impact strengths are computed for all of the entity instances and mutations. The impact strengths of weak-killed mutations and the entity instances corresponding to data state modifications are recorded in a database.

Also, the total cpu time for executing the test suite in the dynamic impact analysis mode is compared with the total cpu time for running the test cases in the standard execution mode. We repeated the time measurement several times to ensure that the variation in the timings was not significant (typically less than 10%). For the `accounting` program, on an average, a test execution with dynamic impact analysis was about 6 times slower than that without any analysis. Section 7.5.7 gives similar performance data for all of the subject programs.

Step 4: Generating, Executing and Detecting State Errors

The execution trace produced in the previous step contains a list of data state modifications, the associated data values, and the computed impact strengths for the associated entity instances. From this information, we generate state errors as described in section 6.1 on page 89. The number of state errors generated for a given test case depends, in part, on the length of the test case execution and the number of variable definition nodes in the execution impact graph. For the `accounting` program, a total of 9,984 state errors were generated for the 50 test case executions, varying from 68 to 520 per execution. For each state error, we run the test case in the state error execution mode and check whether the error is detected as an output error. The impact strength of the associated entity instance and a flag indicating whether the state error was detected are recorded in the database.

Step 5: Plotting State Error Detection Ratio vs. Impact Strength

As mentioned above, in the database, for each test case execution, we keep information about the impact strengths of the entity instances associated with the state errors and whether or not the state errors were detected as output errors.

In order that enough data points with different impact strength values are considered, we merge the information from all test cases in the test suite and process it collectively. Later, we will provide evidence that the ability of a test case to detect state errors did not influence the results.

The impact strengths of the entity instances associated with state errors are grouped into six intervals: $[0.0, 0.0]$, $(0.0, 0.2]$, $(0.2, 0.4]$, $(0.4, 0.6]$, $(0.6, 0.8]$ and $(0.8, 1.0]$. For each interval, the *state error detection ratio* is computed as follows. Let n be the total number of data points in the interval. Out of these, let m be the number of data points for which the associated state errors are detected. Then m/n is the state error detection ratio. As mentioned before, this ratio is meaningful only when n is large enough ($n \geq 30$). The state error detection ratio for an impact strength interval represents the likelihood of error propagation among entity instances with impact strength in that interval. The state error detection ratio is plotted against the mid-point of the corresponding impact strength interval as shown in Figure 7.1. If the number of data points in an impact strength interval is less than 30, the corresponding point is not plotted. For completeness, the actual values of m (#DataPts) and n (#Detected) for each interval are also displayed below the x-axis. The graph is annotated with the name of the program and the value of linear correlation[2] between the state error detection ratio and the entity instance impact strength. In section 7.5.1, we will present similar graphs for all of the subject programs and discuss their behavior.

[2]Linear correlation [44] is one way of measuring the *degree of association* between two variables. It has no bearing on the discrepancy between the *expected* and the *observed* relationships.

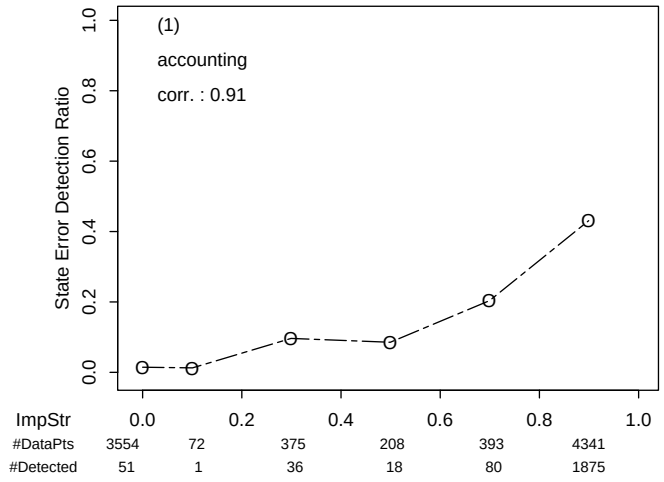


Figure 7.1: Error Detection Behavior of the `accounting` Program

Step 6: Plotting Mutant Kill Ratio vs. Impact Strength

In the database, for each test case, we keep information about the impact strengths of the mutations and flags indicating whether the mutations are weak- and strong-killed by the test case. For a mutation to be strong-killed, it is necessary that it is weak-killed and the corresponding error propagation condition is satisfied. Therefore, for the purpose of validating impact strength computations, there is no point in considering mutations that are not weak-killed.

Once again, in order that there are enough data points with different impact strength values, we merge the information from all the test cases in the test suite and process it collectively. Later, we will provide evidence that the ability of a test case to kill mutants did not influence the results.

The impact strengths of the weak-killed mutations are grouped into six intervals: $[0.0, 0.0]$, $(0.0, 0.2]$, $(0.2, 0.4]$, $(0.4, 0.6]$, $(0.6, 0.8]$ and $(0.8, 1.0]$. For each interval, the *mutant kill ratio* is computed as follows. If n is the total number of data points in the interval and m is the number of data points in the interval for which the associated mutations

are strong-killed, then m/n is the mutant kill ratio. As mentioned before, this ratio is meaningful only when n is large enough ($n \geq 30$). The mutant kill ratio for an impact strength interval represents the likelihood of killing the mutants corresponding to the mutations with impact strengths in that interval. The mutant kill ratio is plotted against the mid-point of the corresponding impact strength interval as shown in Figure 7.2. If the number of data points in an impact strength interval is less than 30, the

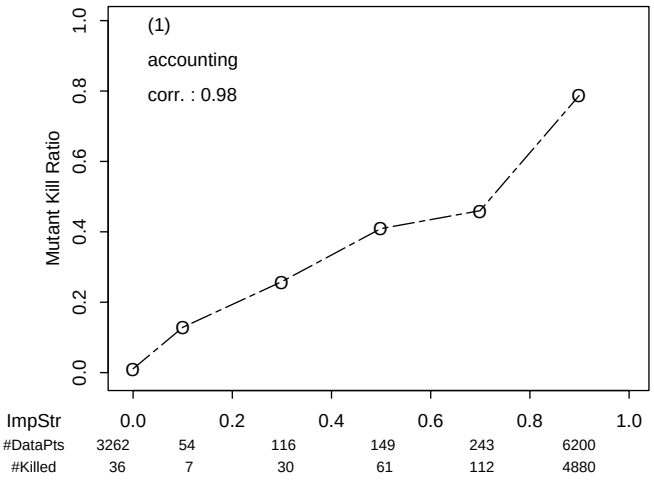


Figure 7.2: Mutant Killing Behavior of the `accounting` Program

corresponding point is not plotted. For completeness, the actual values of m (#DataPts) and n (#Killed) for each interval are also displayed below the x-axis. The graph is annotated with the name of the program and the value of linear correlation between the mutant kill ratio and the mutation impact strength. In section 7.5.2, we will present similar graphs for all of the subject programs and discuss their behavior.

Step 7: Examining Bias Due to Specific Test Cases

It is conceivable that a few test cases that are good at detecting a large number of state errors may bias the results obtained in step 5. In order to examine whether such a bias is

present or not, we carry out the following statistical analysis for each program. The test suite of the program is sorted by the number of state errors detected by each test case and the test cases are ranked by their ability to detect state errors. Then we separately perform step 5 on the top third and the bottom third subsets of the sorted test suite. For each subset, this gives a vector of state error detection ratios. The length of such a vector is six corresponding to the six impact strength intervals. The two vectors of error detection ratios are then compared using the paired t-test[8][3] with the null hypothesis being that the true mean of the observed differences between the corresponding values in the two vectors is zero. Two outputs of the t-test are recorded: the sample mean of the differences and the *p-value* which is the probability that differences equal to or greater than those observed would occur given that the true mean is 0.0. For the `accounting` program, the vector of state error detection ratios is [.0139, .0143, .0853, .0912, .1981, .4380] for the top third test cases and is [.0148, .0137, .0975, .0853, .2063, .4289] for the bottom third test cases. The sample mean of the differences between these vectors is -0.00095 and the p-value is 0.7859. An interpretation of these and similar numbers for other subject programs is presented in section 7.5.3.

A similar procedure is followed to study the bias (if any) introduced in the results obtained in step 6 by a few test cases that are good at killing a large number of mutants. For the `accounting` program, the vector of mutant kill ratios is [.0124, .1290, .2846, .3727, .5102, .7904] for the top third test cases and is [.0103, NA[4], .2000, .4271, .4553, .8121] for the bottom third test cases. From the t-test, we obtain 0.0131 as the sample mean of the differences between the corresponding values in the two vectors and 0.6312 as the p-value. Once again, an interpretation of these numbers is presented in section 7.5.3.

[3]One of the key assumptions while performing a paired t-test is that the differences come from a parent distribution which is *normal*. In our experiment, we have no way of verifying that assumption. However, Box, Hunter and Hunter [8] show that the shape of the parent distribution is less important as long as the random sampling model is appropriate.

[4]An NA value in this context implies that there were not enough data points to compute the mutant kill ratio in that impact strength interval.

CAVEAT

Occasionally, during a mutant execution or a state error execution, the subject program "crashes" due to a runtime error such as a *segmentation fault* or a *bus error* in a Unix environment. Besides the conventional runtime errors in a typical C programming environment, our execution environment detected two additional runtime errors: access of uninitialized locations and out of bound array references.[5] Technically, a runtime error should be considered as the output. However, in the current framework of dynamic impact analysis we define output entities as those entities that participate in calls to predefined output procedures. The operators in a program that may cause runtime errors are not considered as output entities. Therefore it is inappropriate to relate impact strengths with occurrences of runtime errors. Hence, the mutants and state errors that resulted in runtime errors were omitted from the experimental data.

7.5 Empirical Results

In this section, we report the results of the empirical study in which the experimental procedure described in section 7.4 was applied to each of the 26 subject programs. A summary of the results is presented at the end of this section.

7.5.1 State Error Detection Ratio vs. Entity Instance Impact Strength

Figures 7.3 through 7.7 illustrate the relationship between the impact strength of the entity instance associated with a state error and the state error detection ratio, which represents the likelihood of propagating the state error to the output. Each plot is annotated with the corresponding subject program name and the linear correlation measure. For ease of reference, the plots are presented in the alphabetic order of program

[5] In our previous work [19], access of uninitialized locations and out of bound array references were not treated as runtime errors.

names. We make the following observations from these plots.

Strong positive correlation

There is a strong positive correlation (0.86 to 1.0) between the the impact strength of an entity instance and the corresponding state error detection ratio. Usually, the state error detection ratio increases with the impact strength. An impact strength closer to 0.0 implies smaller chances of state error detection and an impact strength closer to 1.0 implies greater chances of state error detection. This supports our claim that the impact strength of an entity instance reflects the sensitivity of the output to errors in the entity instance.

Variety in impact strength distributions

Usually, the number of data points in the highest impact strength interval is the largest. This is to be expected, since typically most entity instances in an execution would have significant impact on the output of the program. However, it is encouraging to observe that there is quite a variety in the distributions of impact strengths. Significant numbers of data points are found in each of the impact strength intervals for subject programs using complex logic such as `pattern-replace` and `chi-square-test`. On the other hand, in subject programs using simple computations such as `mortgage` and `x_power_y`, very few entity instances have low impact strengths. Section 7.5.6 explains how this variety of impact strength distributions justify the computation of impact strengths.

Inaccuracy at zero impact strength

Ideally, we would expect the state error detection ratio to be 0.0 when the impact strength is zero. However, for 15 out of the 26 subject programs, the state error detection ratio at zero impact strength is a small non-zero value, varying from 0.00077 (`list-ops`) to 0.059 (`chi-square-test`). That is, in spite of having zero or no impact, the state errors in a very small fraction of the entity instances got detected. An investigation of

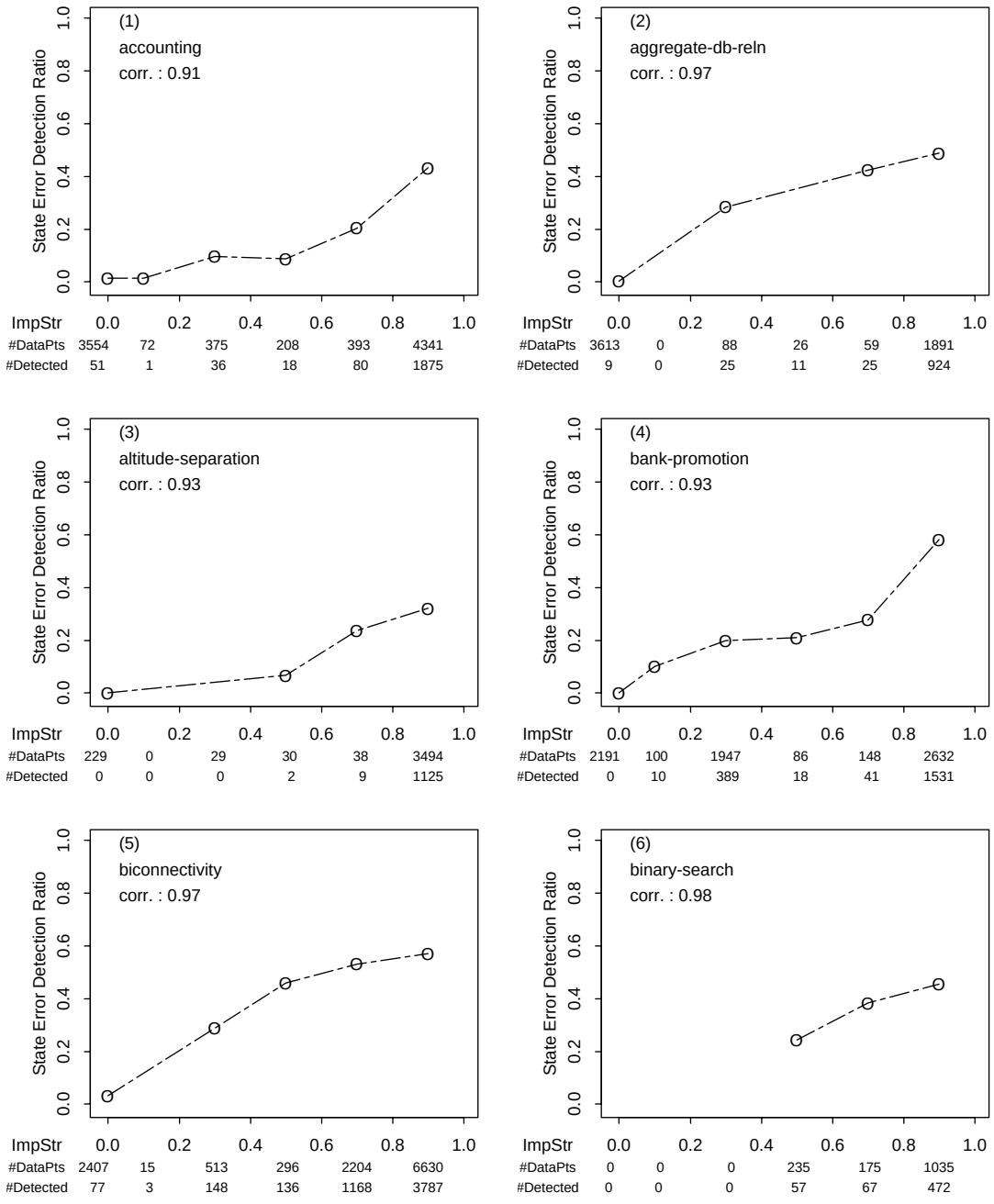


Figure 7.3: Error Detection Behavior of Subject Programs 1-6

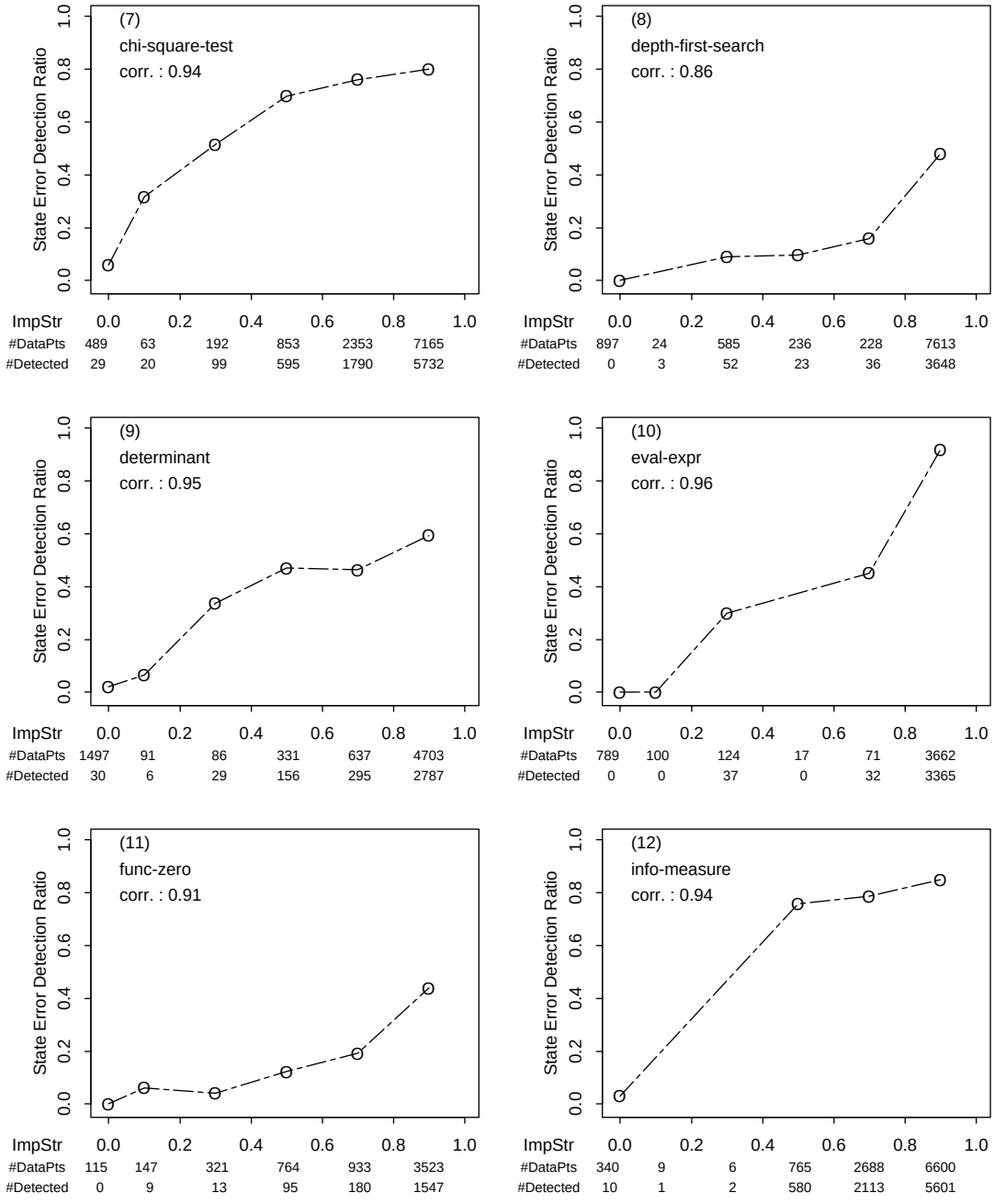


Figure 7.4: Error Detection Behavior of Subject Programs 7-12

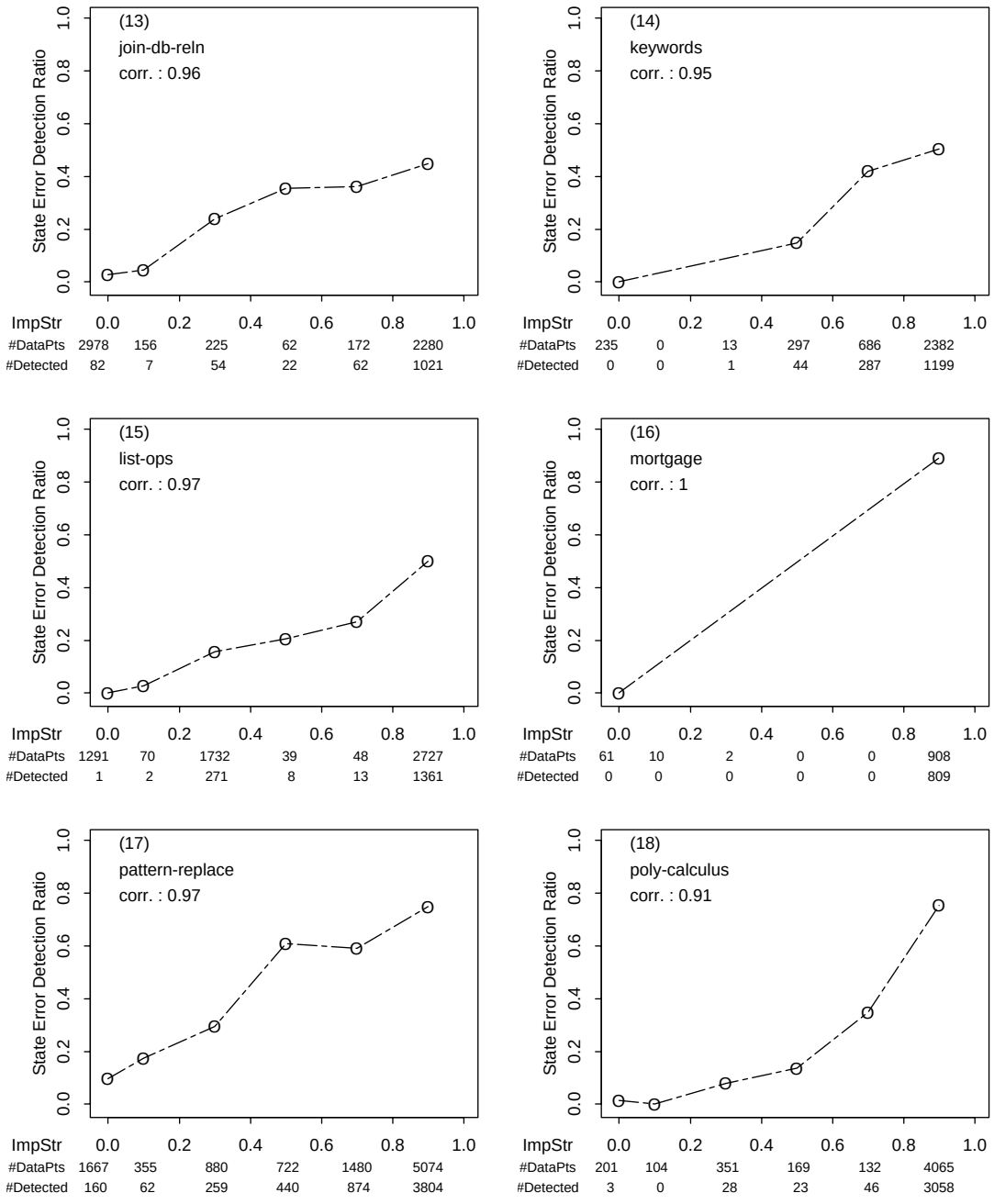


Figure 7.5: Error Detection Behavior of Subject Programs 13-18

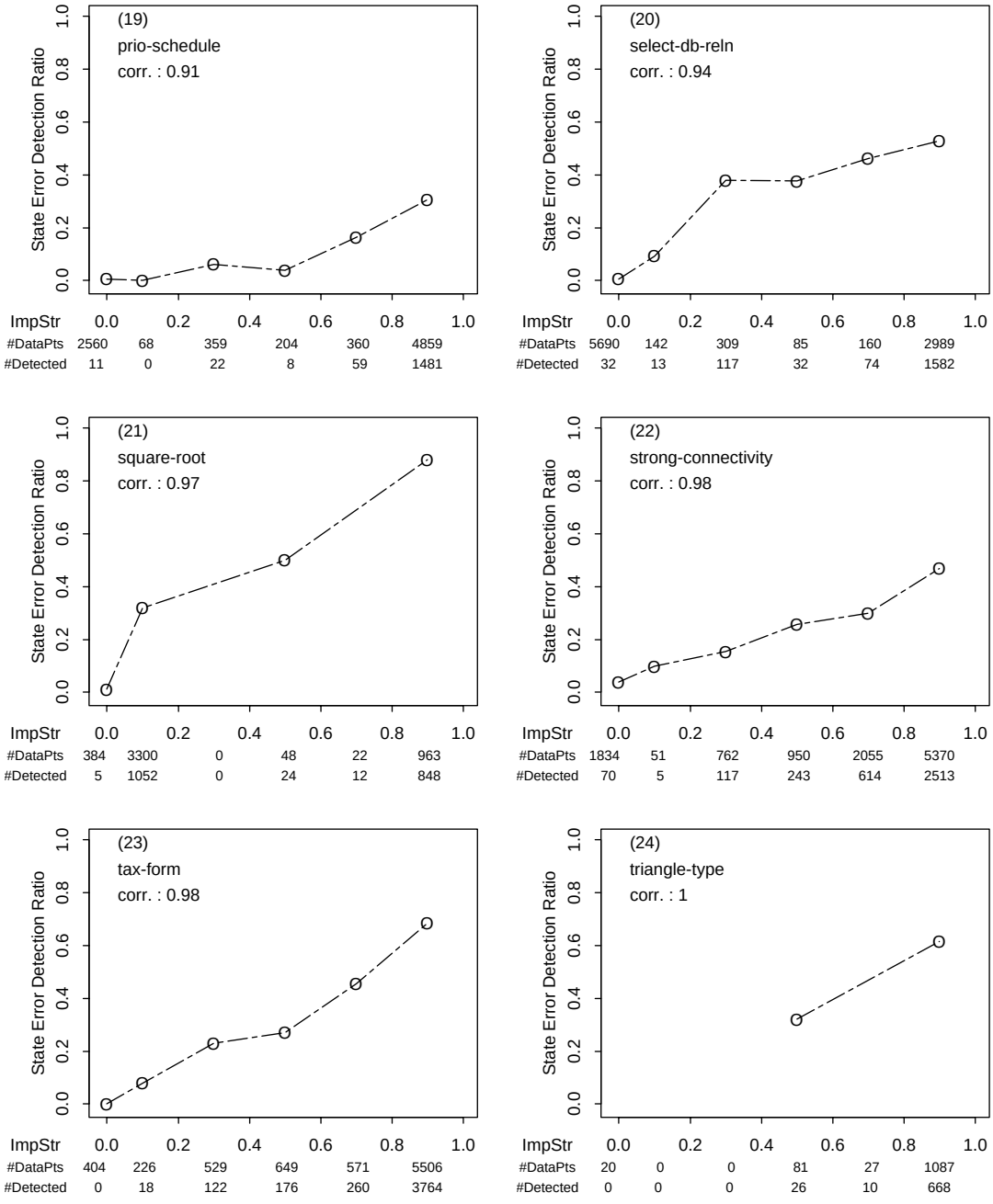


Figure 7.6: Error Detection Behavior of Subject Programs 19-24

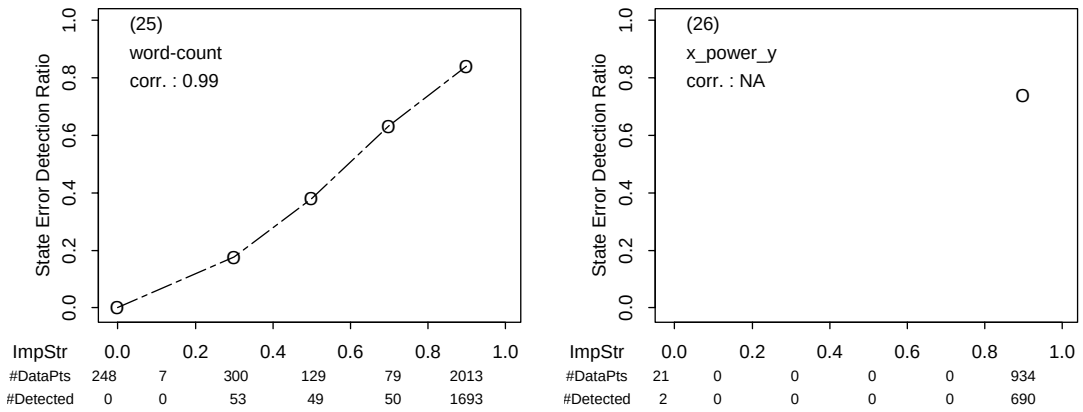


Figure 7.7: Error Detection Behavior of Subject Programs 25-26

this behavior is presented in section 7.5.4.

Inaccuracy at high impact strength

The highest state error detection ratio varies with the subject programs. Specifically, at the highest impact strength interval (0.8,1.0], the state error detection ratio varies from 0.33 (`altitude-separation`) to 0.92 (`eval-expr`). We investigated this behavior further and the findings are reported in section 7.5.5.

7.5.2 Mutant Kill Ratio vs. Mutation Impact Strength

Figures 7.8 through 7.12 illustrate the relationship between the impact strength of a mutation and the mutant kill ratio, which represents the likelihood of killing the corresponding mutant. Each graph is annotated with the corresponding subject program name and the linear correlation measure. For ease of reference, the plots are presented in the alphabetic order of program names. We make the following observations from these plots.

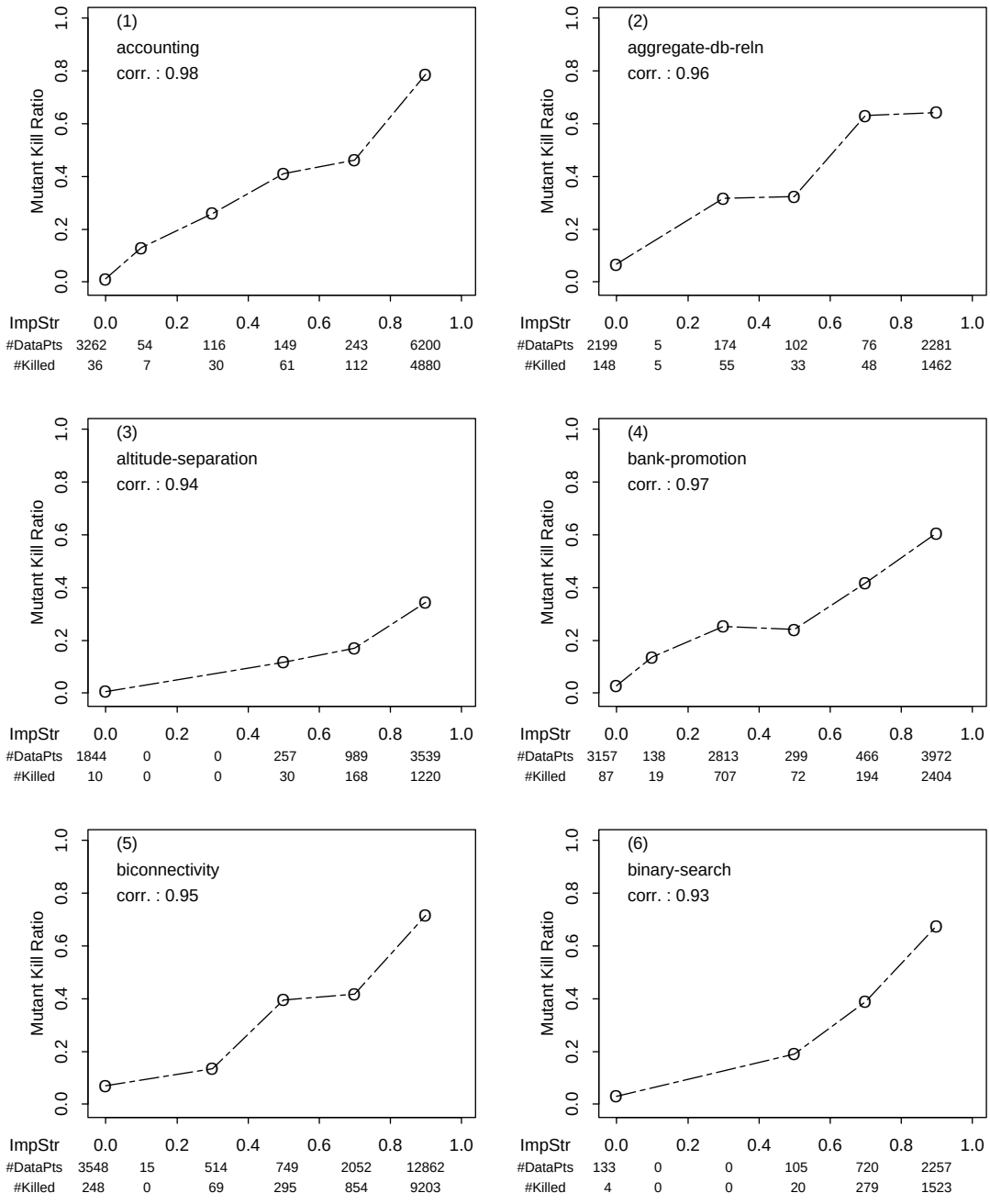


Figure 7.8: Mutant Killing Behavior of Subject Programs 1-6

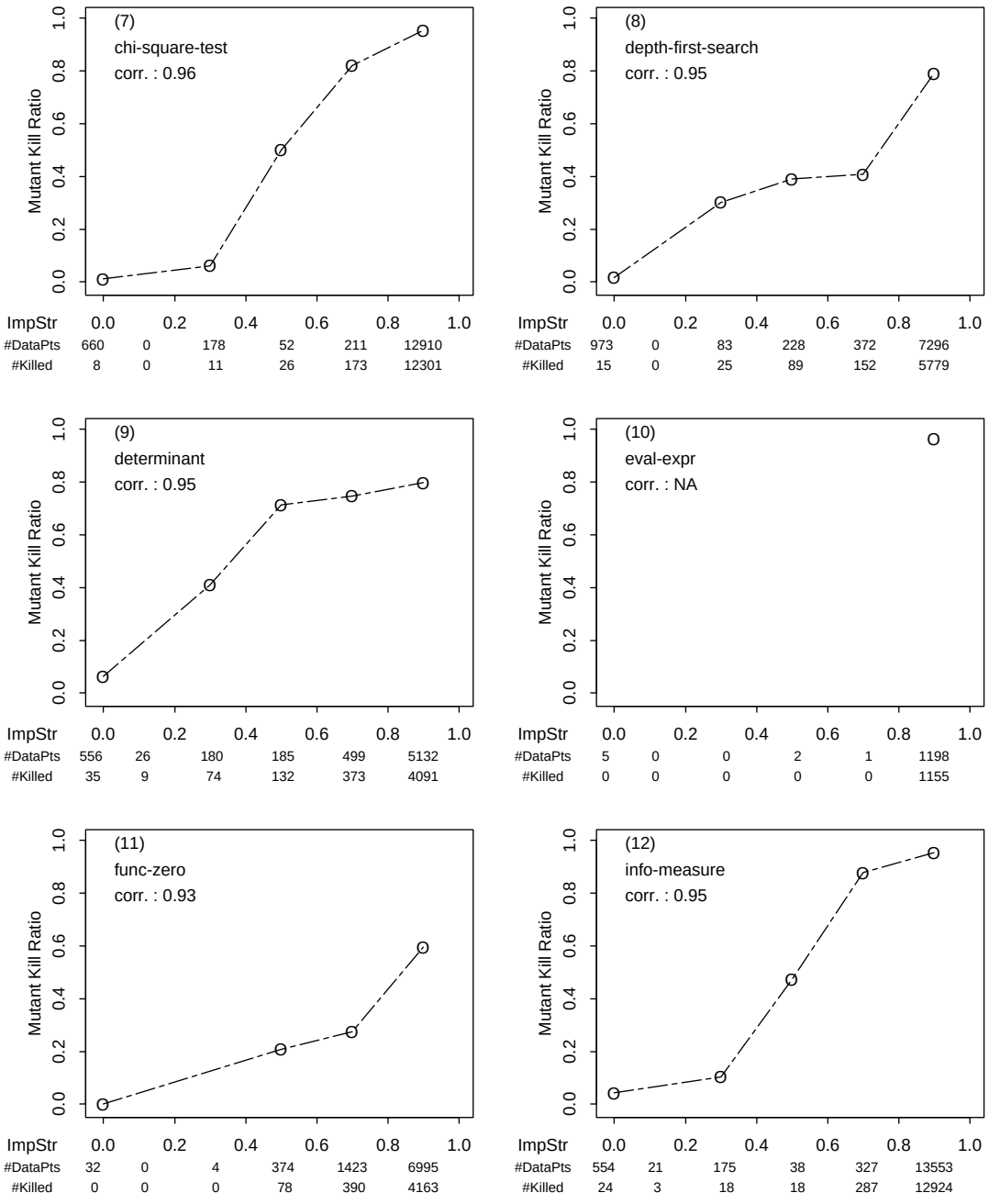


Figure 7.9: Mutant Killing Behavior of Subject Programs 7-12

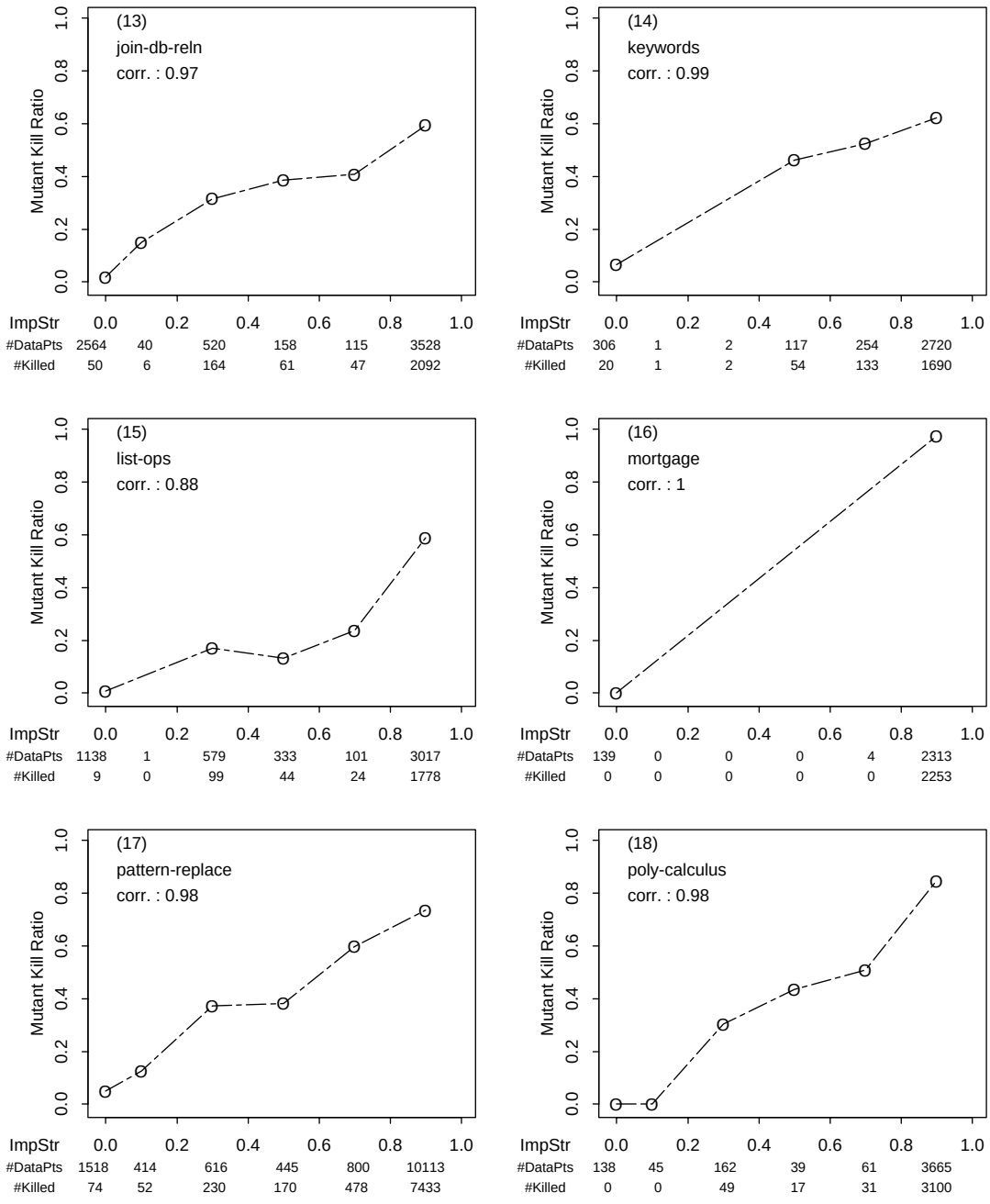


Figure 7.10: Mutant Killing Behavior of Subject Programs 13-18

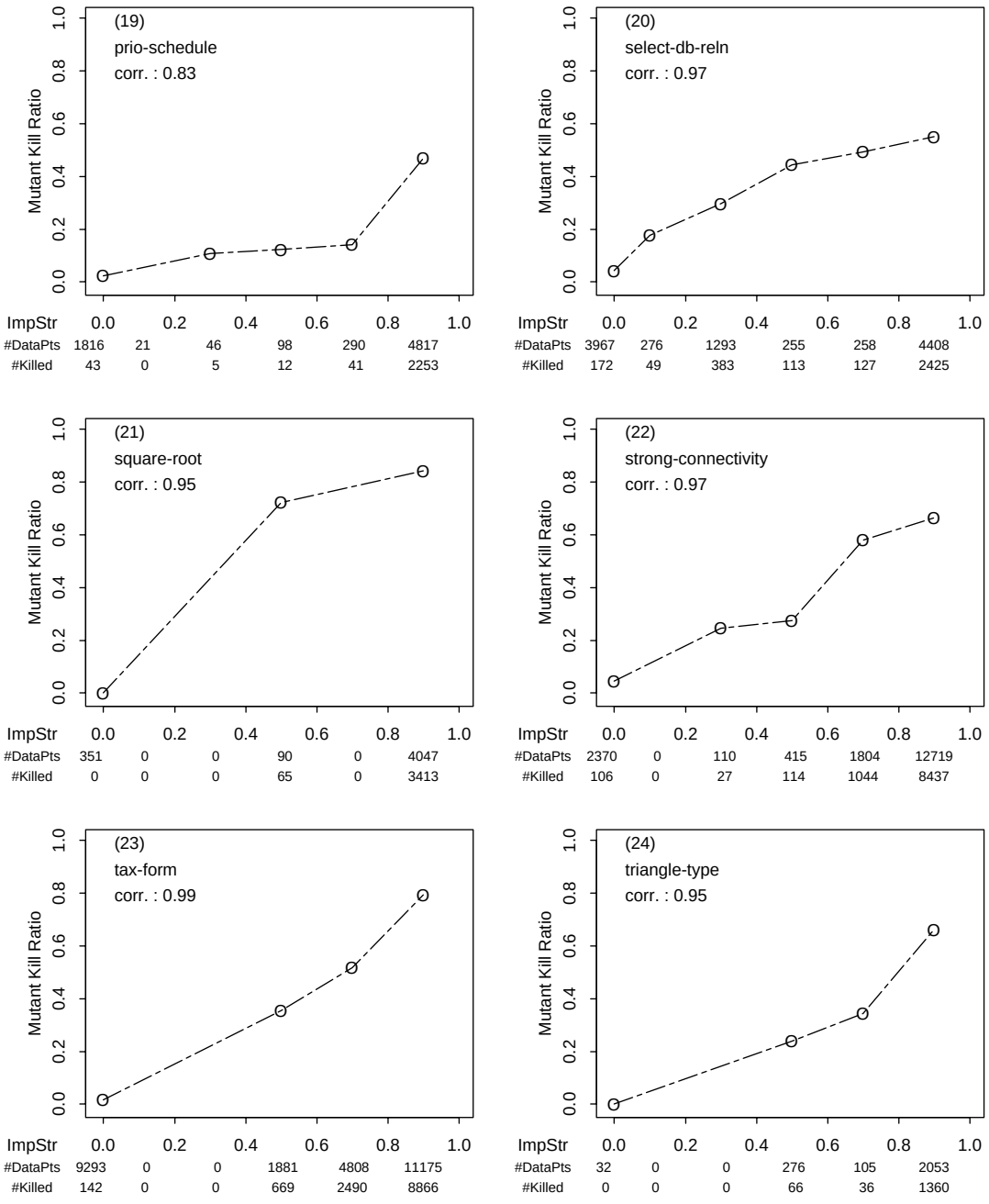


Figure 7.11: Mutant Killing Behavior of Subject Programs 19-24

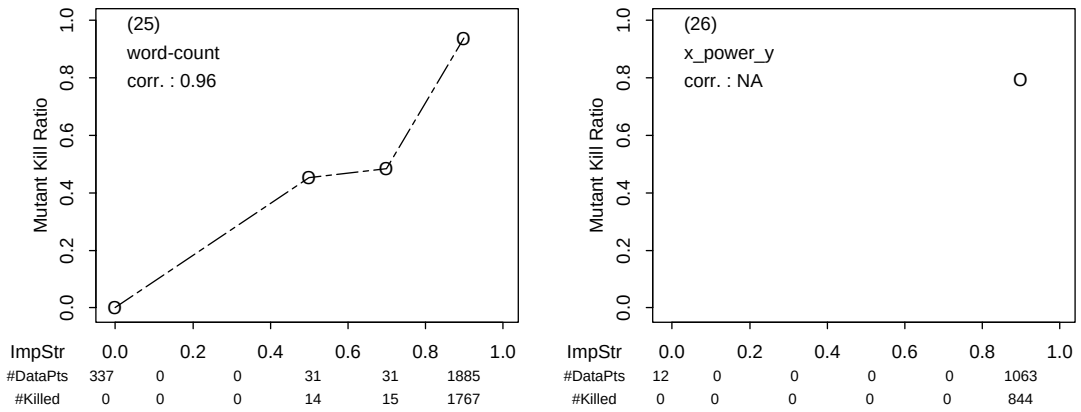


Figure 7.12: Mutant Killing Behavior of Subject Programs 25-26

Strong positive correlation

There is a strong positive correlation (0.88 to 1.0) between the impact strength of a mutation in a test case and the corresponding mutant kill ratio. Usually, the mutant kill ratio increases with the impact strength. An impact strength closer to 0.0 implies smaller chances of killing the mutant and an impact strength closer to 1.0 implies greater chances of killing the mutant. This supports our claim that the impact strength of a mutation in a test case reflects the likelihood that the corresponding mutant will be killed by the test case.

Variety in impact strength distributions

Usually, the number of mutations in the highest impact strength interval is the largest. This is to be expected, since typically a large number of mutations are associated with program entities that always have strong impact the output. However, it is encouraging to observe that there is quite a variety in the distributions of impact strengths. Significant numbers of data points are found in each of the impact strength intervals for subject programs using complex logic such as `pattern-replace` and `bank-promotion`. On the other hand, in subject programs using simple computations such as `eval-expr` and

`x_power_y`, very few mutations have low impact strengths. Section 7.5.6 explains how the variety of impact strength distributions justify the computation of impact strengths as non-boolean values.

Inaccuracy at zero impact strength

Ideally, we would expect the mutant kill ratio to be 0.0 when the impact strength is zero. However, for 18 out of the 26 subject programs, the mutant kill ratio at zero impact strength is a small non-zero value, varying from 0.0079 (`list-ops`) to 0.0699 (`bi-connectivity`). That is, in spite of having zero or no impact, for a very small fraction of the mutations, the corresponding mutants were killed. An investigation of this behavior is presented in in section 7.5.4.

Inaccuracy at high impact strength

The highest mutant kill ratio varies with the subject programs. Specifically, at the highest impact strength interval, the mutant kill ratio varies from 0.34 (`altitude--separation`) to 0.97 (`mortgage`). We investigated this behavior further and the findings are reported in section 7.5.5.

7.5.3 Examining Bias Due to Specific Test Cases

As explained in the experimental procedure (Step 7, page 107), we performed a statistical analysis on the experimental results to study whether or not specific test cases good at detecting errors or at killing mutants had any significant influence on the results. Below we report and interpret the results of this statistical analysis for both the experiments.

As mentioned in the experimental procedure, we used the paired t-test to compare the state error detection vectors of the top third and bottom third of the test cases ranked by their ability to detect state errors. For the `accounting` program, the mean of the differences between the corresponding elements of the vectors was -0.00095. Being a

small negative value, this indicates that in this experiment, on an average, the test cases good at detecting state errors had a slightly poorer rate of error propagation. For the 26 subject programs, the mean of the differences varied from -0.2 to 0.2. It was negative for 12 programs, and positive for 14 programs. The p-value obtained from the t-test quantifies the statistical significance of the observed differences. It is the probability that differences equal to or greater than those observed would occur given that the true mean is 0.0. For example, in the `accounting` program, the p-value was 0.78 indicating a 78% likelihood that the observed differences between the two behaviors would have occurred merely by chance. Due to the lack of sufficient number of observations, the p-value could not be computed for three programs: `x_power_y`, `triangle-type` and `mortgage`. For the other 23 programs, the p-value varied from 0.15 to 0.99. Given such non-negligible p-values[6], we conclude that the observed differences were not statistically significant. Even if the observed differences were significant, it is clear from the mean of differences that the propagation of a specific error in a test case does not depend on the overall ability of the test case to detect state errors.

Similarly, for the second experiment, we used the paired t-test to compare the mutant kill ratio vectors of the top third and bottom third of the test cases ranked by their ability to kill mutants. For the `accounting` program, the mean of the differences between the corresponding elements of the vectors was 0.0131. Being a small positive value, this indicates that in this experiment, on the average, the test cases good at killing mutants had a slightly better rate of error propagation. For the 26 subject programs, it varied from -0.24 to 0.13. The mean of the differences was negative for 17 programs, and positive for 9 programs. For the `accounting` program, the p-value was 0.63 indicating a 63% likelihood that the observed differences between the two behaviors would have occurred merely by chance. Due to the lack of sufficient number of observations, the p-value

[6]Statisticians typically consider p-values less than .05 as negligible.

could not be computed for three programs: `x_power_y`, `eval-expr` and `mortgage`. For the other 23 programs, the p-value varied from 0.10 to 0.98. Once again, given such non-negligible p-values, we conclude that the observed differences were not statistically significant.

The above statistical results indicate that a few test cases good at detecting state errors or at killing mutants did not bias the results in any significant way.

7.5.4 Investigating Inaccuracy at Zero Impact Strengths

Assuming that our definition of impact correctly models reality, one would expect that if an entity instance has zero or no impact on the output, an error in that entity instance would not propagate to the output. However, as observed above, for a very small fraction of entity instances with zero or no impact, the corresponding error propagated to the output. In order to investigate this behavior, we randomly chose from different programs a set of 50 data points demonstrating this behavior and analyzed them in detail. Each of these data points shared a common impact behavior as described below. Let $\mathcal{T}$ be the execution, x be the entity instance and e be the state error corresponding to one such data point. We found that there was an impact path from x to the output on which a *reference impact arc* lead to the definition of a memory location (say *loc*). In the execution $\mathcal{T}$, the memory location *loc* had zero or no impact on the output, and correspondingly the entity instance x had zero or no impact. However, in an altered execution where the error e was introduced in the value of x, the error lead to the definition of memory location loc' instead of *loc*. The location loc' had impact on the output of $\mathcal{T}$, therefore the error in loc' propagated to the output of the altered execution. Thus, even though *loc* has no or zero impact on the output, the error e can propagate to the output by incorrectly defining loc' instead of *loc*. For example, consider an impact path p from a definition of `i` to the output. Let p contain a definition of `a[i]`. In

a specific execution, if the value of `i` is 2, the location `a[2]` gets defined, which does not impact the output of that execution. However, if we alter the execution such that the value of `i` is 1, the location `a[1]` gets defined which impacts the output and thus propagates the error.

Similar investigation was carried out for the mutation experiment as well. We analyzed a sample of 50 data points where the associated mutations had zero or no impact but the corresponding mutants were detected. Each of the data points, without exception, shared the common impact behavior described above.

The results of this investigation points out a deficiency in our handling of the reference impact. Although, the reference impact is given special treatment while computing impact strengths during the forward pass, no such special treatment is given during the backward pass. As described in section 5.4.3, all impact kinds are treated uniformly while back-propagating impact strengths during the second phase of the algorithm. While traversing backwards along the impact arc $\langle x, y\rangle$, the impact strength of x is computed as a function of the impact strengths of y and the impact arc $\langle x, y\rangle$, irrespective of the kind of impact. If y represents a definition of some memory location, and $\langle x, y\rangle$ is a reference impact arc, the impact strength of x should also depend on the impact strengths of the alternate memory locations which would be defined if the errors of x propagated to y. It is not clear to us how to quantitatively account for the dependency of the impact strength of x on the impact strengths of the alternate memory locations. Specifically, we need to come up with a heuristic to estimate the likelihood of propagation of the error of incorrectly defining a location loc' instead of loc. We hope to pursue this issue in a follow up research.

7.5.5 Investigating Inaccuracy at High Impact Strengths

Ideally, when the impact strength of an entity instance x is in the highest interval (0.8,1.0], we would expect that the state error detection ratio is also in the range (0.8,1.0] since most errors in x should propagate to the output. However, it is clear from the above observations that this is generally not the case. Often, the observed error propagation is lower than that predicted by the impact strength. This can be explained by the various compromises made in order to reduce the computational complexity of dynamic impact analysis. However, we also observed that the observed error propagation varied significantly across programs. In programs such as `mortgage`, `word-count` and `eval-expr`, high error detection ratios of 0.89, 0.92 and 0.84 are observed for the (0.8,1.0] impact strength interval. On the other hand, in programs such as `altitude-separation`, `func-zero` and `keywords`, moderate error detection ratios of 0.32, 0.44 and 0.50 are observed for the (0.8,1.0] impact strength interval. This wide variation in the observed error detection at high impact strengths inspired us to probe further and identify the program characteristics primarily responsible for this variety.

We examined a set of 175 data points in which the error did not propagate in spite of 1.0 impact strength. In over 92% of these, the observed discrepancy between the impact strength and the error detection ratio was attributed to program components that are tolerant to errors in control paths. That is, the result produced by an execution of a program component remains correct in spite of an incorrect control path during execution. For example, consider the program `func-zero`. It computes the *zero* of a function in a given interval by using an iterative procedure that terminates when a specific convergence criterion is satisfied. At the end of each iteration, it obtains the next approximation by adding the value of `new-step` to the previous approximation. If the value of `new-step` has an error, the algorithm may still produce the same result by increasing or decreasing the number of iterations. As another example, consider the

`altitude-separation` program. It essentially implements a logic formula consisting of relational and boolean expressions which examines 12 input variables describing the state of a flight and determines whether to keep the same course or move upwards or downwards. An error in the evaluation of a part of the formula may cause a different control path through the program, but may still produce the correct output. In both these examples, the restoration of a correct state may occur long after the first incorrect state was introduced. In order to keep the analysis cost low, our heuristic (for approximating the likelihood of error propagation by following an incorrect branch) does not fully analyze such program components that are tolerant to errors in control paths.

An iteration implementing a many-to-one function is a good example of program components tolerant to errors in control paths. In an effort to capture the importance of the role played by a loop exit in a computation, our heuristic associates 1.0 impact strength with a loop exit. The rationale is that, usually, if a loop exit is not taken either it causes an indefinite iteration or creates an easily detectable incorrect state. We indeed observed these scenarios a large number of times, which justifies the use of this heuristic. However, it is also possible that even if the loop exit is incorrectly not taken, the loop may terminate at a later time without producing any detectable incorrect state. As an example, consider a program component that uses a loop to implement the find operation on a set. While searching for a specific element absent from the set, the iteration can tolerate most errors in the values of the elements or the predicate governing the loop exit. This is because it is unlikely that such an error will change the value of a member to match the value of the element being searched. As another example, consider a `string-match` function that uses a loop to compare two strings for equality. While comparing two strings that differ in several characters, the implementation is very tolerant to errors in exiting the loop. It is very likely that the loop eventually terminates with a correct result after an incorrectly missed loop exit.

Thus, an incorrectly missed loop exit does not always cause a detectable incorrect state. Therefore, instead of assigning 1.0 impact strength to a loop exit, the heuristic should somehow estimate the likelihood of causing a detectable incorrect state by incorrectly missing the loop exit.

As discussed in section 3.4, program components that implement many-to-one functions are often tolerant to errors in control paths. Clearly, the heuristic has been only partially successful in dealing with such components and therefore needs improvement. Designing better heuristics and approaches to handle program components tolerant to errors in control paths is a topic for further research.

7.5.6 Justifying Non-Boolean Impact Strengths

One may argue that the effort in computing impact strengths at such a fine granularity is unnecessary and propose the following relatively simple alternative for computing boolean impact values. Instead of computing impact strengths, one could determine whether entity instances have impact or not by merely following the impact paths in reverse. In comparison, dynamic impact analysis assigns a numeric strength to all entity instances that have impact. The extra cost of computing the impact strengths is justified only if there is a sizable number of entity instances that have impact strengths less than 1.0. The wide variations in the distributions of impact strengths as seen in figures 7.3 through 7.12 provide sufficient evidence to justify the computation of non-boolean impact strengths.

7.5.7 Feasibility of Dynamic Impact Analysis

We showed in section 5.4.6 that the time complexity of the impact analysis of an execution is linear in the execution time. In a claim of linear time complexity, the number of key importance to practitioners is the constant of proportionality, which in this case

is referred to as the *slowdown*. As mentioned in section 7.4, for each program, the cpu time for executing the test suite in the dynamic impact analysis mode was compared with the cpu time for running the test suite in the standard execution mode. We repeated these runs several times and verified that the variations from one measurement to another were within 10%. For the 26 subject programs, the slowdown varied from 2.5 to 14.35 with an average of 6.3. Thus the processing overhead of dynamic impact analysis is very reasonable.

We would have liked to measure the space overhead of dynamic impact analysis. However, since the prototype was implemented in CLOS (Common Lisp Object System) which uses garbage collection, it was not clear how to reliably measure the additional space requirements due to dynamic impact analysis.

7.5.8 Summary of Results

The validation experiments reported in this chapter provide strong evidence in support of the following main results.

1. There is strong positive correlation between the impact strength of an entity instance in an execution and the likelihood that an error in that entity instance would propagate to the output.

2. There is strong positive correlation between the impact strength of a mutation in a test case and the likelihood that the test case would kill the corresponding mutant.

Using statistical analyses, we confirmed that the above two results were not biased by the test cases good at detecting state errors or at killing mutants.

The wide variety of impact strength distributions justifies the computation of impact strength as a non-boolean value.

On the average, the dynamic impact analysis of an execution took only 6.3 times longer than the original execution. This, along with the above results, indicates that we have been fairly successful in our goal of designing a cost-effective technique to estimate error propagation. The following observations indicate the parts of the technique that need improvement.

1. An 1.0 impact strength does not always guarantee error propagation. The presence of program components tolerant to errors in control paths limit the ability of impact strength to predict error propagation in slightly altered executions or fault detection in program mutants. The heuristic for estimating error propagation due to an incorrect branch has been only partially effective and hence it needs to be improved.

2. A zero or no impact does not always guarantee the absence of error propagation. Occasionally, the error of incorrectly defining a memory location loc' instead of loc may propagate even though the correct loc has no or zero impact. In order to produce better accuracy in such situations, the treatment of reference impact needs to be further refined.

Chapter 8

Analyzing Faulty Programs

In the validation experiments reported in chapter 7, we computed impact strengths by analyzing executions of *correct* programs. Then we examined whether or not the impact strengths accurately predicted error propagation in slightly altered executions or fault detection in program mutants. In this chapter, we report an experience study in which the impact strengths are computed by analyzing executions of *faulty* programs. The impact strengths are then related to the observed error propagation in those executions. The major motivations for undertaking this study were as follows:

- to investigate any problems in applying the creation/propagation model of fault detection to a variety of realistic faults,
- to relate the computed impact strengths and the observed error propagation in executions of faulty programs, and
- to better understand the consequences of the approximations made in computing impact strengths.

First, the process of selecting the faulty programs is described. This is followed by a high level description of the study and descriptions of the elements of our analysis of a faulty program. In particular, the problems in identifying an incorrect state and the

methods used for studying impact behaviors are described in detail. Sample analyses of the faulty programs are presented next, followed by a summary of the results and our conclusions from this study..

8.1 Selection of Faulty Programs

In order to serve the goals of this study, it was important to consider a variety of realistic faults in the context of several different application domains. Given that there are no widely-accepted classifications and statistics about faults introduced while developing C programs, it seemed futile to even attempt to define a set of representative fault categories. Instead, we studied several research attempts at fault classification [57, 34, 18, 43] and selected the following fault categories from those that seemed pervasive in the development of C software.

- Incorrect Predicate Expression
- Extra or Missing Conditional Processing
- Incorrect Assignment Expression
- Extra or Missing Assignments
- Faults Related to Iterations
- Faults due to Typographical Errors
- Faults due to Interface Errors
- Faults due to Sequencing Errors

The above list of fault categories should be looked upon as an incomplete set of possibly overlapping fault categories. Some very important fault categories are absent from the above list. In particular, we could not consider faults that cause memory trashing and runtime exceptions primarily due to the limitations of our prototype environment.

Also, we did not consider faults related to memory management, since we believe that they fall outside the scope of dynamic impact analysis. Note that the fault categories enumerated above are interrelated and a fault may often belong to several categories. We also acknowledge that a fault can be classified differently based on different viewpoints.

With the goal of representing each of the above fault categories at least once, we attempted to seed faults in the subject programs used for the validation experiments. The following criteria were used to seed faults:

- the fault should be plausible,
- the fault should not be detectable by all test cases that execute the fault, and
- the fault should not be detectable by a straightforward syntactic analysis.

As a result of this effort, we produced 30 faulty programs derived from 13 base programs. Relevant details about the faulty programs are included along with their analyses. The source code for these faulty programs is available from an anonymous ftp site (see appendix A).

8.2 Elements of the Analysis

The goals of analyzing a faulty program were to apply the creation/propagation model of fault detection to the program, to relate computed impact strengths and the observed error propagation in executions of the faulty program, and to better understand the consequences of the approximations made while computing impact strengths. The following tasks were performed for each faulty program to realize these goals.

Understanding the Program: In order to understand a fault and its behavior, some knowledge about the program containing the fault was necessary. In particular, the input/output specification and the overall structure of the program were carefully studied.

Understanding the Fault: An attempt was made to apply the creation/propagation model of fault detection to the fault in the context of the entire program. The conditions necessary to execute the fault, to create an incorrect state and to propagate the error in the incorrect state to the output were determined. For 4 out of the 30 subject programs, identifying the incorrect state components was subject to the interpretation of what "correct" meant in the specific context. For these four programs, the incorrect behavior was characterized by the fact that one or more state components were inadvertently ignored. This issue is discussed later in section 8.2.1. The entity instances responsible for either the incorrect state or the ignored state components were identified. For brevity, these are referred to as the *affected entity instances*.

Generating Test Cases: For the 26 programs where incorrect state(s) could be clearly identified, in order to study error propagation and the associated impact behavior, test cases were generated to satisfy the fault execution and error creation conditions. The goal of this task was to produce at least 10 test cases that detected the fault and at least 10 test cases that satisfied the fault execution and error creation conditions but did not detect the fault. Probes were inserted into the execution environment to verify that the fault execution and error creations conditions were satisfied. For most faulty programs, TSL [55] test scripts were used to generate the test cases. For some programs, the TSL language was not powerful enough to capture the dependencies among inputs. Hence for these programs, test cases were generated manually. These test cases are available along with the source code of the faulty programs at an anonymous ftp site (see appendix A).

Examining Impact Behavior: For each of the above 26 programs, dynamic impact analysis was carried out for each of the test case executions of the faulty program. Probes were inserted in the execution environment to record the computed impact

strengths of the affected entity instances. These recorded strengths are referred to as the *monitored impact strengths*. The impact behaviors of the detecting and non-detecting test cases were compared and classified using the methods describeΣd in section 8.2.2. For each of the remaining 4 programs, the ignored state components were associated with violation of one or more impact requirements.

8.2.1 Identifying Incorrect State

As mentioned in section 2.2, identifying the incorrect state is often a problem while applying the creation/propagation model of fault detection to real faults. In the literature [67, 72], an incorrect state usually refers to one or more of the following: an incorrect value held by a variable in the data state, an incorrect program counter value (possibly due to an incorrect branch) or an incorrect input state component. For example, consider an iteration with the requirement that the DFS procedure should be called for the first n elements of an array. There is no ordering requirement for these n calls to DFS. Four different versions of a `for` loop implementing the iteration are shown below.

Faulty Version	Correct Version1	Correct Version2	Correct Version3
`for(i=1;i<n;i++)`	`for(i=0;i<n;i++)`	`for(i=1;i<=n;i++)`	`for(i=1;i<=n;i++)`
`DFS(a[i]);`	`DFS(a[i]);`	`DFS(a[i%n]);`	`DFS(a[i-1]);`

The first version is faulty and the other three versions are correct. With respect to the execution of the correct version1, the execution state of the faulty program becomes incorrect after the initialization of loop index `i` to 1 instead of 0. With respect to the execution of the correct version2, the execution state becomes incorrect just before accessing the array element `a[1]` instead of `a[0]`. And with respect to the correct version3, the state becomes incorrect after executing the loop exit condition for the last

time. Thus, it is difficult to identify an incorrect state or determine the first incorrect state in the execution of the faulty version. The only invariant across the three correct versions is that they satisfy the requirement of calling the DFS procedure for each of the first n elements of the array. In the incorrect version, DFS is not invoked for `a[0]`. In this situation, instead of identifying the first incorrect state, it may be more appropriate to characterize the incorrect behavior by saying that "`a[0]` is inadvertently ignored".

8.2.2 Studying Observed Impact Behaviors

Recall that the *affected* entity instances are either those responsible for the incorrect state or those associated with inadvertently ignored state components. In 26 out of the 30 faulty programs, the incorrect state was characterized as the affected entity instances having incorrect values. As mentioned earlier, for these 26 programs, dynamic impact analysis was carried out on test cases that satisfied the fault execution and error creation conditions. In a subject program, if the incorrect state was characterized as the affected entity instances having incorrect values, whether such a test case detected a fault or not depended solely on the propagation of errors in the affected entity instances. Therefore, in order to relate error propagation with impact strengths, the impact behaviors of the detecting and non-detecting test cases were compared to identify the important impact characteristics that differentiated the two sets of test cases. The comparison was based on the kind and strength of impact of the affected entity instances and the output entities on which they had impact. In the remaining 4 programs, the incorrect behavior was characterized by the fact that the affected entity instances were inadvertently ignored. In each of these programs, the presence of a fault could be inferred simply by comparing the observed impact relationships against the expected impact relationships. Both of these situations are discussed below.

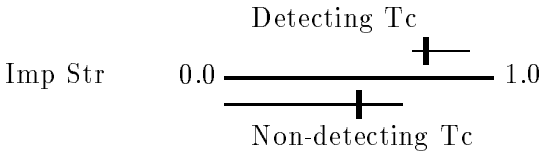
Comparing Impact Characteristics

The impact behaviors of the affected entity instances differed across test cases in one or more of the following: impact kinds, impacted output entities and impact strengths.

Based on our experience, we expected the strength of a data impact to be more reliable than that for a control or a reference impact. Hence, when present, the data impact strength was used as the key discriminating feature between the detecting and non-detecting test cases. In case of the analyses where data impact was absent or insignificant, we examined the average impact strength (see definition 8 on page 69) as the discriminating feature.

For detecting failures by observing the output, some output entities are not as useful as others. For example, in the output of a database query, a constant header string in the output is less useful for detecting a failure as the values representing the result of a query. Therefore, in such cases we measured the strengths of impact of the affected entity instances on specific output entities.

Recall that the *monitored* impact strengths refer to the strengths of impact of the affected entity instances on specific output entities. In our analysis, we provide the following visual aid for comparing the monitored impact strengths in the detecting and non-detecting test cases.



The line in the center represents the scale of impact strength from 0.0 to 1.0. The thin horizontal line above the scale indicate the range of the monitored impact strengths for the detecting test cases and the associated vertical bar indicates the corresponding average impact strength. Similarly, the thin horizontal line and the vertical bar below the scale indicate the range and average of the monitored impact strengths for the

non-detecting test cases. For the purpose of this graphical representation, no impact is treated as zero impact. In 26 out of the 30 faulty programs, where the incorrect states were characterized as the affected entity instances having incorrect values, the impact characteristics of the detecting and non-detecting test cases were compared using this method. Four main types of representative behaviors were observed. Each of them is described with the help of a representative example in section 8.3. Complete analyses of all of the programs are presented in appendix B.

Examining Violation of Dynamic Impact Requirements

Jackson [30] proposed a technique (Aspect) for defining the required dependencies in a functional module and statically analyzing the module to see if the computed static dependencies conform to the required dependencies. A fault is detected when the computed dependencies obtained from analysis do not agree with the required dependencies. In our analyses, we observed that *dynamic impact requirements* could be defined and used more or less the same way. Recall from our earlier discussion (§8.2.1) that an incorrect behavior may be characterized by the fact that a data state component or an input state component is inadvertently ignored. When such a component is an array element or a part of a recursive data structure, it is usually not possible to perform static analysis to accurately determine that the component is ignored on all control paths. However, dynamic impact analysis can accurately determine whether a component is ignored or not in a specific execution. For example, consider the computation of the average of the elements of an array. In this case, one dynamic impact requirement is that *all* elements of the array should impact the resulting average. Depending on the nature of the iteration that computes the average, it may or may not be possible to statically verify that all elements of the array will take part in computing the average. However, using dynamic impact analysis, it is straightforward to determine whether or

not all array elements had data impact on the result. If this dynamic impact requirement is not satisfied in a test execution, we can be certain that a fault is present in the program even if the output is correct.

In 4 out of the 30 faulty programs, the incorrect behavior was characterized by the fact that the affected entity instances were inadvertently ignored. In each of these programs, the presence of the fault results in violation of a dynamic impact requirement. This is illustrated by a representative example in section 8.3. Complete analyses of all of the subject programs are presented in appendix B.

8.3 Sample Analyses

Five sample analyses are presented in this section which are representative of the thirty analyses presented in appendix B. The first four sample analyses represent the following behaviors while comparing the averages and ranges of the monitored impact strengths between the detecting and the non-detecting test cases:

- significant separation between the averages, no overlap between the ranges,
- significant separation between the averages, slight overlap between the ranges,
- significant overlap between the ranges at low impact strengths, and
- significant overlap between the ranges at high impact strengths.

The fifth sample analysis represents the case when a fault is detected by examining the dynamic impact relationships.

The format used for presenting the analyses is described below followed by the sample analyses. At the end of each sample analysis, the observed behavior is characterized and similar observed behaviors of other faulty programs are summarized.

Presentation Format

Each analysis is presented using the following format.

FAULT n: Title Describing the Fault

The fault number *n* is used for the ease of cross-reference.

Application: Name of the Application. Refers to one of the applications described in section A.1.

Context: phrase describing the module containing the fault. Describes relevant context information for understanding the fault.

Fault Description: is a concise description of the fault with respect to the context presented above.

Fault Execution Condition: is the condition necessary for the fault to be executed.

Error Creation Condition: is the condition necessary for an execution of the fault to produce an incorrect state. This assumes that the fault execution conditions are satisfied.

Incorrect State: is a terse description of the incorrect state(s).

Error Propagation Condition: is the condition necessary for the error in the incorrect state to propagate to the output. This assumes that the fault execution and error creation conditions are satisfied. Often, these conditions are too complex to state in a few lines. In such cases, a high level description or a typical scenario is presented.

Impact Analysis: presents the results of the impact analysis of the detecting and non-detecting test cases. For 26 programs, the impact strengths of the affected entity instances are compared between the detecting and non-detecting test cases. For

4 programs, dynamic impact requirements are examined and shown how they can be used for fault detection.

NOTATIONS AND CONVENTIONS

While referring to variables from the faulty programs, we took the liberty of renaming them for better readability. And for better formatting, we use a - (dash) instead of the usual _ (underscore) used for separating words of an identifier in the C language.

Defi(v) denotes a definition of variable v, where i is used as an identifier to distinguish the definition from other definitions of v. Similarly, Usei(v) denotes a use of variable v and Defi-Usej(v) denotes a def-use association from Defi(v) to Usej(v).

Example Illustrating Significant Separation, No Overlap

FAULT 17: Interacting loop initialization and loop exit faults

Application: Database Processing.

Context: performing a join of two relations. A new relation Q is created as a join of relations R_1 and R_2 by performing the following two sequences of steps.

1. First, the headers of R_1 are copied onto Q using a `for` loop. The index of the array filled so far is remembered outside the loop, and used in another `for` loop to copy the headers of R_2.
2. A similar sequence is repeated for adding the attribute values for each joined tuple.

Fault Description: In the second `for` loop (in both sequences), the loop index goes from $1 \cdots n$ rather than $0 \cdots n-1$. As a result, the relation Q has an extra empty attribute column between the attributes of R_1 and R_2.

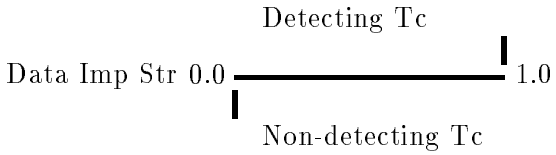
Fault Execution Condition: A join operation is performed on two non-empty relations.

Error Creation Condition: is always satisfied.

Incorrect State: The resulting relation Q has an extra empty attribute column with a blank header and default values.

Error Propagation Condition: A database query is made whose result is sensitive to the values of *all* attributes of a relation is applied to the result relation Q (or a derivative of Q containing the incorrect empty attribute column). If a query uses specific named attributes, the empty attribute column will go undetected.

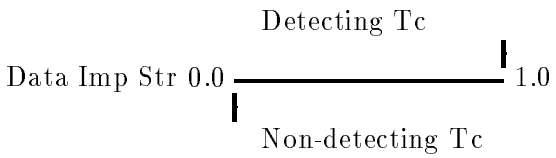
Impact Analysis: The data impact strength of the extra empty attribute is computed as the maximum of the data impact strengths of its blank header and any of the attribute values. The variation in this strength is shown below.



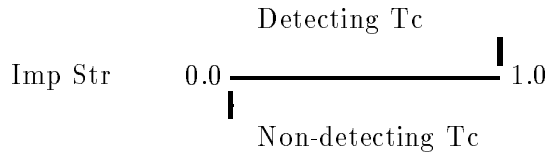
For the detecting test cases, the extra attribute had impact strength 1.0, while for the non-detecting test cases, the extra attribute had no impact.

Fault 17 represents the impact behavior of a group of 15 faulty programs. The monitored impact strengths in the detecting test cases are clearly higher than those in the non-detecting test cases. There is a significant separation between the impact strength averages and the impact strength ranges do not overlap. The impact strength variations for the other 14 faulty programs are summarized below.

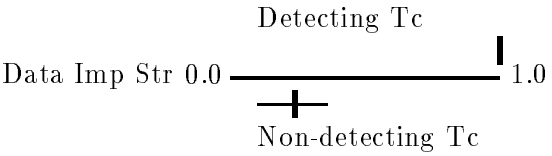
Impact strength variation for faults 1, 3, 4, 12, 15, 20, 23, 26, and 27 :



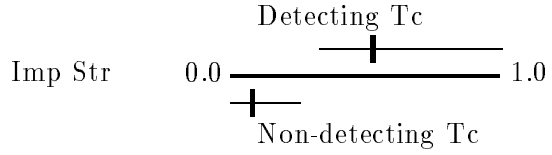
Impact strength variation for faults 8 and 18:



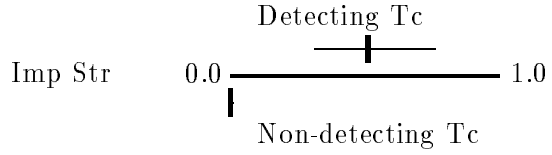
Impact strength variation for fault 2:



Impact strength variation for fault 6:



Impact strength variation for fault 21:



Example Illustrating Significant Separation, Slight Overlap

FAULT 14: Extra `if` statement

Application: Tax Computation.

Context: itemized deductions. If the itemized deduction is greater than the standard deduction, the global variable `excess-deduction` holds the difference.

Fault Description: The assignment to `excess-deduction` is executed only if the taxable income is less than 70% of the adjusted gross income.

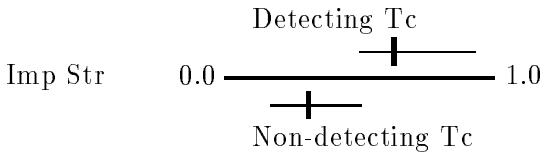
Fault Execution Condition: Input data is valid.

Error Creation Condition: The itemized deduction is greater than the standard deduction but the taxable income is not less than 70% of adjusted gross income.

Incorrect State: The value of `excess-deduction` is incorrect.

Error Propagation Condition: The incorrect value of `excess-deduction` is used in the computation of alternate minimum tax, and the error propagates to the final tax either directly or indirectly by rendering the minimum tax regulation inapplicable.

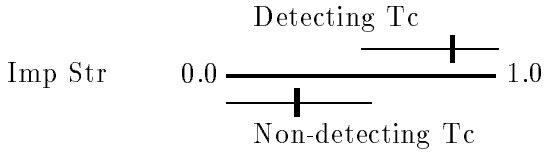
Impact Analysis: The variation in the impact strength of `excess-deduction` is shown below.



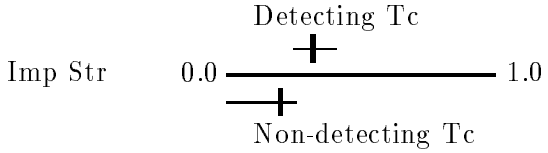
For the detecting test cases, the impact strengths were .84, .67, .5, .5, .67, .75, .84, .92, .75, and .5; and those for the non-detecting test cases were .33, .33, .33, .17, .33, .5, .17, .26, .5, and .2.

Fault 14 represents the impact behavior of a group of 3 faulty programs. The monitored impact strengths in the detecting test cases are generally higher than and occasionally equal to those in the non-detecting test cases. That is, over a *narrow* range of impact strengths, both detecting and non-detecting test cases are present. This interval of overlap is at the lower end of the range of impact strengths in the detecting test cases and at the upper end of the range of impact strengths in the non-detecting test cases. For example, in the above sample analysis, three detecting and two non-detecting test cases have impact strength 0.5. All other detecting test cases have impact strength greater than 0.5 and all other non-detecting test cases have impact strength lower than 0.5. As expected, at 0.5 impact strength, error-propagation occurs in some test cases but not in others. The impact strength variations for the other 2 faulty programs are summarized below.

Impact strength variation for fault 10:



Impact strength variation for fault 22:



Example Illustrating Significant Overlap at Low Impact Strengths

FAULT 28: Incorrect argument

Application: Market Research.

Context: processing account recommendation. In order to determine the account type most appropriate for the customer, among other things, the number of transactions (`num-transactions`) executed by the customer is required as an argument.

Fault Description: The number of overdraft transactions is passed instead of the number of transactions.

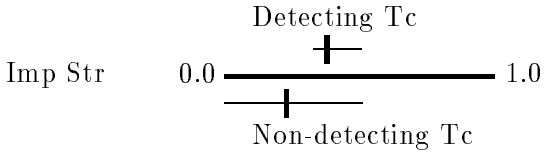
Fault Execution Condition: is always satisfied.

Error Creation Condition: All transactions are not overdraft transactions.

Incorrect State: The `num-transactions` argument has an incorrect value.

Error Propagation Condition: The customer has not paid high overdraft fees or high transaction fees, the number of transactions is greater than 10 and the number of overdraft transactions is less than 5.

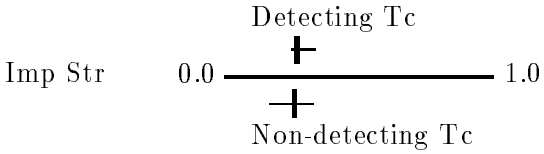
Impact Analysis: The variation in the impact strength of the incorrect `num-transactions` argument is shown below.



For the detecting test cases, the impact strengths were 0.5, .34, 0.5, .34, .34, .34, .33, .5, .33, and .33. For the non-detecting test cases, the impact strengths in nine cases were .17, .28, .5, .26, .28, 0.0, .17, .26, and .33, and in one case the `num-transactions` argument had no impact.

Fault 28 represents the impact behavior of a group of 2 faulty programs. In this case, the affected entity instances generally have low to medium impact strengths in all test cases. As expected, in some test executions the error is propagated resulting in fault detection and in some test cases the error is not propagated. Therefore, there is a considerable overlap between the ranges of impact strengths of the detecting and non-detecting test cases. The impact strength comparison for the other faulty program is summarized below.

Impact strength variation for fault 24:



Example Illustrating Significant Overlap at High Impact Strengths

FAULT 25: Incorrect return value semantics

Application: String Processing (pattern matching and substitution.

Context: processing escape sequence. When the escape character `@` is the last character in the input regular expression or the replacement text, it is to be treated as a regular character. The procedure processing escape sequences is expected to correctly implement such a situation and return the `@` character.

Fault Description: When the `@` is the last character in the input regular expression or the replacement text, the procedure processing escape sequences returns a null character instead of the `@` character.

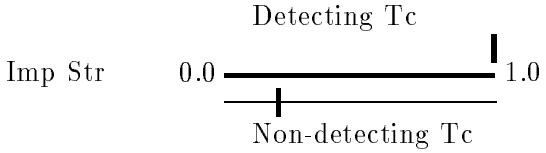
Fault Execution Condition: At least one `@` character is present in either the input regular expression or the replacement text.

Error Creation Condition: An `@` character is present as the last character in the input regular expression or the replacement text.

Incorrect State: The return value of the procedure processing escape sequences is incorrect in the above case. The incorrectly returned null character prematurely terminates the internal representation of either the input regular expression or the replacement text.

Error Propagation Condition: If the regular expression is incorrect, the error is propagated when there is an incorrect match or incorrect mismatch. If the replacement text is incorrect, the error is propagated when there is a match followed by the corresponding substitution.

Impact Analysis: The variation in the impact strength of the incorrect return value is shown below.

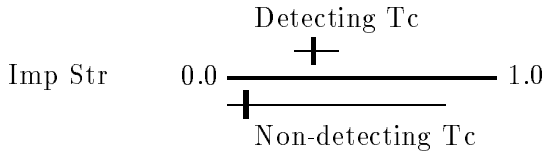


For the detecting test cases, the impact strengths were 1.0. For eight of the non-detecting test cases, the incorrect return value had no impact. However, in the remaining two non-detecting test cases, there was at least one character sequence in the subject that matched the pattern *only* upto the `@` character. Therefore, the incorrect state had impact on the `false` outcome of the match. As we mentioned before (section 7.5.5), when the expected outcome of a string matching function is

`false`, it is tolerant to errors in the control paths or in the strings being compared.

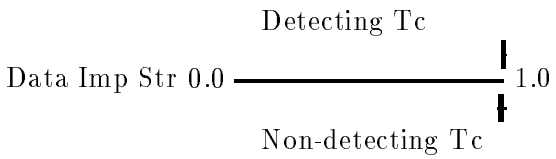
Fault 25 represents the impact behavior of a group of 5 faulty programs. The monitored impact strengths of the affected entity instances are unusually high for one or more of the non-detecting test cases. For 3 out of these 5 programs, this was associated with specific program components that are tolerant to errors in control paths. With each such behavior, we identify and describe the specific reasons for the observed error tolerance. For the other two programs, this behavior was caused due to the limitations of our prototype. In these cases, we identify and report the specific limitation of our prototype responsible for this behavior. The impact strength variations and the reasons for the observed high impact strength in some of the non-detecting test cases are summarized below for the other 4 faulty programs.

Impact strength variation for fault 11 :



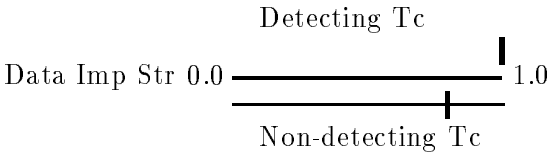
For this program computing the strongly connected components (SCCs) of a directed graph, unusually high monitored impact strengths were observed for 3 out of the 50 non-detecting test cases. In each of these three test cases, every node of the directed input graph (with n nodes) had an outdegree of either $(n-1)$ or 0. When the input digraph has such a property, the scc algorithm is very tolerant to the errors in control paths caused by the fault in this program.

Impact strength variation for fault 30:



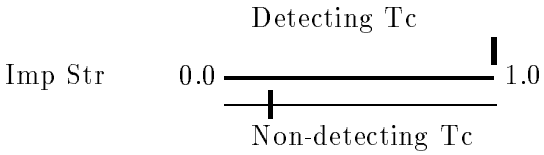
In this faulty program, while loading the input graph, the start and end nodes of the first edge are read incorrectly. In all of the non-detecting test cases, the first edge was of the form $(0, n)$. In such a situation, the fault causes an inversion of the edge from $(0, n)$ to $(n, 0)$, which is a no-operation for an undirected graph. Therefore, even though this error affects the control paths of the execution, the final output remains the same.

Impact strength variation for fault 5:



For this fault, all of the non-detecting test cases with 1.0 impact strength involved precision loss. Our prototype currently uses a different precision floating point arithmetic than that supported by the system C library and hence the impact strengths computed did not accurately reflect the loss of precision.

Impact strength variation for fault 19:



For this program, out of the 23 non-detecting test cases, in 4 test cases the monitored impact strength was 1.0. Out of these, 3 test cases involved loss of precision. As mentioned above, due to the limitations of our prototype, the computation of impact strengths does not accurately reflect the loss of precision. In one test case, error cancellation occurred when integrating a polynomial Q from point v to v. That is, in an operation $Q(v) - Q(v)$, an error in a coefficient of Q got canceled.

Example Illustrating Violation of Dynamic Impact Requirements

FAULT 7: Incorrect Predicate

Application: Graph Algorithms.

Context: performing the depth first search. In order to perform the depth first search, a high level `while` loop scans through all the nodes in the graph to ensure that every node is visited at least once.

Fault Description: Instead of the correct expression `(++last-new-node < num-nodes)`, the incorrect expression `(last-new-node++ < num-nodes - 1)` is used as the predicate of the `while` loop. As a result, node 0 is ignored in this high level scan.

Fault Execution Condition: is always satisfied.

Error Creation Condition: The graph is non-empty.

Incorrect Behavior: Node 0 is ignored in the high level scan.

Error Propagation Condition: There are no edges incident on node 0. (If node 0 was connected to at least one other node, it will be reached in the depth first search and the fault will not be detected).

Impact Analysis: A dynamic impact requirement of the depth-first-search algorithm is that, for valid input graphs, all nodes of the graph should impact the result. For the detecting test cases, node 0 did not impact the output, hence this requirement was violated. For the non-detecting test cases, this requirement was satisfied.

Fault 7 represents the impact behavior of a group of 4 faulty programs. In these programs, the incorrect state is characterized as one or more ignored state components and the entity instances associated with the ignored components do not have impact on the output. In such cases, the violation of a dynamic impact requirement indicates the

presence of a fault. Using fault 7 as an example, we illustrated above how examining impact requirements can be used to detect the presence of faults. Similar analyses for the other three faults (9, 13 and 16) are presented in appendix B.

8.4 Summary and Observations

In 23 out of the 30 faulty programs analyzed, the impact characteristics were closely related to error propagation.

- In 18 programs, the monitored impact strengths in the detecting test cases were generally higher than those in the non-detecting test cases. There was significant separation between the average impact strengths of the two sets. Out of these 18, in 15 cases, there was no overlap between the ranges of the impact strengths in the two sets, while in 3 cases, there was very slight overlap.
- In 4 programs, the incorrect state was characterized as one or more ignored state components. In these faulty programs, it was shown how examining dynamic impact requirements can be used to detect the presence of faults.
- In one faulty program, there were no detecting test cases. That is, the error in the incorrect state did not propagate to the output for any test case. As expected, the affected entity instances did not impact the output in any of the test cases.

In 2 out of the 30 faulty programs, the monitored impact strengths were generally low to medium for all test cases and there was considerable overlap between the ranges of impact strengths for the detecting and non-detecting test cases. Thus the impact strengths correctly predicted the small likelihood of error propagation.

In the remaining 5 faulty programs, the monitored impact strengths were unusually high for one or more of the non-detecting test cases. For 3 out of these 5 programs, this behavior was associated with specific program components tolerant to errors in control

paths. For the other 2 programs, this behavior was caused due to a specific limitation of our prototype. The prototype uses a different precision floating point arithmetic than that supported by the system C library and hence the impact strengths computed did not accurately reflect the loss of precision.

In all of the programs, there was only one test case demonstrating the failure of error propagation due to canceling errors (two or more errors interacting in an operation to cancel each other).

We acknowledge that since this is a small scale study of only 30 faults, it is not appropriate to generalize from its results. Nevertheless, we list below our interpretations of the results of this study.

- While applying the error creation/propagation model to a realistic fault, one often faces the difficulty of identifying the incorrect state caused by the fault. Occasionally, instead of identifying the first incorrect state caused by the execution of a fault, we found it more useful to characterize the incorrect behavior by identifying the state components ignored in the faulty execution.
- If a program execution is sensitive to errors in control paths, impact strengths computed by analyzing the execution are very good indicators of error propagation in that execution.
- In programs involving floating point computations, error propagation often fails due to the loss of precision.
- It is plausible that there may not exist a test case in which at least one instance of a program entity Y has high impact on the output. This implies that the output of the program is very insensitive to Y. Therefore, faults leading to an incorrect value of Y cannot be detected easily. In order to validate the parts of the program computing Y, it may be necessary for the tester to probe internal states that are sensitive to Y.

Chapter 9

Potential Applications

This chapter describes potential applications of dynamic impact analysis. We first describe how dynamic impact analysis can be applied to code-based testing strategies and discuss some of the issues involved. We then briefly describe the application of dynamic impact analysis for computing a dynamic program slice.

9.1 Mutation Testing

Using dynamic impact analysis, weak mutation testing [27] can become a formidable competitor to strong mutation testing. In order to save the cost of strong mutation testing, the weak mutation approach [27] does not require the satisfaction of the error propagation condition and instead relies on the *weak mutation hypothesis*, which states that the error propagation condition will be satisfied when the fault execution and the error creation conditions are satisfied. Our experience has shown that this hypothesis is not based on any sound intuition or expectation of reality. In a study reported by Marick [41], the weak mutation hypothesis was true only half the time even in the context of individual procedures. Also, the results of the mutant killing experiment reported in chapter 7 provide strong evidence against the weak mutation hypothesis. In order to

relax this assumption, dynamic impact analysis can be used to assure that mutations are exercised with a high probability of satisfying the error propagation condition. The validation results reported in this thesis indicate that, in general, the higher the impact strength associated with a *weak-killed* mutant in a test case, the greater the chances of that mutant being *strong-killed* in that execution.

Interestingly, dynamic impact analysis can also be used to reduce the cost of strong mutation analysis [13]. One of the factors contributing to the cost is the practice of executing a test case on *all* remaining live mutants. Test case executions that do not kill mutants are wasteful. In order to reduce the number of wasteful executions, Duncan and Robson [15] proposed an ordering scheme which orders the test cases and mutants based on the control flow and the types of mutants. In a recent paper, Weiss and Fleyshgakker [76] proposed a serial algorithm for strong mutation analysis which ensures that a (test case, mutant) pair is executed only if the corresponding mutation is weak-killed by the test case. The mutation impact strengths computed by dynamic impact analysis provides useful information which can be used to avoid executions in which a weak-killed mutation has a very low probability of affecting the output. The validation results indicate that, given the weak-killed mutations with different impact strengths during a test case execution, the test case is more likely to kill those with higher strengths than those with lower strengths. In order to minimize wasteful executions, one can exploit this observation to design the order in which the (test case, mutant) pairs are executed. For example, it makes sense to run a test case first on the live mutants that have impact strength 1.0 or near 1.0 for that test case. Most of these mutants should get killed by the test case unless their impact was via program components tolerant to errors in control paths. Similarly, there is little point in running a test case on mutants that have no impact or zero impact on the output of that test case. Further research is necessary to determine how to use the intermediate values of impact strength and the

impact kind information in minimizing wasteful executions.

9.2 Syntactic Coverage-based Testing

Recall that a typical syntactic coverage-based testing strategy specifies a set of syntactic components in the program that need to be exercised by a test suite for adequate testing. The requirement of exercising a syntactic component is satisfied if it is executed at least once by some test case in the test suite. This causes the problem that if the execution of the syntactic component did not affect the output, then the faults exercised by executing the component can go undetected. To avoid this problem, we can modify the requirements as follows: a component is considered exercised when it is executed at least once with non-zero impact strength. This will disqualify component executions that had either no impact or zero impact. Duesterwald, Gupta and Soffa propose an approximate solution to disqualify component executions that had no impact. As discussed in section 5.5, our approach is more accurate in disqualifying component executions with no impact and additionally we also disqualify components with zero impact. The results from chapter 7 indicate that it is not uncommon to have a sizable number of components with zero impact.

We can take this approach one step further and require that the exercised components have *high* impact strength. This extension involves two problems. First, it is not straightforward to quantify *high* impact strength in a program independent way. The other problem has to do with helping the tester in satisfying the impact strength requirements. More research is needed in order to come up with heuristics to solve these two problems. We believe that the fault detection capability of a syntactic coverage-based testing strategy will improve by imposing the impact strength requirements. We hope to substantiate this claim in subsequent research.

9.3 Dynamic Program Slicing

In our framework, computing a dynamic program slice (§2.8, [2, 35]) would require processing the execution trace in reverse order and computing a transitive closure with respect to the impact predecessor relation starting at the entity instances corresponding to definitions of specified variables. One can obtain a smaller and more accurate slice if, while computing the transitive closure, the search is pruned when impact arcs with zero impact strength are encountered. While conducting the experiments described in chapter 7, we observed that a sizable number of impact arcs have zero impact strength.

Chapter 10

Conclusion and Future Research Directions

In this dissertation, the technique of dynamic impact analysis was proposed for analyzing error propagation in program executions. The notions of a program impact graph and an execution impact graph were introduced to provide the infrastructure necessary for supporting dynamic impact analysis. The notion of impact strength was defined as a quantitative measure of the error sensitivity of an impact. A cost-effective algorithm to analyze impact relationships in an execution and compute the impact strengths was presented. A prototype implementation that demonstrates the feasibility of dynamic impact analysis was outlined. The experiments conducted to validate the computation of impact strengths were described. An experience study undertaken to relate impact strengths to error propagation in executions of faulty programs was described. Finally, potential applications of dynamic impact analysis in the areas of mutation testing, syntactic coverage-based testing and dynamic program slicing were discussed.

The empirical results provide evidence indicating a strong positive correlation between impact strength and error propagation. The time complexity of dynamic impact

analysis is shown to be linear with respect to the original execution time, and experimental measurements indicate that the constant of proportionality is a small number ranging from 2.5 to 14.5. Together, these results indicate that we have been fairly successful in our goal of designing a cost-effective technique to estimate error propagation. However, in order to reap the potential benefits of the technique, the accuracy of the estimate needs to be improved significantly. In particular, better heuristics are needed for handling reference impact and program components tolerant to errors in control paths. The most important research issues emerging out of this dissertation are listed below.

- A heuristic to estimate in a cost-effective way, the potential control impact of a decision branch due to the fact that it *avoided* certain state changes.

  The usefulness of dynamic impact analysis can be significantly enhanced by designing such a cost-effective heuristic. A related issue is the detection and handling of program components that are tolerant to errors in control paths. As demonstrated in the results, such program components cause overestimation of error propagation.

- An assignment of weights to different kinds of impacts based on their relative importance in error propagation.

  It is possible that such weights may depend on the kind of computation being performed by the program. It would be interesting and useful to study the joint correlation between error propagation, impact strength and impact kind.

- A heuristic to estimate in a cost-effective way the likelihood of propagation of the error of incorrectly defining a location loc' instead of loc.

  This requires an in depth study of the reference impact in different programming contexts. In a sense, the results of our experiments strongly agree with our decision

to treat the reference impact separately from the data impact.

- Refinement of the notion of an error set and the proposed error classes.

  For better accuracy, it may be necessary to refine the notion of an error set and add new application specific error classes. On a related topic, the sensitivity of impact strength to specific error classes needs to be empirically evaluated.

- An algorithm should be designed to determine the order in which the (test case, mutant) pairs should be evaluated in strong mutation testing, based on the impact strengths of corresponding mutations.

  We believe that rather than running a test case on all live mutants or running a live mutant for all available test cases, it may be more productive to schedule the running of the (test case, mutant) pairs based on the impact strength of the corresponding mutation in that test case.

Some of the other related research issues are: alternate ways of propagating and assimilating impact strengths, techniques to compute most of the impact information statically so as to minimize the dynamic overhead, algorithms to partially process very large execution traces, and approaches to model the impact of the `goto` statement.

The results and ideas presented in this thesis have already inspired several research efforts in other software testing topics. For example, we are examining the possibility of using impact strengths to select the best test case among those with almost identical syntactic coverage. If successful, this would greatly help the problem of removing redundant test cases from a large test suite. In another endeavor, the concept of impact strength is being evaluated for use in generating error sensitive test cases. We hope that the research effort described in this dissertation continues to inspire new ideas in software debugging, testing and maintenance.

Appendix A

Experimental Data

We first describe the 26 subject programs used in the validation experiments (Chapter 7). This is followed by instructions for accessing the experimental data from an anonymous ftp site.

A.1 Subject Programs

The 26 subject programs used in the validation experiments are briefly described below grouped by the applications. Along with a brief description for each program, information is provided about the source from where the program was obtained.

Accounting: The `accounting` program was derived from a basic home accounting application available as a public domain software on a personal computer. It supports deposits, expenses by categories, monthly payments, and monthly interest accrual on the account balance. A simple query interface supports inquiries about the balance, and various expense categories.

Aircraft Control: Leveson et al. [40] describe a formal specification technique that uses a modified statechart notation to specify TCAS II, an aircraft collision avoidance system. The `altitude-separation` program is a C translation of the selected functionality from the above specification. It essentially implements a logic formula consisting of relational and boolean expressions which examines 12 input variables describing the state of a flight and determines whether to keep the same course or move upwards or downwards.

Calculus: The `poly-calculus` program is a set of functions supporting integration and differentiation of polynomials. The program takes an input polynomial P and based on the operation requested, either differentiates P at a specified point or integrates it over a specified interval. It was developed as an undergraduate class assignment.

Compilers: The `eval-expr` program implements a small recursive decent parser for parsing arithmetic expressions with constant operands and evaluates the expression. The program was implemented as an undergraduate course assignment.

Database Processing: The three programs `aggregate-db-reln`, `join-db-reln` and `select-db-reln` were derived from a relational database program developed as a graduate class project. It supports loading relations into memory, indexing the relations for efficiency, performing join, select or aggregate operations on the relations to create new relations, and printing the relations. The operations and queries are issued using a simple ascii interface.

Graph Algorithms: The three programs `depth-first-search`, `bi-connectivity` and `strong-connectivity` implement a set of functions supporting graph algorithms based on the depth-first-search. They are C translations of algorithms presented in an algorithms book [4]. These support loading of directed or undirected graphs, carrying out depth first search while printing the nodes in a depth first order and

computing bi-connected or strongly-connected components.

Investment-related Computation: The `mortgage` program was derived from a public domain spreadsheet software running on Unix. It implements a set of investment-related functions and supports computing monthly payment for a mortgage, the future value of an investment and the present value of a future dollar amount.

Market Research: The `bank-promotion` program classifies individuals based on some information about their financial status. It takes as input the information about a bank's customer and the vital statistics about his account to determine the target groups that he/she belongs to for various sales promotions. We developed this program based on the information from a management consultant of a bank.

Numeric Algorithms: The three programs `func-zero`, `square-root` and `determinant` represent the numeric processing domain. The `func-zero` program takes a function and an interval and attempts to find a point in the interval where the function value is zero. It is a public domain program available on *netlib* (anonymous ftp sites *netlibornl.gov* or *research.att.com*). The `square-root` program computes the square root of a floating point number with the requested precision. It is a C translation of the program used by Rapps and Weyuker [64], which in turn was translated from the Wensleysqroot program used by Boyer, Elspas and Levitt [9]. The `determinant` program is a C translation of the revised version of the algorithm 44 in the *Collected Algorithms from CACM* [1]. It computes the determinant of a square matrix of floating point numbers.

Operating System: The `prio-schedule` program is a driver for testing the process scheduling scheme implemented in a small operating system used for teaching an undergraduate class. The program implements priority scheduling with three priorities low, medium and high. A sequence of input commands specify the events relevant to process scheduling such as process entry, blocking, unblocking, priority

update, time slot expiration, and process exit.

Other Programs: The `list-ops` program is a driver for testing the insert, delete and find operations in a module implementing doubly linked lists. It was implemented as an undergraduate class assignment. The `binary-search` program is a toy program implementing binary search in a small array. It was derived from an example program used by Bob Horgan of AT&T Bell Laboratories in a research discussion about dataflow testing. The `x_power_y` program is a toy program computing X^Y for integer values of X and Y. It is a C translation of the program used by Rapps and Weyuker [65]. The `triangle-type` program classifies an input triangle as equilateral, isosceles, scalene or not a triangle. It is a C translation of a similar program distributed with the Mothra mutation testing tool [12].

Statistics: The two programs `chi-square-test` and `info-measure` represent the domain of statistical computations. The `chi-square-test` program implements the Pearson's chi-square test for a 2-way contingency table. The `info-measure` program computes Kullback's information measure for a 2-way contingency table. Both programs were obtained from *netlib* (accessible via email to *netlib@ornl.gov* or from the anonymous ftp site *research.att.com*).

String Processing: The `word-count`, `keywords` and `pattern-replace` programs represented the string processing domain. The `word-count` program counts the number of lines, characters in each input file, determines the maximum number of columns used and the maximum print width needed for printing the file without truncation or wrap-around. It was developed by Marcus Yoo of Siemens Corporate Research, Inc. The `keywords` program matches tokens in the input stream with a set of keywords and outputs the frequency of occurrence of each keyword. This program was obtained from a friend with no information about its primary author. The `pattern-replace` program takes a pattern in the form of a regular

expression, matches the pattern on input text and replaces the matched text with the replacement text. This program was a C translation of the pascal program used by Weyuker [78]. The pascal program was derived from a suite of programs by Kernighan and Plauger [33].

Tax Computation: The `tax-form` program implements a simplified 1040 tax form. It validates the input data and computes the tax, taking into account the itemized deductions, IRA deduction, and alternate minimum tax regulation. It was derived from a public domain application on personal computers.

A.2 Data Available on Anonymous Ftp-site

At the anonymous ftp site *cs.nyu.edu*, the file tarak.tar.Z in the /pub/theses directory contains the following:

1. a postscript version of this thesis,
2. the source code for all of the 26 programs and the associated test suites used in the validation experiments reported in chapter 7, and
3. the source code for all of the 30 faulty programs and the associated test cases used in the experience study reported in chapter 8.

Appendix B

Analyses of Faulty Programs

As mentioned in chapter 8, thirty faulty programs were analyzed using the procedure outlined in section 8.2. These analyses are presented here. The format and the notations used for the presentation were described section 8.3. The analyses are grouped according to the major fault category represented by the fault in the program.

B.1 Incorrect Assignment Expression

FAULT 1: Missing part of an expression

Application: Tax Computation.

Context: alternate minimum tax computation. If the computed tax is less than the overall alternate minimum tax, then the final tax is computed by applying the `alt--min-tax-rate` to the sum of `non-taxable-interest` and `excess-deductions`.

Fault Description: The `alt-min-tax-rate` is applied only to the the the `non-taxable--interest`.

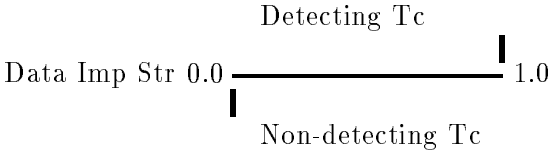
Fault Execution Condition: Input data is valid and the computed tax is less than the overall alternate minimum tax.

Error Creation Condition: The value of `excess-deductions` is non-zero.

Incorrect State: The tax (Def1(x)) computed by applying the `alt-min-tax-rate` is incorrect.

Error Propagation Condition: The erroneous tax value is not superseded by the overall alternate minimum tax.

Impact Analysis: The variation in the data impact strength of Def1(x) is shown below.



For the detecting test cases, the impact strengths of Def1(x) were 1.0, while for the non-detecting test cases, Def1(x) had either no impact or zero impact. It was interesting to observe that a \$0.001 change in the value of mortgage interest deduction was sufficient to change a non-detecting test case to a detecting test case. And correspondingly, the impact strength of Def1(x) changed from {no data impact, 0.5 control impact} for the non-detecting test case to {1.0 data impact, 1.0 control impact} for the detecting test case.

FAULT 2: Extra term introducing 1% error in the value of an operand

Application: Calculus.

Context: performing integration. First, an indefinite integral Q is obtained by integrating the input polynomial P. The required finite integral is evaluated over the interval (`from-x`, `to-x`] by computing Q(`from-x`) $-$ Q(`to-x`).

Fault Description: While calling the polynomial evaluation routine on `from-x`, the actual argument was `1.01 * from-x`. (This was a real fault found in the program. It was probably left behind after debugging or testing).

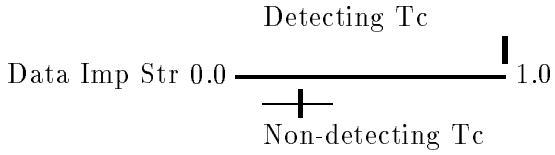
Fault Execution Condition: The input polynomial is valid and the integration operation is performed.

Error Creation Condition: The value of `from-x` is non-zero.

Incorrect State: The formal argument `x` of the second polynomial evaluation has a 1% error.

Error Propagation Condition: The error propagates through several interacting impact paths during the second evaluation of Q and the subtraction $Q(\texttt{from-x}) - Q(\texttt{to-x})$.

Impact Analysis: The variation in the impact strength of `x` in the second polynomial evaluation is shown below.



For the detecting test cases, the impact strengths of `x` were 1.0. For the non-detecting test cases, the impact strengths were .24, 0.1, .35, .29, 0.1, .26, .13, .29, .32, and .33.

FAULT 3: Missing part of an expression, incorrect in specific context

Application: Accounting.

Context: monthly interest computation. The variable `last-balance` keeps track of the account balance at the start of a month and `balance` keeps track of the current balance. Let x denote the variable `last-balance`. At the end of every month, interest is computed by applying the interest rate to the average of `last-balance` and `balance`. During initialization, `last-balance` is set to initial balance (Def1(x)). After all monthly processing is complete, `last-balance` is set to the current balance (Def2(x)).

Fault Description: During the monthly interest computation, the interest rate is applied to `last-balance` (Use1(x)) rather than the average of `last-balance` and `balance`.

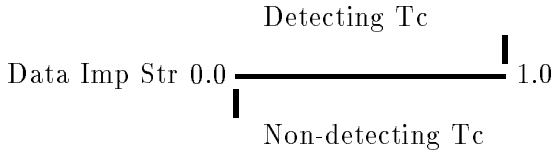
Fault Execution Condition: At least one `end-of-month` command is processed. Depending on whether it is a first month or not it results in the execution of either Def1-Use1(x) or Def2-Use1(x).

Error Creation Condition: The variables `last-balance` and `balance` have different values at the end of a month.

Incorrect State: An incorrect amount participates in the interest computation.

Error Propagation Condition: The interest rate is non-zero and a report inquiry is made to check the balance after an incorrect state.

Impact Analysis: The error propagation condition is satisfied iff one or both of the two def-use associations (Def1-Use1(x) or Def2-Use1(x)) have data impact on the output. The variation in the maximum of the data impact strengths of these def-use associations is shown below.



For the detecting test cases, at least one of the two def-use associations had impact strength 1.0 and for the non-detecting test cases, both du-associations had either no impact or zero impact.

FAULT 4: Incomplete Update

Application: Accounting.

Context: fee for monthly transactions. As per a new requirement, a fee should be charged for monthly transactions. In the procedure updating monthly transactions, there

are two places where the `amount` (x) of the transaction is updated: once when a new monthly transaction is added (Def1(x)) and once when the amount of a previous monthly transaction is changed (Def2(x)). Both places should be updated to take care of this new requirement.

Fault Description: The transaction fee is added to the assignment expression computing transaction amount at Def1(x) but not at Def2(x).

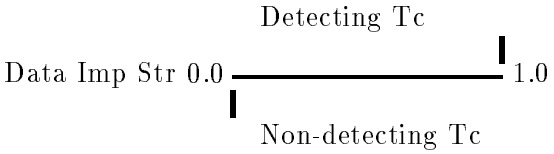
Fault Execution Condition: A monthly transaction is added and then modified, executing the incorrect expression at Def2(x).

Error Creation Condition: is always satisfied.

Incorrect State: The amount of a monthly transaction is incorrect.

Error Propagation Condition: A monthly transaction containing an incorrect amount is executed at least once, thus affecting the balance. The resulting balance is checked by a report inquiry.

Impact Analysis: The variation in the maximum of the data impact strengths of various instances of Def2(x) is shown below.



For the detecting test cases, at least one instance of Def2(x) had impact strength 1.0, while for the non-detecting test cases, the instances of Def2(x) had either no impact or zero impact.

FAULT 5: Off-by-one fault in a floating point expression

Application: Investment-related Computation.

Context: future value computation. If r is the periodic interest rate, n is the number of periods and P is the present value of an investment, then $P * ((1 + r)^n - 1)/r$ gives the future value of the investment.

Fault Description: The expression $P * ((1 + r)^n)/r$ is used instead of the above expression.

Fault Execution Condition: The future value function is invoked.

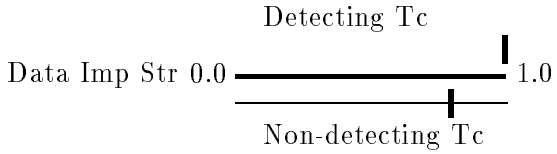
Error Creation Condition: is always satisfied.

Incorrect State: The entity instance x representing the value multiplied with P has an incorrect value.

Error Propagation Condition: The incorrect value propagates through the multiplication and division operations. That is, initial investment P and the periodic rate r should be non-zero, and $(1 + r)^n$ should not be such that due to *precision loss* one of the following is true.

- $(1 + r)^n \simeq (1 + r)^n - 1$ or
- $P * (1 + r)^n \simeq P * ((1 + r)^n - 1)$ or
- $P * ((1 + r)^n - 1)/r \simeq P * ((1 + r)^n)/r$.

Impact Analysis: The variation in the impact strength of x, the entity instance representing the incorrect value multiplied with P, is shown below.



For the detecting test cases, the impact strength of x was 1.0. There were two kinds of non-detecting test cases: those that did not involve precision loss and those that did. In the former kind, the impact strength of x was always 0.0. In the latter kind, the impact strengths were usually 1.0. The main reason for

this anomalous behavior is that our prototype currently uses a different precision floating point arithmetic than that supported by the system C library and hence the impact strengths computed did not accurately reflect the loss of precision.

B.2 Incorrect Predicate Expression

FAULT 6: Missing part of a compound predicate

Application: Market Research.

Context: analyzing account balance status. In order to determine that an account has a `strong-balance` status, three parameters are examined: `avg-monthly-balance`, `avg-annual-balance` and `avg-monthly-deposit`. The account is assigned a `strong-balance` status when a compound criterion consisting of five relational sub-expressions and four boolean operators is true.

Fault Description: One of the relational sub-expression and the associated boolean operator is missing from the compound criterion. As a result, a `strong-balance` status will be incorrectly assigned when the average monthly and annual balances satisfy the criterion but the average monthly deposit does not satisfy the criterion.

Fault Execution Condition: is always satisfied.

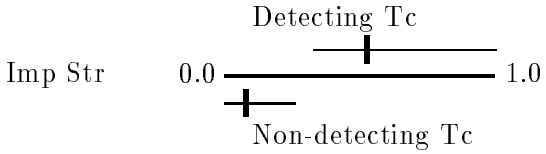
Error Creation Condition: The average annual balance is in the interval (1000, 1500], the average monthly balance is in the interval (1500, 200], and the average monthly deposit is less than 3000.

Incorrect State: The variable `account-status` has the incorrect value `strong-balance` rather than `average-balance`.

Error Propagation Condition: This error will propagate to the output if one or more of the following holds:

- The `account-status` is one of the several parameters that determine the recommended account type. The values of the other parameters is such that this error causes an incorrect recommended account type to be computed.
- The `account-status` is one of the several parameters that determine whether the customer should be sent a CD offer. The values of the other parameters is such that this error causes a CD offer to be sent.

Impact Analysis: The variation in the impact strength of `account-status` is shown below.



For the detecting test cases, the impact strengths were .34, .33, .51, 1.0, .33, .53, .33, .34, .63, and 1.0. For three of the non-detecting test cases, the impact strength was 0.25 while for the remaining seven test cases, `account-status` had no impact.

FAULT 7: Incorrect Predicate

The analysis for this fault has been already described in section 8.3 on page 149.

B.3 Extra or Missing Assignment

FAULT 8: Extra Assignment Statement

Application: String Processing (pattern matching and substitution).

Context: processing replacement text. The replacement text is converted to an internal representation after processing escape characters and the `&` special character. The variable `loop-index` keeps track of the progress of the `while` loop implementing the conversion and is incremented for every iteration.

Fault Description: An extra assignment statement in the loop causes `loop-index` to be incremented twice after processing the `&` special character.

Fault Execution Condition: An `&` character is present in the replacement text.

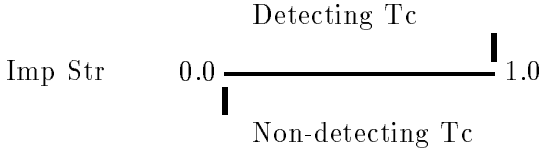
Error Creation Condition: is always satisfied.

Incorrect State: The internal representation of the replacement text is incorrect. The sequence of characters in the internal representation after the `&` character is erroneous. Let x denote the array element corresponding to the beginning of the incorrect part of the internal representation.

Error Propagation Condition: At least one character sequence in the subject text matches the specified pattern. Thus, the incorrect replacement text is printed on the output at least once. (Also, there is a possibility of an out of bound array reference causing runtime error).

Impact Analysis: In two test cases with `&` as the last character in the replacement text, the program accessed uninitialized memory and caused runtime error. Currently, our prototype does not carry out impact analysis with respect to output caused by runtime errors, so these two test cases were removed from the test suite.

The variation in the impact strength of x is shown below for the rest of the test suite.



For the detecting test cases, the impact strength of x was 1.0, while for the non-detecting test cases, x had no impact.

FAULT 9: Missing Assignment

Application: Operating System.

Context: unblocking a process. After unblocking a process from the blocked queue, it should be placed at the end of its priority queue.

Fault Description: The statement extracting the priority field from a process-control-block and assigning it to a local variable `prio` is missing. So the default value of `prio` is used and the process is placed in the `low` priority queue.

Fault Execution Condition: A process is removed from the blocked queue.

Error Creation Condition: The priority of an unblocked process is other than `low`.

Incorrect Behavior: The process is placed in a wrong priority queue.

Error Propagation Condition: There are several scenarios in which the error that a process A in the wrong priority queue can lead to an incorrect output. One such scenario is described here. There is at least one process B at a priority less than or equal to the priority of A, either present in the priority queues when A is unblocked, or coming in into the system after A is unblocked. The priority of A is not updated after unblocking and the process B finishes before A (while A should have finished first as per the test specification).

Impact Analysis: Let A denote a process which is placed in the wrong priority queue due to the fault. There were three kinds of detecting test cases:

- the process id of A was expected to appear in the output and it appears in the wrong order, or
- the process id of A was not expected to appear in the output but it appears, or
- the process id of A was expected to appear in the output but does not appear.

In the detecting test cases of the first kind, the dynamic impact requirement is satisfied and the process id of A has data impact strength 1.0. In each of the other kinds of detecting test cases, the dynamic impact requirement was violated.

There were two kinds of non-detecting test cases:

- the process id of A was not expected to appear in the output and it did not appear, or
- The process id of A was expected to appear in the output and it appears in the correct order.

In the first kind, the dynamic impact requirement is satisfied and the impact strength is 0.0. The second kind of non-detecting test cases impact requirement is satisfied, but the impact strength was non-zero because the test cases exhibited tolerance to errors in control paths. In these test cases, no processes were present or introduced between the correct and incorrect places of A in the priority queue, so A finished in the correct order in spite of the incorrect placement.

FAULT 10: Missing Assignment

Application: Calculus.

Context: performing integration. When the integration operation is specified, both `from-x` and `to-x` are supplied as command line arguments. An indefinite integral Q is obtained by integrating the input polynomial P. The required finite integral is evaluated over the interval (`from-x`, `to-x`] by computing $Q(\texttt{from-x}) - Q(\texttt{to-x})$.

Fault Description: The statement extracting the `from-x` value from command line arguments is missing. Therefore, `from-x` has the default value 0.

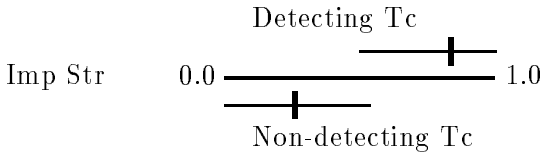
Fault Execution Condition: The integration operation is specified.

Error Creation Condition: The specified value of `from-x` is non-zero.

Incorrect State: The value of `from-x` is incorrect.

Error Propagation Condition: The incorrect value of `from-x` propagates through the polynomial evaluations and subtraction.

Impact Analysis: The variation in the impact strength of `from-x` is shown below.



For the detecting test cases, the impact strengths of `from-x` were 1.0, 1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 1.0, .86, and 0.5, and those for the non-detecting test cases were 0.0, .33, .35, .53, .33, .42, 0.0, 0.3, .32, and 0.0.

FAULT 11: Missing Assignment

Application: Graph Algorithms.

Context: finding strongly connected components (SCCs). When an scc is found, the nodes of an scc are popped from the stack and for each of these nodes, the `on-stack` field is assigned the value `false`.

Fault Description: The statement assigning the `false` value to `on-stack` is missing.

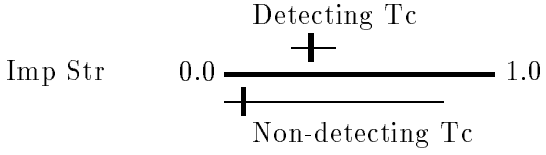
Fault Execution Condition: The input graph is non-empty.

Error Creation Condition: is always satisfied.

Incorrect State: The `on-stack` field is incorrectly `true` for a node already processed as a member of an scc.

Error Propagation Condition: The incorrect `on-stack` field of a node from a processed scc prevents the detection of at least one more scc.

Impact Analysis: The variation in the impact strength of the incorrect state(s) represented by the `on-stack` field is shown below.



There were 10 detecting test cases and 40 non-detecting test cases. For the detecting test cases, the impact strengths of the incorrect state(s) were were 0.31, 0.4, 0.25, 0.37, 0.31, 0.31, 0.33, 0.27, 0.31, and 0.37. In 37 out of the 40 non-detecting test cases, the incorrect state(s) had no impact. However, in the remaining 3 test cases, the impact strengths were 0.94, 0.82 and .82. In each of these test cases, every node of the directed input graph (with n nodes) had an outdegree of either $(n-1)$ or 0. When the input digraph has such a property, the scc algorithm produces the correct result even when some of the `on-stack` fields incorrectly have the `true` value.

B.4 Incorrect or Missing Conditional Processing

FAULT 12: Missing error checking

Application: Tax Computation.

Context: input data validation. The input data such as the filing status, earned incomes, paid taxes and qualified deductions are to be validated against obvious data entry errors such as negative values or out of bound values.

Fault Description: Error checking for the qualified `mortgage-interest` deduction is absent.

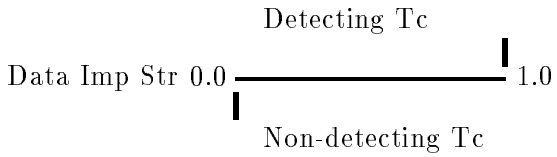
Fault Execution Condition: is always satisfied.

Error Creation Condition: The input value of `mortgage-interest` is inadvertently negative due to a data entry error. (If it were intentionally negative, the tester would expect an error report and thus detect this fault).

Incorrect State: Since the data validation is missing, compute-tax will be called when the global variable `mortgage-interest` has a negative value.

Error Propagation Condition: The error will be detected when the person's net taxable income is above the minimum income bracket and itemized deduction is preferred over standard deduction.

Impact Analysis: The variation in the data impact strength of `mortgage-interest` is shown below.



For the detecting test cases, the data impact strengths of `mortgage-interest` were 1.0 and for the non-detecting test cases, `mortgage-interest` had no data impact.

FAULT 13: Misplaced case statement

Application: Database Processing.

Context: determining the maximum value of an attribute in a relation. The `max` operation is a special case of a generic aggregate function applied to the values of an attribute in a relation. Based on the specific operation requested, the generic function performs an appropriate operation on the current `aggregate-value` and the next `attr-value` to yield the next `aggregate-value`. Starting with an identity element of the operation, this is repeated for all values of the attribute..

Fault Description: In the generic function, the case statement for processing the `max` function is misplaced. Instead of returning the maximum of the `aggregate-value` and `attr-value`, it returns the `attr-value`.

Fault Execution Condition: The `max` function is invoked.

Error Creation Condition: The sequence of attribute values is not monotonically increasing.

Incorrect Behavior: The attribute values of all but the last tuple are ignored while computing the maximum.

Error Propagation Condition: The last element in the sequence of attribute values is not equal to the maximum of the sequence and the result of the `max` operation is queried.

Impact Analysis: A dynamic impact requirement for computing the aggregate of an attribute in a relation is that all of the attribute values should have impact on the result. However, due to this fault, the attribute value of only the last tuple impacts the output in all test cases, thus violating the dynamic impact requirement.

FAULT 14: Extra `if` statement

The analysis for this fault has been already described in section 8.3 on page 142.

FAULT 15: Missing conditional processing

Context: monthly deposit transactions. A fee is charged for monthly payment transactions but not for deposit transactions. This fee is added to the amount of monthly transaction at Def1(x). This amount gets used (Use1(x)) as an actual parameter for a call to the routine processing monthly transaction. The corresponding formal argument y gets defined (Def1(y)) and then used either for a deposit transaction (Use1(y)) or for a payment transaction (Use2(y)).

Fault Description: While adding a transaction for the first time, the fee is added to the transaction amount at Def1(x) without checking whether the transaction is a payment transaction. So the amount on a monthly deposit will be incorrect.

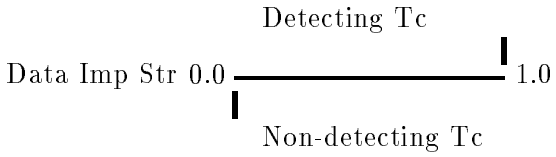
Fault Execution Condition: A monthly transaction is added for the first time.

Error Creation Condition: The added monthly transaction is a deposit transaction.

Incorrect State: The amount of a monthly deposit transaction is incorrect.

Error Propagation Condition: The incorrect amount of a monthly deposit transaction is not modified by a subsequent `change-monthly` command. The monthly transaction containing the incorrect amount is executed at least once, thus affecting the balance. The resulting balance is checked by a report inquiry.

Impact Analysis: The data transfer from Def1(x) to Use1(y) represents the execution of a monthly deposit transaction. The variation in the maximum of the impact strengths of the instances of Def1(x) associated with this data transfer are shown below.



For the detecting test cases, the impact strength was 1.0 for the instances of Def1(x) that participated in the above data transfer, while for the non-detecting test cases, such instances had either no impact or zero impact.

B.5 Iteration Errors

FAULT 16: Incorrect Loop Exit

Application: Database Processing.

Context: computing an aggregate function. In order to apply a generic aggregate function to an attribute of a relation, a `while` loop iterates over all of the tuples.

Fault Description: The exit condition is `(p->next != NULL)` instead of `(p != NULL)`. As a result, the last tuple is not processed.

Fault Execution Condition: An aggregate function is invoked and the relation has at least one tuple.

Error Creation Condition: is always satisfied.

Incorrect Behavior: The attribute value of the last tuple is ignored in computing the aggregate function.

Error Propagation Condition: Ignoring the last attribute value produces an incorrect aggregate value and the result is examined by an inquiry.

Impact Analysis: A dynamic impact requirement for computing the aggregate of an attribute is that the values of that attribute in all of the tuples of the relation should have impact on the result. However, due to the fault, the attribute value of the last tuple does not impact the output in all test cases, thus violating the dynamic impact requirement.

FAULT 17: Interacting loop initialization and loop exit faults

The analysis for this fault has been already described in section 8.3 on page 140.

FAULT 18: Duplicate expression for loop progress

Application: Database Processing.

Context: performing a select operation on a relation. A new relation Q is created by selecting tuples from a relation R. The selection is carried out using the following two steps. First, the headers are copied from R to Q using a `for` loop. Then a `while` loop iterates over all tuples of R, invoking another `for` loop for copying the attributes of each selected tuple.

Fault Description: In the first `for` loop copying the headers, the expression `i++` ensuring loop progress is present twice – once in the loop header and once at the end of the loop. As a result, starting with 0, only even numbered headers of R are copied onto Q. The odd numbered headers have the default blank names in Q.

Fault Execution Condition: A select operation is performed on a relation.

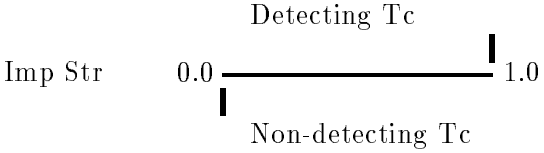
Error Creation Condition: The relation has more than one attributes.

Incorrect State: The odd numbered attribute names are blank in the resulting relation.

Error Propagation Condition: There are two alternate propagation conditions.

- A database query prints out the attribute names of the result relation Q (or a derivative of Q with at least one incorrect attribute).
- A query is made using one or more of the missing attribute names and the expected output is non-empty.

Impact Analysis: The variation in the maximum of the impact strengths of the blank attribute names is shown below.



For the detecting test cases, at least one of the blank attribute names had 1.0 impact strength. For the non-detecting test cases, all of the blank attribute names had no impact.

FAULT 19: Incorrect Loop Initialization

Application: Calculus.

Context: reading polynomial coefficients. The coefficients of the input polynomial are read in a loop. For computing the derivative of a polynomial, the constant coefficient is not significant. For computing the integral of a polynomial P over an interval, the constant coefficient of the indefinite integral of P is not significant, however, the constant coefficient of P is significant.

Fault Description: The loop initialization is designed to ignore the constant coefficient of the input polynomial P.

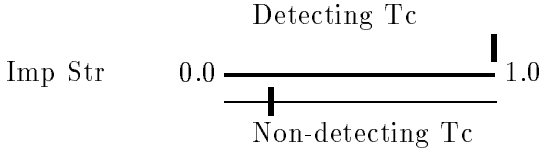
Fault Execution Condition: The polynomial is valid.

Error Creation Condition: The constant coefficient of the input polynomial is non-zero.

Incorrect State: The constant coefficient (denoted by C) in the internal representation of the input polynomial is incorrect.

Error Propagation Condition: The integration operation is requested. The constant coefficient C of the input polynomial becomes the coefficient of x^1 in the polynomial Q resulting after integration. The error in that coefficient propagates to at least one of the two evaluations of the polynomial Q.

Impact Analysis: The variation in the impact strength of C is shown below for 36 test cases.



For each of the 12 detecting test cases, the impact strength of C was 1.0. Out of the 23 non-detecting test cases in 19 test cases, C had no impact. In one test case, C had 1.0 impact but error cancellation occurred when integrating a polynomial from point v to v. That is, in an operation $Q(v) - Q(v)$, an error in a coefficient of Q gets canceled. In three test cases, there was loss of precision at the end of the computation. As mentioned before while analyzing fault 5, due to the limitations of our prototype, the computation of impact strengths does not accurately reflect the loss of precision.

B.6 Typographical Errors

FAULT 20: Missing `else` token

Application: String Processing (counting lines and characters).

Context: processing a newline character. A newline character is processed differently than other characters. The true part of an `if` statement processes a newline character and the `else` part processes all other characters.

Fault Description: The `else` token is missing for the `if` statement. As a result, a newline character is also processed as a regular character. This incorrectly increments the `num-columns` and `print-width` counters.

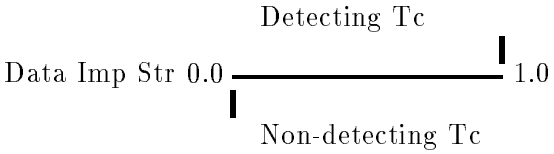
Fault Execution Condition: A newline character is present in the input.

Error Creation Condition: is always satisfied.

Incorrect State: The `num-columns` and `print-width` values are off by one for all but the first line.

Error Propagation Condition: Any line except the first line has the maximum number of columns or maximum print width.

Impact Analysis: The variation in the maximum of the data impact strengths of the incorrect entity instances of `num-columns` and `print-width` is shown below.



For detecting test cases, at least one of the incorrect instances had data impact on the output with strength 1.0. For non-detecting test cases, none of the incorrect incorrect had data impact on the output.

FAULT 21: Misplaced brace, Incorrect scope of an assignment

Application: String Processing (pattern matching and substitution).

Context: creating internal representation for the input pattern. The input pattern (a regular expression) is converted into an internal representation. The variable `cur-pat-index` keeps track of the current index into the pattern and the variable `last-pat-index` keeps track of the last index before processing the current sub-expression. After processing every sub-expression, `last-pat-index` should be updated.

Fault Description: Due to a misplaced brace, `last-pat-index` gets updated only after processing a non-special literal character in the input pattern.

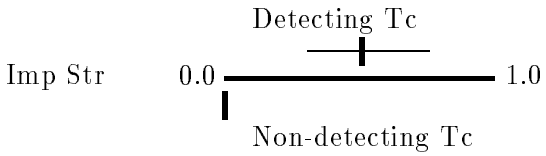
Fault Execution Condition: The input regular expression is valid and it contains at least one of the special characters (e.g. `?`,`*`,`[`, etc.)

Error Creation Condition: is always satisfied.

Incorrect State: The value of `last-pat-index` is incorrect just after each execution of the fault.

Error Propagation Condition: An incorrect pattern representation is generated when a special character is followed by a closure operator, and the special character is not `%` , `$` or `*`. For this incorrect pattern representation to cause an output error, either the pattern incorrectly matches some text, or the pattern incorrectly does not match some text, or the program goes into an indefinite loop.

Impact Analysis: In this case, impact of `last-pat-index` on the output entity (`replace-char`) corresponding to the replacement text was the main feature that distinguished the detecting test cases from the non-detecting test cases as shown below.



For the detecting test cases, the impact strengths of `last-pat-index` were .47, .51, .75, .52, .67, .31, .48, .61, .38, and .41, while for the non-detecting test cases, `last-pat-index` had no impact on `replace-char`. Out of the 10 non-detecting test cases, in six of them, the incorrect state had absolutely no impact on the output; in two of them, the incorrect state had strong impact on loop-exits; and in one test case, the incorrect state had strong impact on the output entities corresponding to printing of unmatched original text. Thus, considering impact on *all* output entities, in three out of ten test cases, due to error-tolerance, the error did not propagate in spite of strong impact.

FAULT 22: Operator Reference Fault

Application: Market Research.

Context: determining strong balance status. Based on certain criteria, an account is classified as having `strong-balance`, `weak-balance` or `average-balance`. One of the sub-criterion for determining `average-balance` is that `avg-monthly-debit` should be greater than 95% of the `avg-monthly-deposit`.

Fault Description: The sub-criterion is mistyped with a `<` instead of a `>`.

Fault Execution Condition: The criteria for `strong-balance` are not satisfied, and `avg-monthly-bal > 500` or `avg-annual-bal > 400`.

Error Creation Condition: The `avg-monthly-debit` is not equal to 95% of the `avg--monthly-deposit`.

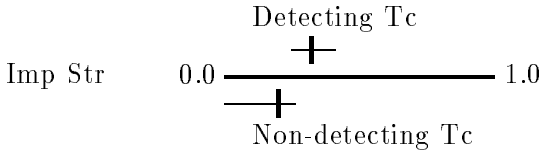
Incorrect State: The value of the predicate is incorrect, and hence incorrect branch is taken and as an immediate effect, the value of variable `account-status` is

incorrect. Depending on the error, the value is either `average-balance` (Def1(x)) or `weak-balance` (Def2(x)), while it should have been the other.

Error Propagation Condition: The incorrect value of the `account-status` affects the output by causing one or more of the following:

- incorrect computation of the recommended account type as overdraft protection type, or
- incorrectly issuing or not issuing the first home loan promotion or home equity loan

Impact Analysis: The variation in the maximum of the impact strengths of the incorrect instances of `account-status` is shown below.



For the detecting test cases, the impact strengths were 0.26, 0.33, 0.25, 0.33, 0.33, 0.33, 0.33, 0.4, 0.25, and 0.4. For two of the non-detecting test cases the incorrect instances had no impact while for the remaining eight test cases, the impact strengths were 0.25.

FAULT 23: Variable Reference Fault

Application: Database Processing.

Context: loading a relation. While loading a relation from a file, the program first reads the name of the relation, number and names of the attributes, and then reads the rows of attribute values until the end-of-file is reached. While reading the rows of attribute values, there are two read (`scanf`) calls: one to check the end-of-file condition while getting the value of the first attribute, and the other to read the remaining attribute values in a loop.

Fault Description: In the first read(`scanf`) call, instead of passing the address of `value`, the address of `svalue` is passed. Due to this fault, the first attribute of a loaded relation has incorrect values for all tuples.

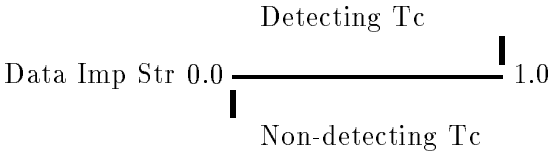
Fault Execution Condition: A non-empty relation is loaded from a file.

Error Creation Condition: The following unusual conditions *do not* hold: the value of the first attribute in the first row is 0.0, and for $i > 1$, the value of the first attribute in row i is the value of the last attribute in row $i - 1$.

Incorrect State: Some of the values for first attribute of the loaded relation are incorrect.

Error Propagation Condition: One or more of the incorrect first attribute values take part in subsequent database operations and produce an incorrect output.

Impact Analysis: The variation in the maximum of the data impact strengths of the incorrect attribute values is shown below.



For the detecting test cases, at least one of the incorrect attribute values had data impact on the output with strength 1.0. For the non-detecting test cases, none of the incorrect attribute values had data impact on the output. However, for some of the non-detecting test cases, one or more incorrect attributes had control impact on loop-exits with impact strengths ranging from 0.19 to 1.0.

FAULT 24: Constant Reference Fault

Application: Market Research.

Context: classifying account balance status. Based on certain criteria, an account is classified as having `strong-balance`, `weak-balance` or `average-balance`. One of

the sub-criterion for determining `strong-balance` is that `avg-annual-bal` should be greater than `1000.0`.

Fault Description: The sub-criterion is mistyped with the constant `1100.0` instead of `1000.0`.

Fault Execution Condition: The value of `avg-monthly-bal` is greater than 1500.0 and either `avg-monthly-bal` is less than 2000.0 or `avg-annual-bal` is less than 1500.0.

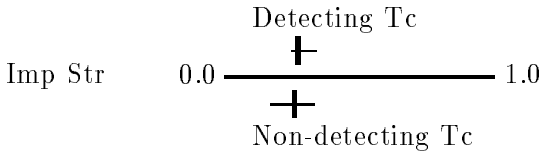
Error Creation Condition: The value of `avg-monthly-bal` is between 1000.0 and 1100.0.

Incorrect State: The false branch is incorrectly taken, so `account-status` has the value `average-balance`. This value is incorrect only if the remaining sub-criterion for `strong-balance` is true.

Error Propagation Condition: The `avg-monthly-deposit` is greater than 3000 and the incorrect value of `account-status` affects the output by causing one or more of the following:

- computation of an incorrect recommended account type, or
- incorrectly determining eligibility for a first home loan or a home equity loan.

Impact Analysis: The variation in the impact strength of the incorrect instance of `account-status` is shown below.



For the detecting test cases, the impact strengths were .28, .29, .26, .25, .33, .26, .25, .25, .26, and .25, and for the non-detecting test cases, the impact strengths were .25, .32, .17, .28, .25, .17, .28, .29, .29, and .29. As expected, when impact strengths are low, the error does not always propagate.

B.7 Interface Errors

FAULT 25: Incorrect return value semantics

The analysis for this fault has been already described in section 8.3 on page 145.

FAULT 26: Incorrect order of arguments

Application: Accounting.

Context: processing transactions on demand. The main procedure calls the routine for processing transaction with three parameters: `command` (whether payment or deposit), expense `category` and the `amount` of the transaction.

Fault Description: The `command` and `category` parameters are interchanged at the call-site. Since both parameters have the same type, compiler does not complain.

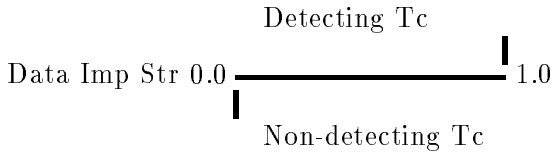
Fault Execution Condition: An on-demand payment transaction is requested.

Error Creation Condition: The integer values of command and category are different.

Incorrect State: Depending on the value of `category`, the transaction will either do nothing or incorrectly update the `balance` (say X). In either case, the `expense--category` (say Y) which should have been updated is left untouched. Thus, the error creation condition ensures that Y is not modified, and (optionally) X gets updated.

Error Propagation Condition: The incorrect state propagates to the output when the previous definition of Y participates in a computation whose results are examined by a report query. In addition, if variable X is incorrectly updated, then the incorrect state is also propagated when X participates in a computation whose results are examined by a report query.

Impact Analysis: The variation in the maximum of the impact strengths of the incorrect value of Y and the incorrect value of X (when present) is shown below.



For detecting test cases, one or more of the following were observed:

- previous definition of Y has impact strength 1.0, or
- some variable X is corrupted and that definition of X has impact strength 1.0.

For non-detecting test cases, both of the following were observed:

- previous definition of Y has either no impact or zero impact, or
- if variable X is corrupted, the corresponding definition of X has either no impact or zero impact.

FAULT 27: Failure to update a global variable

Application: Accounting.

Context: book-keeping at the end of a month. The procedure processing the `end-of-month` is required to update the global variable `last-balance`.

Fault Description: The procedure processing the `end-of-month` command does not update `last-balance` assuming that a higher level procedure will carry out the update.

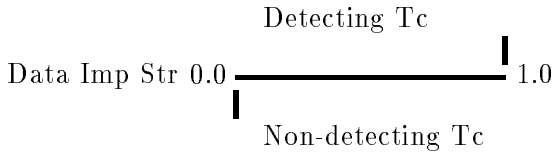
Fault Execution Condition: At least one `end-of-month` command is executed.

Error Creation Condition: For one or more months, the balance at the end of the month is different from that at the end of the previous month.

Incorrect State: The global variable `last-balance` has an incorrect value.

Error Propagation Condition: At least one more `end-of-month` command is processed after the incorrect state. The interest rate is non-zero, and a balance inquiry is made.

Impact Analysis: The variation in the maximum of the data impact strengths of the incorrect values of `last-balance` is shown below.



For the detecting test cases, at least one of the incorrect value of `last-balance` had impact strength 1.0, while for the non-detecting test cases, the incorrect values of `last-balance` had either no impact or zero impact.

FAULT 28: Incorrect argument

The analysis for this fault has been already described in section 8.3 on page 144.

B.8 Other Errors

FAULT 29: Sequencing fault

Application: Database Processing.

Context: performing select operation on a relation. While adding selected tuples to the resulting relation, a `counter` keeps track of the number of tuples added so far.

Fault Description: The statement incrementing the `counter` is placed after, rather than within, the conditional statement adding a tuple to the resulting relation. As a result, the *counter* value is incorrect.

Error Creation Condition: There is at least one tuple *not* selected from the original relation.

Incorrect State: The `counter` value is incorrect.

Error Propagation Condition: The incorrect `counter` value is used to update (Def1(x)) the `num-tuples` field in the resulting relation. Although, the `num-tuples` field is used in several computations, it is not possible for a test case to execute the def-use association from Def1(x) to any such use. Hence, the error cannot be propagated.

Impact Analysis: As expected, there were no detecting test cases. For the non-detecting test cases, the incorrect `counter` value had no impact on the output.

FAULT 30: Incorrect read format

Application: Graph Algorithms (bi-connected components).

Context: reading an undirected graph. The input undirected graph is presented as a sequence of node pairs representing the edges of the graph.

Fault Description: While restructuring a loop using cut and paste edit operations, several faults were created. The initial call to `scanf` is intended to read both the nodes of the first edge, but due to an incorrect read format string only one node will be read. The loop initialization, which should have been deleted, contains a `scanf` call for reading the first node. Each of these individual faults is easily detectable, but the resulting complex fault is harder to detect. As a result of the complex fault, instead of inserting the first edge e (from node m to node n) from the input, an incorrect edge between node 0 and node n is inserted in the graph.

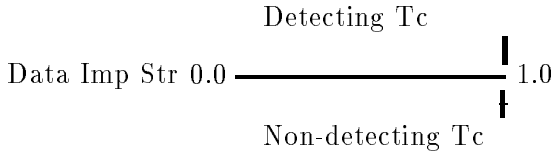
Fault Execution Condition: The input graph is non-empty.

Error Creation Condition: The first edge e of the input graph is not a self-loop at node 0.

Incorrect State: While loading the input graph, the start and end nodes of the first edge are read incorrectly.

Error Propagation Condition: The computation of a binary connected component (bcc) in the program is always affected by the first edge (whether correct or incorrect). However, if the correct edge was (0,n), the incorrect edge will be (n,0), still giving the same undirected graph. This is a good example of an incorrect state which is not really incorrect for the problem being solved. In such cases, the error in internal state will not propagate to the output. Thus, the error will be detected only if the start node of the first edge is not 0.

Impact Analysis: The data impact strengths of the entity instances corresponding to the incorrect edge were more or less the same for the detecting and the non-detecting test cases as shown below.



This is so, since all edges of the graph including the first edge always has impact on the computed bi-connected components. In all of the non-detecting test cases, the first edge was of the form $(0, n)$, and the fault caused a simple inversion of the first edge to $(n, 0)$, which is a no-operation for an undirected graph. This is a good example, where an incorrect state for the program is not really incorrect for the problem being solved by the program. Due to specific representation of the undirected graph, such an incorrect state affects the control paths but not the output.

Bibliography

[1] ACM. *Collected Algorithms from ACM*, volume 1. Association for Computing Machinery, Inc., New York, 1980.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 246–256, White Plains, New York, June 1990.

[3] Hiralal Agrawal, Joseph R. Horgan, E. W. Krauser, and S. L. London. Incremental Regression Testing. In *Proc. IEEE Conf. on Software Maintenance*, pages 348–357, White Plains, New York, Sept. 1993.

[4] Aho, Hopcroft, and Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[6] Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand. Automatic Generation of Test Scripts from Formal Test Specifications. In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 210–218, KeyWest-Florida, Dec. 1989.

[7] James M. Bieman and Janet L. Schultz. Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion. In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 179–186, KeyWest-Florida, Dec. 1989.

[8] G.E.P. Box, W.G. Hunter, and J.S. Hunter. *Statistics for Experimenters.* John Wiley & Sons, 1978.

[9] R.S. Boyer, B.E. Elspas, and K.N. Levitt. SELECT–A Formal System for Testing and Debugging. In *Proc. of the Intl. Conf. on Reliable Software*, Los Angeles, April 1975.

[10] Timothy A. Budd. Mutation Analysis: Ideas, Examples, Problems and Prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland Publishing Company, 1981.

[11] Lori A. Clarke and Debra J. Richardson. Symbolic Evaluation Methods – Implementations and Applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–101. North-Holland Publishing Company, 1981.

[12] R. A. Demillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An Extended Overview of the Mothra Software Testing Environment. In *Proc. ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pages 142–151, Banff-Canada, 1988.

[13] R. A. Demillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practising programmer. *Computer*, 11(4):34–41, April 1978.

[14] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Rigorous Data Flow Testing through Output Influences. In *Proc. 2nd Irvine Software Symposium (ISS'92)*, pages 131–145, Irvine, CA, March 1992.

[15] I.M.M. Duncan and D. J. Robson. Ordered Mutation Testing. *ACM SIGSOFT Software Engineering Notes*, 15(2):29–30, Apr 1990.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[17] Phyllis G. Frankl and Elaine J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[18] Tom Gilb. *Software Metrics*. Studentlitteratur (Sweden), 1976.

[19] Tarak Goradia. Dynamic Impact Analysis: A Cost-effetive Technique to Enforce Error-propagation. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 171–181, Cambridge, Massachusetts, June 1993.

[20] Michael Greenberg. The Frame System. Technical report, Siemens Corporate Research, Inc., March 1990.

[21] M.H. Halstead. *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[22] Richard Hamlet. Theoretical Comparison of Testing Methods. In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 28–36, KeyWest-Florida, Dec. 1989.

[23] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, July 1977.

[24] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-interfering Versions of Programs. Technical Report 690, Computer Science Dept., University of Wisconsin, March 1987.

[25] Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs For Representing Programs. Technical Report 699, Computer Science Dept., University of Wisconsin, June 1987.

[26] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, June 1988.

[27] William A. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[28] William E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, Oct. 1986.

[29] IEEE. Standard Glossary of Software Engineering Terminology,. In *IEEE Standard 729-1983*. IEEE, 1983.

[30] Daniel Jackson. Abstract Analysis with Aspect. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–27, Cambridge, Massachusetts, June 1993.

[31] BingChiang Jeng. *A New Approach to Domain Testing*. PhD thesis, New York University, June 1990.

[32] Guy L. Steele Jr. *Common Lisp*. Digital Press, 2nd edition, 1990.

[33] B. W. Kernighan and P. J. Plauger. *Software Tools in Pascal.* Addison-Wesley, 1981.

[34] Donald Knuth. The Errors of TEX. *Software Practice & Experience*, Jul 1989.

[35] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. In *Information Processing Letters 29*, pages 155–163. Elsevier Science Publishers B.V. (North-Holland), October 1988.

[36] Bogdan Korel and Janusz Laski. Algorithmic Software Fault Localization. In *Proc. the Twenty Fourth Annual Hawaii International Conf. on System Sciences*, pages 246–252, 1991.

[37] D. J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 207–218, Williamsburg, Virginia, Jan 1981.

[38] D.J. Kuck, Y. Muraoka, and S.C.Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transaction on Computers*, C-21:1293–1310, Dec 1972.

[39] Janusz W. Laski and Bogdan Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[40] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. Technical report, University of California, Irvine, Nov 1992.

[41] Brian Marick. Two Experiments in Software Testing. Technical Report UIUCDCS-R-90-1644, University of Illinois, 1990.

[42] Brian Marick. The Weak Mutation Hypothesis. In *Proc. ACM SIGSOFT Fourth Workshop on Software Testing, Analysis and Verification*, 1991.

[43] Brian Marick. A Question Catalog for Code Inspections. Testing Foundations, Chapaign, Illinois, June 1992.

[44] Edward W. Minium. *Statistical Reasoning in Psychology and Education.* John Wiley & Sons, 2nd edition, 1978.

[45] Larry J. Morell. A Theory of Error-Based Testing. Technical report, Dept. of Computer Science, TR-1395, 1984.

[46] Larry J. Morell. Theoretical Insights into Fault-Based Testing. In *Proc. ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pages 45–62, Banff-Canada, 1988.

[47] Larry J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[48] Larry J. Morell and Branson Murrill. Error Flow Testing. In *Proc. of the 8th Int'l Conf. on Testing Computer Software*, pages 105–121, Washington D.C., 1990.

[49] Larry J. Morell and Branson Murrill. Semantic Metrics through Error Flow Analysis. *J. Systems Software*, 20(3):253–265, Mar 1993.

[50] Branson W. Murrill. *Error Flow in Computer Programs.* PhD thesis, The College Of William and Mary in Virginia, 1991.

[51] Simeon C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Nov. 1984.

[52] Simeon C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

[53] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Dept. of Information and Computer Science, Georgia Institute of Technology, 1988.

[54] A. J. Offutt. The Coupling Effect: Fact of Fiction? In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 131–140, KeyWest-Florida, Dec. 1989.

[55] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[56] Thomas J. Ostrand and Elaine J. Weyuker. Error-based Program Testing. In *Proc. Conf. on Information Sciences and Systems*, Baltimore, March 1979.

[57] Thomas J. Ostrand and Elaine J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Sofware*, 4:289–300, 1984.

[58] Karl J. Ottenstein and Steven J. Ellcey. Experience Compiling Fortran to Program Dependence Graphs. *Software Practice & Experience*, Jan 1992.

[59] Hemant Pande, Barbara Ryder, and William Landi. Interprocedural Def-Use Associations in C Programs. In *Proc. ACM SIGSOFT Symposium on Testing, Analysis and Verification*, pages 139–154, Victoria, British Columbia, Oct. 1991.

[60] Hemant Pande, Barbara Ryder, and William Landi. Interprocedural Reaching Definitions in the Presence of Single Level Pointers. *IEEE Transactions on Software Engineering*, 1993. Accepted for publication.

[61] Michael Platoff, Michael Wagner, and Joseph Camaratta. An Integrated Program Representation and Toolkit for the Maintenance of C Programs. In *Proceedings of*

the Conference on Software Maintenance. IEEE Computer Society Press, October 1991.

[62] Andy Podgurski and Lori A. Clarke. The Implications of Program Dependences for Software Testing, Debugging and Maintenance. In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 168–178, KeyWest-Florida, Dec. 1989.

[63] Andy Podgurski and Lori A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sept. 1990.

[64] Sandra Rapps and Elaine J. Weyuker. Data Flow Analysis Techniques for Program Test Data Selection. In *Proc. Sixth International Conference on Software Engineering*, pages 272–278, Tokyo-Japan, Sept. 1982.

[65] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[66] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to Specification-Based Testing. In *Proc. ACM SIGSOFT Third Workshop on Software Testing, Analysis and Verification*, pages 86–96, KeyWest-Florida, Dec. 1989.

[67] Debra J. Richardson and Margaret C. Thompson. The RELAY Model of Error Detection and its Application. In *Proc. ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pages 223–230, Banff-Canada, 1988.

[68] Debra J. Richardson and Margaret C. Thompson. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, Jun 1993.

[69] Hasan Ural and Bo Yang. A Structural Test Selection Criterion. *Information Processing Letters*, 28(3):157–163, July 1988.

[70] G. A. Ventkatesh. The Semantic Approach to Program Slicing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 107–119, Toronto, June 1991.

[71] Jeffrey M. Voas. Factors That Affect Software Testability. In *Proc. Ninth Pacific Northwest Software Quality Conference*, 1991.

[72] Jeffrey M. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug. 1992.

[73] Jeffrey M. Voas and Keith Miller. Improving Software Reliability by Estimating the Fault Hiding Ability of a Program Before it is Written. In *Proc. of the 9th Software Reliability Symposium*, Colarado Springs, May 1991.

[74] Jeffrey M. Voas and Jeffery E. Payne. Designing Programs that are Less Likely to Hide Faults. *J. Systems Software*, 20(1):93–101, Jan 1993.

[75] Stewart N. Weiss. What to Compare When Comparing Test Data Adequacy Criteria. *ACM SIGSOFT Software Engg. Notes*, 14(6):42–49, Oct. 1989.

[76] Stewart N. Weiss and Vladimir N. Fleyshgakker. Improved Serial Algorithms for Mutation Analysis. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 149–158, Cambridge, Massachusetts, June 1993.

[77] Elaine J. Weyuker. An Empirical Study of the Complexity of Data Flow Testing. In *Proc. ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pages 188–195, July 1988.

[78] Elaine J. Weyuker. The Cost of Data Flow Testing: an Empirical Study. *IEEE Transactions on Software Engineering*, 16(2):121–128, Feb. 1990.

[79] Elaine J. Weyuker, Stewart N. Weiss, and Richard Hamlet. Comparison of Program Testing Strategies. In *Proc. ACM SIGSOFT Fourth Workshop on Software Testing, Analysis and Verification*, 1991.

[80] Lee J. White, Edward I. Cohen, and Steven J. Zeil. A Domain Strategy for Computer Program Testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 103–113. North-Holland Publishing Company, 1981.

[81] Steven J. Zeil. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.

[82] Steven J. Zeil. Equate Testing Strategy. In *Proc. ACM SIGSOFT Workshop on Software Testing, Analysis and Verification*, pages 142–152, Banff-Canada, 1986.

Index

Bibliography Index