The Influence of Multiple Artifacts on the Effectiveness of Software Testing

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Matthew Staats

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Mats P.E. Heimdahl, Advisor
May, 2011

Acknowledgments

Many, many people have ultimately contributed to this accomplishment, and this set of acknowledgements is necessarily incomplete. Thanks go to Mats Heimdahl, my advisor, whose support, guidance, and wisdom were, and remain, invaluable. Most advisors appear mediocre to poor; he is not, and I am fortunate for it. Thanks go to Mike Whalen, whose practical work experience and technical wizardry have made this dissertation possible, and whose friendship has supported me in the rougher parts of producing this work. Thanks go to all the Crisys lab members: Ajitha Rajan, Myung-Hwan Park, Weijia Deng, Anjali Joshi, Greg Gay, Ian De Silva, Hung Tuan Pham—you have all supported me with both friendship and research contributions.

Thanks go to Rockwell Collins, who have graciously allowed me, through Mike Whalen, to study their systems and publish my findings. Without their support, this dissertation would be far less practically relevant.

Thanks go to the many coaches and teachers that have pushed me to excel in all I do. A special thanks go to Dennis Brown and Clayton Price, both of whom especially challenged me to exceed expectations.

Thanks go to the many pubs and coffee houses in Minneapolis and St. Paul, notably Town Hall and Sporty's, for providing the right chemicals, at the right time, with free wifi to match. Much of this dissertation was written, or conceived of, over a cup of coffee or a pint of beer.

Finally, thanks go to my family: my parents, Jim and Darlene, my in-laws, Tom and Dorothy, and especially my wife, Brenna. You have supported me through five years of graduate school, four moves, and innumerable long nights. Without your support and encouragement this dissertation would not have been possible.

Abstract

The effectiveness of the software testing process is determined by several artifacts used in testing, including the program, the set of tests, and the test oracle. However, when evaluating software testing techniques, the influence of some testing artifacts is often overlooked, with much of testing research focusing solely on the test inputs selected. This oversight represents both a potential problem, and a potential opportunity. If there exists interrelationships between these artifacts, a failure to consider these relationships represents a threat to the validity of software testing studies. However, by understanding these potential relationships, we may discover insights that allow us to improve the testing process.

In this dissertation, we explore the interrelationship between three testing artifacts: the program, the test oracle, and the test inputs, establishing a *foundation* for understanding how they interact. We provide two core contributions towards this goal. First, we propose a theoretical framework for discussing testing based on previous work in the theory of testing. Second, we perform a rigorous empirical study controlling for program structure, test coverage criteria, and oracle selection in the domain of safety critical avionics software.

We find that these interactions are both theoretically and practically present, with contributions including: an improved conceptual framework for discussing software testing; a demonstration of errors in existing theoretical work on testing related to test oracle assumptions; demonstration that program structure strongly impacts both the cost and effectiveness of structural coverage criteria, with negative implications for current testing practice; results indicating that considering the interactions and tradeoffs between test oracles and test inputs may yield improvements in testing

practice; demonstration of how program structure influences test oracle effectiveness, thus indicating methods of improving observability may improve testing effectiveness. These results thus highlight the importance of understanding the interaction between testing artifacts, quantify how these interactions occur in avionics software, and point towards future directions in testing research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software testing is a key component of the validation and verification (V&V) phase of software development. This phase of software development is a potentially costly and time consuming process, often requiring significant manual effort on the part of developers and testers. Consequently, determining how to effectively test software is both a crucial and common task, and evaluating proposed testing techniques is an active area of testing research.

Conceptually, the process of software testing is simple: we run the software using some specified input, observe the software's execution, and decide if the execution appears to be correct. While the specific goal or focus of testing can vary, generally speaking we are interested in detecting faults within the software. Several artifacts are involved in the testing process, including the set of *test inputs* to be run (more commonly referred to as *test data* in the testing literature), the *test oracle*, or method of determining the correctness of the software, the *software* or *program* to be tested, and the *specification* the software is intended to implement.

Testing research, however, is predominately focused on determining what *test data* to use, e.g., creation and evaluation of test coverage criteria or automatic test generation tools. Evaluations of these techniques often only explicitly vary a single factor—the test data used—while simultaneously using a small number of software systems and a single, often implicit, test oracle. The influence of the software system and the nature of the test oracle are often unexplored, with the reader left to infer what, if any, effect these artifacts may have on the results.

The *research objective* of this dissertation is to provide an impetus for testing researchers to consider how the various testing artifacts may jointly influence the effectiveness of the testing process. Two primary reasons motivate this objective. First, as with all empirical research, effective empirical testing research requires us to be aware of the factors influencing our results. If we do not control for, or at least understand, the factors underlying our results, our conclusions will be at best limited in their generalizability and at worst incorrect. Second, the research community may derive new testing techniques or improve existing techniques through a better understanding of the artifacts influencing the effectiveness of testing.

Of course, this is a broad goal, not a problem that can be "solved" in a definitive fashion. In our dissertation, we work towards a *solid scientific foundation* for thoroughly understanding the interrelationship between four required testing artifacts: tests, programs, oracles, and specifications. We provide two primary contributions towards this foundation: (1) a theoretical framework for discussing testing, and (2) a rigorous multifactor empirical study on test effectiveness examining several potential interactions between testing artifacts in the context of safety critical avionics systems.

1.1 A Holistic View of Testing Effectiveness

Early work on the testing process focused on test selection, leading to the definition of test data adequacy criteria by Goodenough and others [26, 27, 73]. Subsequent work in testing has reinforced this test-data-selection-centric view of testing, with one author even observing that the problem of testing is to *"find a graph and cover it"* [5]. Unsurprisingly, a large quantity of work explores which test coverage criteria to use, how to generate tests satisfying various criteria, etc., while mostly ignoring the influence of other testing artifacts such as test oracles.

Nevertheless, some work does explore the influence of other artifacts on testing

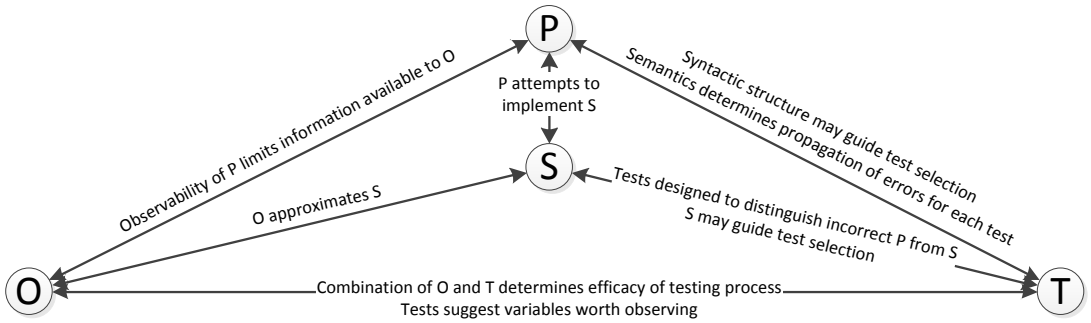


Figure 1.1: Example Relationships Between Testing Artifacts
P = Programs, S = Specification, T = Test Data, O = Oracles

research. In Figure 1.1, we illustrate potential interrelationships between the four testing artifacts: programs, specifications, tests, and oracles. These interrelationships are drawn from our own observations and those of other authors, which we detail in Chapter 2. Some of these relationships are straightforward and intuitive. For example, the relationship between the specification and the program is fairly obvious: the program is intended to implement the specification. The relationship between programs and tests can be explicit, as is the case when using structural coverage criteria, where the adequacy of a test suite is based on how well it covers syntactic aspects of the program structure [83]. Other relationships are less obvious. Examples include using testability information to identify program locations where faults are difficult to detect, thus indicating locations where assertions may be placed to improve the quality of the oracle [67] (program structure used to inform oracle construction), and the joint influence of test sets and oracles on fault finding effectiveness [47, 10].

However, studies exploring these relationships are relatively uncommon in the body of testing research. This leads to a generally one-dimensional view of testing effectiveness centred on test inputs. We believe that by adopting a more holistic

view of testing that explicitly considers multiple factors, we can (1) improve our understanding of existing testing techniques through better constructed evaluations and subsequently (2) improve the effectiveness of the testing process through a better overall understanding of testing. Several surveys exploring existing testing research support this belief, with two main points emerging.

First, several authors call for a stronger empirical foundation in testing research [33, 8, 42]. Such a foundation is needed to evaluate the effectiveness of existing and proposed techniques and to further evolve the state of the art. Of particular relevance to this dissertation are calls for work beyond test selection, with a specific call for work on test oracles [8].

Second, the quality of empirical studies in testing research is perhaps not as high as we would like [9]. While several problems are highlighted, of particular relevance to this dissertation is the existence of several potential confounding factors in testing research, and their ability to negatively impact the internal validity of studies. These confounding factors include the test oracle selected, the system under test, the type of faults used, and the training and skill of testers (when performing experiments with human subjects) [9].

1.2 Methods of Evaluating Testing Techniques

In testing research, we often wish to evaluate a testing technique, generally to determine the cost of applying the technique and the effectiveness of the technique in terms of fault detection ability. We divide approaches for discussing and evaluating testing techniques into two basic approaches: the theoretical approach and the empirical approach.

In the theoretical approach, testing techniques are formalized using some model of testing. Often, this framework is then used to show properties of interest hold

or do not hold. Potential properties of interest include proving a testing technique can or cannot demonstrate complete correctness of a program [73], proving a testing technique can or cannot distinguish between a correct program and a specific set of "almost" correct programs [11], or proving a testing technique is always or probabilistically "better" than some other testing technique [74].

In the empirical approach, traditional scientific methods of experimentation and observation are applied to testing techniques. In these studies, the test subjects are collections of software engineering artifacts (including programs), and the experiments consist of applying one or more testing techniques to the selected artifacts. The results, which may consist of fault finding measurements, test coverage measurements, etc., are then analyzed and conclusions are drawn concerning the effectiveness of the examined testing techniques.

Both approaches have benefits and disadvantages, and we therefore employ both in this work. The formal descriptions of testing inherent in theoretical work can be useful for discussing the problems of software testing in a rigorous and unambiguous fashion. Furthermore, the theoretical approach excels at highlighting the limitations of testing, e.g., the inability of testing to (in general) prove the correctness of software [11, 73, 27]. However, such results may not apply to actual systems developed by programmers, and therefore may not be practically applicable.

The empirical approach allows us to examine questions relevant to effectively applying testing techniques. Specifically, the results of controlled experiments allow us to precisely quantify questions of interest using actual systems, e.g., measure the relative fault finding of testing techniques. Unlike the theoretical approach, however, no consistent conceptual framework needs to exist in empirical studies; the experimenters can describe the problem studied in whatever way they wish. Consequently, important relevant information may be omitted from the description of empirical

studies. This makes interpreting the results and comparing across studies difficult, and prevents, for example, the conduction of rigorous meta-studies in software engineering. However, demonstrating that some conclusion holds in general is difficult or impossible, and therefore generalizability suffers relative to theoretical approaches.

1.2.1 Relationship Between Theoretical and Empirical Approaches

In principle, both approaches exist independent of one another. Indeed, the approaches often address different questions and can thus be thought of as complementary tools in testing research.

Nevertheless, theoretical work should influence empirical work in at least two ways. First, theoretical work often includes concise, clear descriptions of testing techniques. When used to frame our discussion, these descriptions provide a rigorous conceptual framework for discourse, helping to describe the problem of interest, why the problem is relevant, and how the empirical study is conducted.

Second, theoretical work often highlights problems worthy of study. Clearly, empirical testing research, like empirical research in any field, frequently highlights interesting problems for study. However, the problems highlighted in theoretical testing research, being derived from formal descriptions of testing, tend to be concise, clear and general—in other words, they tend to be key high-level problems.

Furthermore, theoretical work often describes the characteristics of an ideal solution. For example, early work in testing research by Gourlay defines a *test method* as a function mapping a program and a specification to a set of tests [27]. This is a concise, clear, and general description of a problem common in testing, that of determining the test data used. Gourlay then continues on to define the *power* comparison for test methods, stating that test method M is more powerful than test method N if, when N finds a fault, M finds a fault. Unfortunately, this comparison is

limited in practical application. These limitations are demonstrated by Weyuker [74], who shows that few test methods are usefully comparable under Gourlay's definition. Thus, Gourlay's original work clearly defines a problem and a method of comparing solutions, while subsequent work highlights the practical limitations of possible solutions.

This example demonstrates how theoretical work can help define the problems and ideal solutions of testing research. We therefore revisit this work, extend it to better account for certain overlooked aspects of testing (such as test oracles), and highlight how artifacts may potentially interact.

1.3 Objective and Contributions

Our long-term *research objective* is to improve the quality of testing by improving our understanding of how testing artifacts may interact to influence the effectiveness of the testing process. This objective is motivated by three factors discussed above: (1) the need for a solid foundation for testing research, (2) shortcomings in existing empirical studies, and (3) a lack of work exploring how multiple artifacts can influence testing effectiveness, despite evidence that this interaction exists. By better understanding how testing artifacts interact, we may be able to improve the effectiveness of the testing process, both through better understanding of testing and through better evaluations of proposed testing techniques.

Of course, we recognize that much work needs to be done to improve our understanding of how testing artifacts influence testing effectiveness, and that this dissertation can only form a small part of this work. Accordingly, the *objective of this dissertation* is to improve our understanding of how testing artifacts interact to influence testing effectiveness specifically within the domain of safety critical avionics systems. As outlined in the previous section, in pursuing our goal we employ two

approaches: (1) the development of a theoretical framework for discussing testing, defining relevant testing concepts and highlighting problems in existing frameworks, and (2) the conduction of a large-scale empirical study exploring several practically relevant interactions between testing artifacts.

In this dissertation we make the following specific contributions to towards understanding how multiple artifacts influence the effectiveness of the testing process:

Theoretical Testing Framework: We develop a theoretical functional framework for discussing testing. This framework improves upon the conceptual framework laid down by previous work in the area, notably that of Gourlay [27]. We apply this functional framework to discuss and define new, useful and relevant testing concepts, particularly related to properties of test oracles and the selection of test oracles. We also apply this functional framework to demonstrate how *test oracles* can impact the testing process; notably, we demonstrate that previous theoretical results rely on an implicit assumption of a fixed oracle, and that varying this test oracle can sometimes result in different conclusions. (Chapter 3.)

Impact of Program Structure on Structural Coverage Criteria: We empirically demonstrate how program structure impacts the effectiveness of the testing process when using structural coverage criteria (e.g., branch coverage). We find that by varying the complexity of expressions in the system under test, we can impact both the size and effectiveness of test suites satisfying structural coverage criteria. Unfortunately, we also find that these two potential impacts are correlated—a reduction in test suite size corresponds to a reduction in effectiveness. In the context of avionics systems, these results are disconcerting, as the use of structural coverage criteria is required when testing certain avionics systems. We demonstrate that financial rewards through decreased test suite

size may be possible when restructuring systems, but that this restructuring may also cause significant decreases in effectiveness. (Chapter 5.)

Influence of Test Oracles and Test Inputs on Effectiveness: We empirically demonstrate how test oracle selection can impact the effectiveness of the testing process when using a variety of test inputs. Notably, we demonstrate tradeoffs are often present in testing: depending on the costs associated with testing artifacts and the system under test, the most cost effective results may be achieved using relatively powerful test suites with relatively weak test oracles, or relatively weak test suites with relatively strong test oracles, or some combination in between. These results provide an impetus for exploring methods of oracle selection, particularly methods of oracle selection which pair test input with test oracles. These results also hint at possible methods of selecting oracle data. (Chapter 6.)

Impact of Program Structure on the Influence of Test Oracles: We empirically demonstrate how program structure can negatively impact the influence of test oracles. These results provide an additional example demonstrating the importance of considering factors in empirical studies, as well as provide evidence supporting the (non-trivial) development of tools for increasing observability in software systems. (Chapter 7.)

These results provide strong evidence that program structure, test oracles, and test data may interact to influence the effectiveness of the testing process. With respect to our research objective, this dissertation provides both a theoretical framework improving our ability to discuss interesting problems in software testing, and an empirical foundation quantifying how several key artifacts interact. This provides sufficient motivation for testing researchers to consider in future studies how

interactions between artifacts may impact their results. Furthermore, this dissertation identifies several questions that may lead to improvements in software testing. Notable questions include:

- For each structural coverage criterion explored, can we develop a canonical method (or methods) of structuring programs such that tests generated to satisfy these criterion will be effective?
- Can we develop new coverage criteria that allow us to construct small, but highly effective test sets (relative to existing methods of test input generation)?
- For a given test suite, how can we efficiently determine the most effective oracle for a given system?
- For a given system under test, how can we efficiently determine the most effective combination of test suite and oracle for a given system?
- How can we structure programs such that the test oracle will be very effective without unnecessarily increasing the cost of software testing?

1.4 Organization

This dissertation is organized as follows. In Chapter 2, we outline related work. In Chapter 3, we present our formal framework, and use it to provide a discussion of testing concepts, definitions, and the impact of previously ignored artifacts (notably test oracles) on existing theoretical testing work. In Chapter 4, we outline how our studies were conducted; results are discussed in subsequent chapters. We explore the impact of program structure on structural coverage criteria in Chapter 5. In Chapter 6, we explore the impact of oracle selection on testing effectiveness, in particular focusing on how oracle selection and test input selection interact. In our final

empirical chapter— Chapter 7—we explore how the program structure impacts the influence of test oracles. We conclude our dissertation in Chapter 8.

Chapter 2

Related Work

Testing research in general is a vast area of research that has been ongoing for several decades, producing voluminous work. Thus, while our research objective is aimed at improving the quality of testing research in general, a complete survey of all potentially affected or related work is well beyond the scope of our research. Instead, we will focus on two core areas of related work: the theory of testing, and empirical work on the effectiveness of testing.

2.1 Theory of Testing

It is difficult to precisely determine what work contributes to the "theory of testing"—work ranges from mathematical frameworks for describing testing, complete with proofs illustrating the implications of the framework, to formally defined, but intuitively based, "axioms" of test coverage criteria. We consider work on testing to be part of the theory of testing if it (1) describes the process of testing using a formalism, (2) defines concepts or definitions related to testing using a formalism, or (3) explores the effectiveness or limitations of testing techniques mathematically, without experimentation. This is a broad definition, but it captures work with descriptive power; that is, work that can be used to aid discussions of testing, or work that is often used as motivation in empirical work.

Of specific interest to us is work falling under (1) and (2). Exploring frame-

works for describing testing is thus the focus of the survey in this section, though work concerned with test set selection provides relevant background and is thus also explored.

We begin our discussion by exploring early work in the theory of testing, as this work provided the original ideas and definitions used in much of the later work. We then continue on to explore subfields of the theory of testing.

2.1.1 Beginnings of the Theory of Testing

The earliest work in the theory of testing was performed by Goodenough and Gerhart [26]. The theory proposed by the authors is based on test data selection and defines two properties a test data selection criterion should have. These properties are *reliability*, which states that a test satisfying the criterion will always produce the same results (either declaring the program correct or incorrect), and *validity*, which states that for every error in the program, there exists test data satisfying the criterion which will detect the error. Thus every test satisfying a reliable and valid test data selection criterion will detect all errors in a program.

Weyuker and Ostrand notes several flaws in Goodenough and Gerhart's work [73]. In particular, they show (1) that reliability and validity of a test data selection criterion depends on the program under test, (2) for any given program, *every* test data selection criterion is necessarily either reliable or valid, and (3) determining if a test data selection criterion is both reliable and valid requires knowledge of the errors in the program under test, information generally unavailable (and whose existence would make testing for the purpose of fault detection unnecessary). (Note with respect to point (2), the authors assume determinism.) Furthermore, Howden shows no computable methods can establish correctness in general [37], while Budd and Angluin show that for two notions of adequate test data, where the success of adequate test

data implies program correctness, generating or even recognizing adequate test data is not always possible [11]. Nevertheless, the general goal put forth by Goodenough and Gerhart—finding a test data selection criterion such that any test satisfying the criterion is guaranteed to detect faults—underlies much of the work in the theory of testing. In particular, this goal is often used in testing research to characterize what an "ideal" test set should accomplish.

Gourlay presents an influential mathematical framework for testing [27], and uses it to re-interpret previous work by Goodenough and Gerhart [26], Howden [37]. The core of this framework is Gourlay's definition of a *testing system*: a collection $< \mathcal{P}, \mathcal{S}, \mathcal{T}, corr, ok >$ where $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$ are arbitrary collections of programs, specifications and tests, respectively, and where the predicates *corr* and *ok* are defined as $corr \subseteq \mathcal{P} \times \mathcal{S}$ and $ok \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$, where $\forall p \in \mathcal{P}, \forall s \in \mathcal{S}, \forall t \in \mathcal{T}(\text{corr}(p, s) \Rightarrow \text{ok}(t, p, s))$. The predicate *corr* represents the correctness of a program p with respect to a specification s, while the predicate *ok* represents the test t performed on a program P is judged successful with respect to a specification s. As correctness of the program under test is generally not known during testing, *corr* is theoretical and used in discussions by Gourlay to relate testing to correctness. The predicate *ok* effectively acts as a test oracle, determining if an executed test is successful. A test set is a set of tests, i.e., $T \subseteq 2^{\mathcal{T}}$.

Gourlay then defines a *test method* as a function $M : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}$. This function represents a test coverage criterion (as a generator) in Gourlay's framework. Gourlay then explores how test methods can be evaluated with respect to their ability to detect faults and compared theoretically. He defines the *power* comparison for test methods, where a test method M is at least as *powerful* as a test method N ($M \geq N$) if any test set satisfying N finds a fault, any test set satisfying M will also find a fault. Gourlay then demonstrates that *reliability* and *power* are related (as well as other

formulations of test method effectiveness). Weiss transforms Gourlay's functional test method into a equivalently defined predicate: $T_C : P \times S \times 2^T$ (this states that every program, specification and set of tests maps to *true* if the coverage criterion is satisfied or *false* if not satisfied). [68]. This definition more closely matches how test coverage criteria are defined in practice and thus most subsequent work is based on this definition rather than Gourlay's generator version.

Gourlay's definition of a testing system is rarely explicitly used to describe testing techniques; nevertheless, his formalizations of the test set selection problem and the theoretical power of a testing method often are used in subsequent work in the theory of testing, as we will see in the remainder of this section.

2.1.2 Test Data Adequacy

As noted in the introduction, the effectiveness of testing techniques is often viewed as being determined by the test data used. Unsurprisingly, significant research effort has focused on how to select tests to meet testing objectives, a problem we term the *test set selection problem*. Many research efforts, both theoretical and empirical, have focused on formulating and evaluating objective metrics of test suite quality. These metrics are known by many names, including *test data adequacy criteria*, *test coverage metrics*, *test methods*, etc., and can be formulated equivalently as predicates or generators [83]. In the remainder of this work, we will refer to these metrics as test coverage criteria.

Coverage Criteria Axioms

Weyuker et al. propose eight axioms for test coverage criteria [70], later revising this set and extending it to eleven axioms [71]. The objective of these axioms is to identify a set of essential properties for test coverage criteria. Such a set of axioms can then be

used to check the adequacy of test coverage criteria. Weyuker then applies her axioms against common coverage criteria such as branch and statement coverage, and finds them lacking. While many of these axioms are fairly obvious, e.g., empty test sets should not satisfy a coverage criterion, some of the axioms provide definitions useful for discussion, including *monotonic* coverage criteria, which are coverage criteria such that if test set T satisfies coverage criterion C for program P, and $T \subseteq T'$, then T' does as well.

Parrish and Zweben further refine and analyze these axioms, noting several dependencies among the eleven properties and demonstrating the axioms can be reformulated to not depend on program structure [51]. The authors then note how the axioms can be reduced to seven axioms that are conceptually simpler. The authors continue this work [52], defending their choice of axioms (specifically one which contradicts monotonicity) arguing that different axioms may be used in different testing environments. The definition of a test coverage criterion is borrowed from Gourlay.

Zhu and Hall formally explore axioms of test coverage criteria, providing additional axioms, such as the axiom of *diminishing returns*, which states the contribution of a single test to satisfying a coverage criterion diminishes as the number of selected tests increases (a property common to most test coverage criterion) [82]. The authors also explore the implications of the existing axioms, proving the consistency of the general axioms proposed. Zhu later extends the axiom system to handle control flow graphs [80]. The definition of a test coverage criterion is again borrowed from Gourlay.

Work on axiom systems for test coverage criteria has not seen widespread adoption in that axioms of coverage criteria are not explicitly cited at motivations when developing new coverage criteria. The terms defined, however such as monotonic, do appear in current discussions of testing; such appearances demonstrate how theoreti-

cal work can be used to improve discussion concerning testing, even when such work is not directly referenced.

Partition Testing

Most test coverage criteria operate by defining a set of obligations a test suite must satisfy. Each obligation can be satisfied by zero or more test inputs; to satisfy the criteria, we must select at least one test input satisfying each obligation. The motivation behind such coverage criteria varied, but generally an intuitive argument is made that by satisfying each obligation, we will be less likely to fail to detect some class of faults. For example, in statement testing, we must execute every program statement, with the argument being that any statement left unexecuted may contain a fault.

This concept was first formalized by Weyuker and Ostrand [73] as *partition testing*. In partition testing, the input domain is divided into subdomains, and a test from each subdomain is selected. Once developed, the concept was used to demonstrate flaws in Goodenough and Gerhart's work discussed above.

Hamlet and Taylor theoretically explore the effectiveness of partition testing methods, and determine that partition testing can be comparable to random testing unless the partitions are chosen such that faults are concentrated in a partition [32]. The authors highlight the goal of constructing a test coverage criterion based on partition testing should be to construct *homogeneous* partitions in which all tests in the partition detect the same faults. Weyuker and Jeng, and Chen and Yu theoretically demonstrate similar results [72, 13, 14]. Later theoretical work by Gutjahr [29], however, provides a stronger case for partition testing.

While the theoretical contributions are interesting, the work on partition testing highlights how in the pursuit of one problem—in this case a method of demonstrating

flaws in existing work—theoretical work in testing can provide concise descriptions of testing practice and lead to interesting avenues of research. In this case, the concept of partition testing distils the intuition underlying virtually all test coverage criteria and led other researchers to highlight flaws in that intuition.

Comparing Test Coverage Criteria

Given two coverage criteria, an obvious question of practical interest is: *which coverage criterion is better?* This has led to a fairly large body of work, both theoretical and empirical, exploring the problem of comparing coverage criteria.

Perhaps the earliest method of theoretically comparing two coverage criteria is the *subsumes* relation, which states that a coverage criterion C_1 subsumes a coverage criterion C_2 if any test set satisfying C_1 also satisfies C_2. (We cannot determine the origin of this relation; even in very early work, it seems to be assumed as common knowledge.)

This relation says nothing of the effectiveness of the coverage criterion, however. Gourlay's power relation, outlined above in Section 2.1.1, is a relation for comparing coverage criteria based on their fault finding effectiveness [27]. A detailed analysis of the power relation, and other theoretical methods of comparing test coverage, by Weyuker, Weiss and Hamlet [74] note several desirable properties of comparison methods. The authors demonstrate that for coverage criteria C_1 and C_2, if C_1 subsumes C_2, then C_1 is at least as powerful as C_2 ($C_1 \geq C_2$). They also demonstrate the relation is one way, that is, power does not imply subsumption.

Having done this, the authors conclude both relations impart little certainty that one criterion is better than another because the comparisons tend to be *vacuous*—if $C_1 \geq C_2$, it generally means neither coverage criterion is guaranteed to find faults, not that C_1 is guaranteed to find additional faults. The authors then define the

PROBBETTER relation to probabilistically compare coverage criteria, stating C_1 is *PROBBETTER* than C_2 for some program P and specification S if test sets C_1 will on average find faults more often than test sets satisfying C_2. The probabilistic comparison better matches our intuition of what it means for a criterion to be "better", and does not fall into the trap of vacuous satisfaction.

Frankl and Weyuker propose several more comparison methods using the partition testing approach discussed previously [21]. Each of the relations is examined using several different measures of fault detection ability. The comparison methods are designed to provide increasingly better guarantees on fault detection improvement. Whereas subsumption provides few guarantees as to fault detection ability, as noted by Weyuker et al. previously, the most rigorous comparison method proposed—properly partitions—guarantees that a metric C_1 is more effective than C_2 for two different measures of fault finding effectiveness.

Zhu notes that if a tester generates tests in a *posterior testing scenario*—in which tests are generated using some deterministic method, and the coverage criterion is merely a stopping signal for generation—that subsumption does imply better fault finding [81].

Hierons explores how explicitly defining *test hypotheses* can improve the rigor of comparisons between coverage criteria [36]. Test hypotheses are properties believed to be true by the tester, i.e., assumptions made about the software under test. Using test hypotheses, we can introduce additional information into our comparisons of coverage criteria for a given program, or a class of program for which the property holds. Hierons notes that such hypotheses are often used in practice, as they are the intuitions that underly test selection (e.g., all faults can be found within 4 steps of execution).

The body of work again highlights the utility of theoretical testing research. In

each study the authors begin with a practical problem, define precise terminology to discuss it, formalize both current practice (e.g., subsumes) as well as ideal solutions (e.g., power), and proceed to explore the implications. Their results indicate that the existing comparisons—subsumption and power—do not yield useful, practical methods of comparing test coverage criteria. The authors then explore alternatives, with perhaps the most promising being probabilistic comparisons of coverage criteria. Given the difficultly of rigorously measuring the probability of detecting a fault with a given coverage criterion, this implies that empirical studies must be undertaken to compare test coverage criteria. Indeed, most work subsequent to these theoretical studies focuses on empirical comparisons.

2.1.3 Gaudel-Bernot Theory of Testing

Most of the work outlined above is based on Gourlay's formal framework, either implicitly (in that it is based on a functional view of testing) or explicitly. Nevertheless, other frameworks exist.

Gaudel and Bernot have developed a theory of testing based on formal specifications [24, 6, 7]. In this work, they define a *testing context* as a triple (H, T, O) where H is a set of testing hypotheses (i.e., assumptions) about the program and specification, T is a set of tests, and O is a test oracle. This body of work uses algebraic specifications as the basis of testing; tests are created based on the axioms of the algebra that define the program interface.

This body of work is notable in that (1) the intuitions that are often used to guide the testing process are made explicit as test hypotheses and (2) the problem of selecting an oracle is a core part of the framework and discussion, unlike work based on Gourlay's framework in which the test oracle is generally assumed.

The authors argue the use of testing hypotheses are beneficial in two ways. First,

the authors note these assumptions often underly the definition of test coverage criteria. By making these assumptions explicit, we make clear the intuitions that underly our testing processes. Second, if the testing hypotheses can be proved, then the testing can act in a manner similar to an inductive proof—we begin by proving that some finite set of tests is sufficient to show correctness (e.g., all tests of length up to three), and then demonstrate our tests do not detect any faults. This draws a connection between formal methods and testing.

The authors also discuss the difficulty of defining test oracles. The concept of *oracle hypotheses*, similar to the concept of testing hypotheses is defined and explored. The authors note that certain parts of the program necessary to ascertain correctness are not always observable, e.g., "opaque" type equality and internal component state [24].

This work presents a number of interesting ideas, one of which—the explicit discussion of test oracles—addresses an issue we have with Gourlay's framework. However, the use of formal specifications based on algebraic sorts leads to a cumbersome formalization. (The use of algebraic sorts also limits the applicability.) Such a formalization is not a good candidate for improving testing discourse; while more precise than natural language, it is also difficult to understand and to relate to testing in practice.

2.1.4 Behavior Observation Schemes

Zhu and He have proposed a testing theory based on *behavior observation schemes* [84]. In this theory, a *scheme* denotes a systematic method of observing dynamic behaviors, e.g., statements covered, states explored, etc. This theory is notable in that it allows for non-determinism, i.e., tests whose outcome can change between executions. Zhu and He have redefined several existing coverage criteria in the theory [85]

and have also defined a theory of testing Petri nets [86]. The theory encompasses a number of existing contributions in the theory of testing, including test axioms, and test coverage criteria. Thus, it can be viewed as method of defining and comparing coverage metrics.

This body of work is aimed at defining coverage criteria; it is not as general a theory as Gourlay's or Gaudel/Bernot's and, thus, is not suited for facilitating discussion of testing in general.

2.1.5 Other Theories of Testing

There exist a handful of other theories of testing, none of which have received significant attention. Howden introduces a theory of functional testing in [39], which views programs as the synthesis of functions. This work is (by its own admission) fairly informal. Hamlet presents an outline of a theory of software dependability in [31], arguing that the foundations of software testing should be statistical in nature. Morell introduces a theory of fault-based testing in [48], in which the goal of testing is to show the absence of a certain set of faults. None of the above theories appear to provide anything that might improve our discussion of testing beyond what appears in Gourlay's more commonly used framework.

2.1.6 Theory of Testing Conclusions

As discussed, work in the theory of testing has provided a conceptual framework for testing research, and has produced or at least formalized influential ideas such as test coverage criteria, partition testing, and axioms of testing. However, this work has largely neglected to explore concepts unrelated to selecting test inputs. We believe this has led to a mental model of testing in which test inputs are considered almost exclusively, with very little work being done on other factors—notably test oracles—

that can impact the effectiveness of testing. Indeed, while there is a rich vocabulary in which to discuss selecting test inputs, properties of test suites, etc., basic properties and concepts have yet to be defined for test oracles. In Chapter 3, we begin address these oversights chiefly by extending Gourlay's concept of a testing system to more accurately capture testing in practice.

2.2 Empirical Studies on the Effectiveness of Testing

Empirical work on the effectiveness of testing is an extensive area, and it is beyond the scope of this report to survey it all. We are primarily interested work that considers how testing artifacts interact, and how this interaction may be used to improve the testing process, as our proposed study will explore the same general problem. Given that the motivation for our work stems from a desire to improve the quality of testing research, we are also interested in influential empirical work that does *not* consider how testing artifacts interact, as the conclusions of such work may be impacted by the results of our proposed study.

In some sense, all testing research necessarily explores the relationship between testing artifacts, as testing cannot be accomplished using a single artifact. However, little work explicitly explores how testing artifacts interact, e.g., *how does one testing artifact influence another?*, *how do combinations of testing artifacts affect testing effectiveness?*.

2.2.1 Test Coverage Criteria

As noted previously, work on test coverage criteria is a popular topic of study, encompassing dozens of criteria and hundreds of papers. As noted by Zhu in his extensive survey of the area [83], test coverage criteria are often specified in terms of some other testing artifact. Notably, Zhu identifies two broad areas of coverage criteria,

specification-based coverage criteria, and *program-based* coverage criteria.

In specification-based coverage criteria, test adequacy is judged using the program specification or requirements. These coverage criteria tend to be tuned to a specific form of requirements, and thus can vary significantly in their details. For example, in our own work we have defined requirements coverage metrics over Linear Temporal Logic (LTL) properties [75], while Amla and Ammann have demonstrated how the input space can be partitioned using Z specifications [1].

In program-based coverage criteria, test adequacy is judged using the the program under test. A common form of program-based coverage criteria are *structural* coverage criteria, in which the coverage criterion specifies that certain syntactic constructs must be exercised by the test set, e.g., all branches covered or all statements executed. Another common class of program-based coverage criteria is *mutation* coverage criteria, in which single faults are intentionally seeded into the original program to create multiple faulty versions of the program (i.e., *mutants*) and a test sets is constructed to differentiate the mutants from the original program.

The common thread between program-based and specification-based coverage criteria is that both judge the adequacy of tests based on another testing artifact, either the specification or the program. Clearly, one artifact influences another. However, while this is clear from the construction of these criteria, it is not widely discussed, and most evaluations of coverage criteria do not consider it. In such evaluations the reader is left with only an intuitive notion of how the coverage criterion is influenced by the construction of the specification or program.

However, Rajan et al. do explore the effect of program structure on the Modified Condition/Decision Coverage (MCDC) criterion [55, 16]. In this study, for each program studied, an "inlined" version containing complex conditions and a "noninlined" version containing simpler conditions are used in conjunction with the MCDC cover-

age criteria. The authors find that the inlined version requires more tests to satisfy MCDC as compared to the noninlined version.

2.2.2 Test Oracles

As noted by Baresi and Young, test oracles represent a relatively small portion of software testing research [4]. The authors outline a number of approaches for deriving automated test oracles, almost all of which are based on deriving the test oracle from a formal specification. Such approaches explicitly link the oracle and the specifications. This link is fairly obvious—a test oracle determines if the software under test is correct. Given a formal notion of correctness in the form of a specification, it makes sense to define the oracle using the specification. As with test coverage criteria based on specifications, the specification clearly influences the test oracle. Unfortunately, it does not appear that this impact is explored in the existing work.

Test oracles can also be influenced by other testing artifacts, including test cases and the program (specifically the structure). Memon et al. have performed an empirical study using GUI applications in which both the test set and test oracle are varied [47]. The authors demonstrate that when testing GUI applications, test oracles considering more information can significantly improve the effectiveness of testing when using short tests or a small number of tests. The authors also conclude that when using longer tests or a larger number of tests, it may be that sophisticated test oracles are not cost-effective, and that simpler, cheaper test oracles may be used with little loss in testing effectiveness.

Fraser and Zeller [22] use mutation testing to generate both test inputs and test oracles. The test inputs are generated first and then assertions capable of distinguishing the mutants from the program, with respect to the test inputs, are created. However, little discussion exists concerning the impact of the test oracles with most

of the work focusing on how test oracles can created.

The *observability* of the program is the degree to which we can observe the program's state. The degree of observability present in a program places restrictions on the construction of test oracles—if some variable is not observable, we cannot use it to determine the correctness of the system during testing. This is discussed by Gaudel [24] (outlined in Section 2.1.3) and in general this seems to be a recurring issue in test oracles based on formal specification [4]. The issue is that when using formal specifications as an oracle, abstractions may be defined such that relating the specification to the program under test is a non-trivial task.

Testability is the ease with which software can be tested. (Note multiple definitions of testability exist; in some definitions testability includes observability as a property [77].) It has been proposed to use testability information to direct test oracle generation, observing internal program variables/locations that are difficult to test otherwise. We are not aware of any rigorous studies exploring this idea.

2.2.3 The PIE Technique

Voas, Miller, and other collaborators have produced an extensive body of work based on the Propagation, Infection, and Execution (PIE) analysis technique [65, 64]. This technique is designed to determine the probability of faults at each program location propagating to the output, termed the *testability* information for the program. It is suggested that this testability information can be used (among other things) to direct testing efforts, thus using the characteristics of the system under test to guide both what tests are selected [66] and the construction of assertions [67] (a form of test oracles). The general intuition is the less likely a location is to reveal faults, the more effort should be focused on the location, either through additional testing, or direct observation (through assertions) of the location.

Although the intuition seems reasonable—use the characteristics of the program to inform testing decisions—empirical evaluation is sparse. Chen et al. [15] evaluate the effectiveness of using fault-exposure-potential estimates (metrics similar to those defined by the PIE method) to improve the statement coverage criterion. Results were mixed, as statistically significant improvements were found, but these improvements were quite small.

2.2.4 Example of Potential Issues Empirical Studies

We noted in the introduction that existing empirical studies often do not consider how testing artifacts can interact, and that this may lead to conclusions that are misleading or possibly incorrect. Here, we illustrate this problem using two studies on test suite reduction, and discuss the potential implications of failing to consider these interactions. We view these studies as influential, as they are both viewed positively by the research community (being landmark papers within the area) and widely cited. We also view these studies to be, for the most part, well conducted. They are merely used to highlight general and systemic problems that exist in testing research.

Wong et al. explore the effect of test suite reduction on fault finding effectiveness [76]. 1,000 random tests are generated and test sets are minimized to meet varying levels of block coverage. Ten Unix programs are used. Faults are seeded into each program to produce a set of faulty programs, with the original program serving as a test oracle. Tests are run to distinguish the original program from the faulty programs. Faults are classified according to difficulty, with the difficulty determined by the number of tests catching the fault. The authors conclude that test suite reduction does not significantly impact the fault finding effectiveness of the testing process.

In this study, the authors have varied only the program. The same oracle is used

in each case example, and the same coverage metric is used to reduce the test suites. This presents two questions: *how does the use of the oracle influence the results?*, and *how does the use of block coverage influence the results?* The authors address neither question, though they do acknowledge their results apply only to the single coverage criterion used. The former question, however, is not acknowledged at all—the reader is left to infer how test suite reduction would, for example, influence the effectiveness of a testing process using assertion-based oracles.

Rothermel et al. revisit the problem of test suite reduction, again studying its influence on the fault finding effectiveness of the process [59]. The authors are aware of the previous study by Wong et al., and cite several open questions. The authors then perform a similar study, using seven programs differing from the Wong study (the commonly used Siemens programs [40]), edge coverage for reducing the test suite, tests provided with the Siemens programs, and faults provided with the Siemens programs. A large number of test suites of varying size are constructed for each program. The test suites are then reduced according to edge coverage 1,000 times per program. Each test suite is run against each fault version, using the original program as an oracle, and the number of faults detected are measured. The authors conclude (in contrast to the Wong study) that test suite reduction negatively impacts the fault finding effectiveness of the testing process.

In this study, the authors have varied the program and the original (unreduced) test suite size. The same oracle is used in each example and the same coverage metric is used to reduce each test suite. This presents the same two questions from the Wong study: how does the use of the oracle influence the results, and how does the use of edge coverage influence the results? The authors clearly acknowledge the latter issue in their threats to validity, but again fail to acknowledge the former question.

This presents a problem: in two influential empirical studies in the area of test

suite reduction, the authors entirely omit discussion of how one testing artifact, the oracle, influences the results. By selecting a single metric and acknowledging the limitations in the discussion of the results, one can argue the test coverage metric used is controlled. However, by not acknowledging the oracle used, it is left to the reader to infer the influence of the oracle on test suite reduction, which can lead to misleading results.

2.2.5 Empirical Study Conclusions

Work exploring how testing artifacts can interact is clearly sparse, with only a handful of studies addressing the issues. Few of these studies use case examples from our domain of interest, critical systems; consequently, we have very little understanding of even basic issues related to artifact interaction within this domain. Potential issues relevant to critical systems that are currently unexplored include: the impact of test oracles on the testing process, particularly interactions with test coverage criteria; the impact of program structure on the effectiveness of test suites satisfying structural coverage criteria; and the impact of program structure on test oracle construction.

As demonstrated in the previous section, this lack of knowledge represents a potential problem in our empirical studies: without an understanding of how these artifacts interact, we may produce misleading conclusions. Furthermore, by not studying how these interactions influence testing effectiveness, we deny ourselves some of the potential benefits of other (non-test input selection related) avenues of improving testing effectiveness, such as test oracle selection (as performed for Java programs by Fraser and Zeller [22]).

Chapter 3

Theory of Testing Revisited

In this chapter, we outline our work on the theory of testing. We begin by revisiting our motivations for expanding the theory of testing, briefly outlined in Chapter 1. We then present two formalisms for describing testing, one extending the functional framework of Gourlay [27], and one based on Kripke structures [44], defining testing concepts using both formalisms. We then explore the theoretical implications of our extension of Gourlay's work. Finally, we conclude by highlighting how our results demonstrate the need to consider how various testing artifacts influence the effectiveness of testing.

Most of the work here has already been published in [61]. This work differs primarily in the presentation of the Kripke structure-based formalism. It has also been expanded for clarity.

3.1 Motivation

In the previous chapter, we considered several foundational works on the theory of testing. While these early contributions are valuable and have helped shape our understanding of testing practice, this body of work has unfortunately not established itself as a foundation for continued testing research; new testing approaches are typically informally described and, in general, poorly evaluated. This lack of a formal foundation and rigorous evaluation of proposed new approaches has been a persistent

problem in the research community [8].

Consider as examples the generally well conducted and highly influential studies of Rothermel et al. [58], and Wong et al. [76], previously discussed in [**?**]. Although these studies provide insight into test suite reduction, crucial aspects of the experimental setup such as what test oracle was used and the nature and structure of the programs under test are omitted or left implicit, leaving the reader to infer these properties of the artifacts. These implicit artifacts may explain the conflicting conclusions reached in the studies. We believe these problems can be traced to the lack of a common foundation for empirical testing research, making it difficult—if not impossible—to conduct, for example, meta-analysis to synthesize the empirical results from several independent investigations.

In this chapter, we attempt to remedy this situation and provide a common framework for empirical testing research by revisiting work on the formal foundations of testing. We have identified two issues with the existing formalizations that we believe should be addressed. First, the existing formalizations overlook certain factors influencing testing—notably test oracles—leading to implicit assumptions about testing that may not be true in practice. These assumptions make it straightforward to prove properties about different aspects of testing, for example, properties about test coverage criteria, but also lead to research results that may be misleading or may not be generally applicable. In addition, such implicit assumptions makes comparisons of research efforts difficult. We therefore extend the existing formalisms to account for one testing artifact that is commonly overlooked, the test oracle, and define several concepts related to test oracles.

Second, most foundational research has focused on narrow aspects of the testing problem, for example, criteria for selecting tests or techniques to generate tests from programs. As noted in Chapter 1, we believe a holistic view of the testing problem

is essential to move the field forward and yield practically useful results in testing research, both in terms of theoretical and empirical results. We therefore explore how the interaction of artifacts can in theory influence the effectiveness of the testing process.

3.2 Functional Model of Testing

Beginning with Goodenough and Gerhart's seminal work [26], a significant portion of the research in the theory of testing has used a functional model for testing, a convention we follow here. We define our functional model of testing based on Gourlay's framework [27], extending and modifying it for our discussion. We have selected Gourlay's framework as a basis for two reasons. First, a significant quantity of relevant theoretical work is based on this formalization; by extending his framework, we can easily reexamine this previous work. Second, the framework is easy to understand and mostly matches our intuitive sense of the testing process.

In Gourlay's approach, a *testing system* is defined as a collection $< \mathcal{P}, \mathcal{S}, \mathcal{T}, corr, ok >$ where:

- $\mathcal{S}$ is a set of specifications
- $\mathcal{P}$ is a set of programs
- $\mathcal{T}$ is a set of tests
- $corr \subseteq \mathcal{P} \times \mathcal{S}$
- $ok \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$

Each specification $s \in \mathcal{S}$ represents an abstract, perfect notion of correctness. The predicate *corr* is defined such that for $p \in \mathcal{P}$, $s \in \mathcal{S}$, $corr(p, s)$ implies p is correct

with respect to s. Of course, the value of $corr(p, s)$ is generally not known; this predicate is thus theoretical and used to explore how testing relates to correctness. The predicate ok is defined such that for $p \in \mathcal{P}, s \in \mathcal{S}, t \in \mathcal{T}$ $ok(t, p, s)$ implies that p is judged as correct with respect to specification s for test t. Furthermore, ok is defined such that $\forall p \in \mathcal{P}, \forall s \in \mathcal{S}, \forall t \in \mathcal{T}$ $corr(p, s) \Rightarrow ok(t, p, s)$, i.e., if p is correct with respect to s then ok is true for all tests. The predicate ok approximately corresponds to what is now called a *test oracle* or simply *oracle*.

While intuitively appealing, there are problems with this framework. First, each testing system has only one possible oracle (ok). Just as there exist many possible tests and programs, however, there exist many possible oracles for determining if test executions are successful [57]. Selecting an oracle is the *oracle selection* problem, and we cannot easily discuss or even formulate this problem using Gourlay's framework.

Second, the notion of correctness and how it relates to test oracles is—in our opinion—too coarse. If for $p \in \mathcal{P}$ and $s \in \mathcal{S}$, if $corr(p, s)$ then we know that $\forall t \in \mathcal{T},\ ok(t, p, s)$. However, there are no requirements on oracles in terms of their *effectiveness* in finding faults. For example, the oracle that universally returns true for all programs and specifications satisfies this relationship. Furthermore, it will generally be the case that the program p does not satisfy the specification s (otherwise, why test?), i.e., $\neg corr(p, s)$, in which case the framework places no constraints on ok.

Both the inability to discuss oracle selection, and the loosely specified relationship between program correctness and oracle behavior create difficulties and ambiguities when discussing the effectiveness of test selection techniques and test oracles. We therefore make two major changes to Gourlay's definition of a testing system. First, we remove the predicate ok, replacing it with the set $\mathcal{O}$ of test oracles. We state that an oracle $o \in \mathcal{O}$ is a predicate:

$$o \subseteq \mathcal{T} \times \mathcal{P}$$

An oracle determines, for a given program and test if the test passes. Second, we add a predicate defining correctness with respect to a test $t \in \mathcal{T}$. This predicate is:

$$corr_t \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$$

(Note that the t subscript in $corr_t$ is used to differentiate the predicate from Gourlay's *corr* predicate.) The predicate $corr_t(t, p, s)$ holds if and only if the specification s holds for program p when it runs test t. Thus

$$\forall p \in \mathcal{P}, \forall s \in \mathcal{S}, \quad corr(p, s) \Rightarrow \forall t \in \mathcal{T}\ corr_t(t, p, s)$$

In summation, we define a testing system to be a collection $(\mathcal{P}, \mathcal{S}, \mathcal{T}, \mathcal{O}, corr, corr_t)$ where:

- $\mathcal{S}$ is a set of specifications
- $\mathcal{P}$ is a set of programs
- $\mathcal{T}$ is a set of tests
- $\mathcal{O}$ is a set of oracles
- $corr \subseteq \mathcal{P} \times \mathcal{S}$
- $corr_t \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$

To keep things general, we make no attempt to define what exactly constitutes a test, oracle, specification, or program. We state that a test (sometimes called test data) is a sequence of inputs accepted by some program. As in Gourlay's framework, we consider a specification $s \in \mathcal{S}$ to be the true (idealized) specification of the desired functionality of program $\mathcal{P}$, possibly including internal state behavior. It is quite likely that the stated software requirements or formal specifications used in the development of a program differ from s. Finally, we note that the predicates are partially defined: not all tests can be executed on all programs, and not all oracles can be used to determine if a test t is successful when run against a program p.

These modifications to the framework allows us to have a more realistic discussion of the testing problem and explore the interrelationships between programs, tests, and oracles.

3.2.1 Test Oracles

A test oracle determines if the result of executing a program p using a test t is correct. There are many methods of creating an oracle, including manually specifying expected outputs for each test, monitoring user-defined assertions during test execution, and verifying if the outputs match those produced by some reference implementation such as an executable model. A uniform method of describing the numerous types of oracles is outside the scope of this work.

Nevertheless, we can define general oracle properties. We begin by defining oracle properties related to correctness of the program being tested, borrowing terms commonly used in software verification. An oracle is *complete* with respect to program p and specification s for a test case t if:

$$corr_t(t, p, s) \implies o(t, p)$$

Complete oracles relate to correctness as we intuitively expect: if the result of running t over p is correct with respect to s, the oracle o will state the test passes. Most oracles discussed in testing research and used in practice are designed to be complete, though like all software engineering artifacts, oracles are imperfect and may contain flaws. For example, a common problem is an oracle that is too precise. The oracle may have been defined to expect an output of $1\frac{1}{3}$ but the program generates 1.3334. In the application domain, this accuracy in the computation is perfectly fine and the program is thus correct, but the oracle will reject the test. Nevertheless, in order to discuss the efficacy of the testing process, researchers often assume the

oracle used is complete. Note that Gourlay's *ok* predicate is by definition complete if $corr(p, s)$.

An oracle is *sound* with respect to p and s for a test case t if:

$$o(t, p) \implies corr_t(t, p, s)$$

Sound oracles represent the conventional wisdom in testing that *"testing may be imperfect, but at least we know that the program is correct for the tests we have run."* For this statement to hold, we must use a sound oracle. Unfortunately, in practice, oracles are rarely sound. For example, an oracle might only observe a subset of the program outputs (and/or the program's persistent state) and would naturally miss any faults manifested in the variables (or state) not observed by the oracle. For this reason, we do not assume sound oracles in our discussion.

We say that an oracle is *perfect* with respect to p, s, and t if it is both *sound* and *complete*. We can now generalize the definitions of *complete*, *sound* and *perfect* to test suites and again to the entire set of tests. For example, an oracle is *perfect* for p and s if:

$$\forall t, o(t, p) \Leftrightarrow corr_t(t, p, s).$$

As mentioned above, oracles need not be sound nor complete; oracles may both fail to detect faults and may report faults that do not exist. Heuristic oracles may be neither sound nor complete, and may be used in domains like image processing where precisely defining correctness is difficult or time consuming [46]. Relating this type of oracles to correctness would require probabilistic arguments and is beyond the scope of this work. Weyuker informally discusses these oracle characteristics in [69], noting several practical challenges concerning test oracle construction; we are unaware of any formulation of these oracle properties however.

In this work, we will often consider oracles which base correctness partly on the internal state of the program. Such oracles may be constructed if the specification

defines behaviors internal to the program (e.g., state invariants or class invariants). In some situations, these oracles will detect faults that do not propagate to the output (at least not immediately). We use Avizienis and Laprie's terminology [3], in which a fault is defined as a system state where a design error manifests. Thus, detecting a *fault* is not synonymous with detecting a *failure*.

Highly related to our discussions in this section, Richardson et al. discuss the *oracle problem*—the need for testers to provide a test oracle for the testing process [57]. This work has served as the standard for oracle terminology; the authors define the *oracle information* and the *oracle procedure*. The oracle information specifies the correct behavior, and the oracle procedure verifies the test execution with respect to the oracle information.

We consider the issues of oracle information and oracle procedure to be specific to the method of oracle construction. Defining precisely what constitutes oracle information and what constitutes oracle procedure is difficult, and we therefore make no attempt to incorporate them into our framework. We instead opt to use definitions that are useful in discussing all oracles (such as *complete*).

3.2.2 Adequacy of the Testing Process

One common task in software testing is determining when we have tested enough. Methods of determining the adequacy of the testing process are termed *test adequacy criteria*. Exploring the effectiveness of these criteria is a key task in testing research, and forms the basis of much of the later work in Chapter 4. Furthermore, formalizing the concept of test adequacy is a common goal in the theory of testing. We therefore formalize these concepts here, beginning with test adequacy.

Gourlay defines a *test method* M as a function:

$$M : \mathcal{P} \times \mathcal{S} \to \mathcal{T}$$

Thus, a test method takes a program and a specification and generates a test. Gourlay appears to also consider test methods

$$M : \mathcal{P} \times \mathcal{S} \rightarrow 2^{\mathcal{T}};$$

that is, test methods producing sets of tests. Nevertheless, the use of *test coverage criteria* (also called *test data adequacy criteria* [70] or *test selection criteria* [26]) where zero or more of the test sets are acceptable for a given program and specification is much more common in both the testing literature and in practice. We thus adopt it here, using the predicate definition originally presented by Weiss [68]:

$$T_C \subseteq \mathcal{P} \times \mathcal{S} \times 2^{\mathcal{T}}.$$

However, in this work, we also explore how test oracles influence the testing process. We therefore propose an analogous concept for oracles, termed an *oracle adequacy criterion*. An oracle adequacy criterion O_C is a predicate:

$$O_C \subseteq \mathcal{P} \times \mathcal{S} \times \mathcal{O}.$$

This predicate reflects how oracle selection is usually done in practice: a single oracle is used to evaluate the result of every test. Most testing approaches used in practice or described in the testing literature can be described using T_C and O_C. However, it is possible to define adequacy of the testing process in terms of both the test set and the test oracle used, i.e., define adequacy as a pairing of a test set and an oracle. We define a *complete adequacy criterion* as the following predicate:

$$TO_C \subseteq \mathcal{P} \times \mathcal{S} \times 2^{\mathcal{T}} \times \mathcal{O}$$

For example, a stateful program responsible for mode switching in an avionics systems may be best combined with a test suite providing MCDC coverage and an oracle observing a majority of the internal state variables in addition to all outputs. In Section 3.5, we will explore an existing example of a complete adequacy criterion.

3.3 Computational Model of Testing

The extension outlined in Section 3.2 provides a simple abstract framework in which to discuss testing. It captures the core problem facing testers – the need to select, from very large or infinite sets of tests and oracles, a subset of tests and an oracle. However, it abstracts away the factors underlying that decision, including the nature of the system under test and the observability of the system's states. By considering such factors in our formalisms, we can sometimes more precisely define testing concepts. In this section, we use Kripke structures [18], a well known formalism for describing computation, to formalize several testing concepts in terms of the system under test.

3.3.1 Altitude Switch Example

To examine how our computational framework can be used to explore testing problems, we present a small case example in Figure 3.1; an *Altitude Switch*. This system turns power on to a Device Of Interest (DOI) when the aircraft descends below 1,000 meters. If the altitude cannot be determined for more than three seconds, the ASW turns an alarm on. The user can manually inhibit the device, disabling the alarm functionality and preventing the ASW from controlling the DOI.

3.3.2 Kripke Structures in Testing

Let AP be a set of atomic propositions. A Kripke structure is a 4-tuple $M = (S, I, R, L)$ where:

- S is a set of states
- $I \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is transition relation between states
- $L : S \to 2^{AP}$ is a labeling function for states

```
0   input: alt: int
1   input: altQuality: bool
2   input: inhibit: bool

3   var: badTime : int = 0
4   var: prevAlt : int = -1
5   var: alarm : bool = false
6   var: doiOn : bool = false

7   while (true):
8     input(alt, altQuality, inhibit)
9     if !inhibit:
10      if altQuality:
11        if prevAlt > 1000 && alt < 1000:
12          doiOn = true
13        elif prevAlt < 1000 && alt > 1000:
14          doiOn = false
15        alarm = false

16      if !altQuality && !alarm:
17        badTime += 1

18      if badTime > 3:
19        alarm = true
20        badTime = 0

21      if altQuality:
22        prevAlt = alt
23    else:
24      badTime = 0
25      alarm = false
26    output(alarm, doiOn)
```

Figure 3.1: ASW Case Example

Kripke structures are commonly used in model checking to represent system behavior [18]. In this context, a Kripke structure is effectively a labelled graph where

each state corresponds to a reachable system state, and each edge represents a state transition. Note while each system state corresponds to a state in the corresponding Kripke structure, the labels contain the usable information about the state.

For the purposes of our discussion, we define the atomic predicates AP in terms of a function mapping from variable labels $\mathcal{L}$ to values $\mathcal{V}$. We therefore extend the basic Kripke structure to a 6-tuple: $M = (S, I, R, \mathcal{L}, \mathcal{V}, L)$ where

- S is a set of states
- $I \subseteq S$ is a set of initial states
- $R \subseteq S \times S$ is transition relation between states
- $\mathcal{L}$ is a set of variable identifiers
- $\mathcal{V}$ is a set of variable values
- $L : S \to (\mathcal{L} \to \mathcal{V})$ is a labelling function for states

This extended Kripke structure is isomorphic to a standard Kripke structure (for finite types) through an encoding function to map between assignments to variables to Boolean propositions [18]. However, it is much more straightforward for our discussion to think in terms of assignments to variables rather than atomic propositions.

3.3.3 Tests and Oracles

Recall from Section 3.2 that a testing system is a collection:

$$< P, S, T, O, corr, corr_t >$$

In this computational model of testing, every program $p \in P$ defines a Kripke structure. For a program p, each test $t \in T$ that can be run over p corresponds to a finite path beginning at the initial state. Each oracle $o \in O$ is thus a decision procedure relating a Kripke structure to a finite path through the Kripke structure.

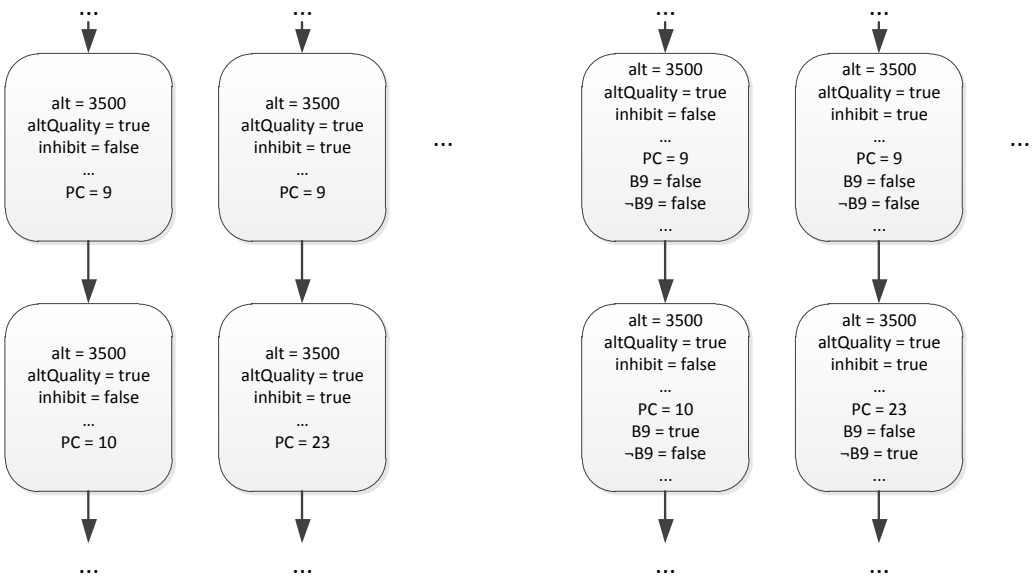
The problem of test selection is then effectively the problem of selecting a finite number of finite paths starting from initial states within a Kripke structure. The problem of oracle selection is partitioning the set of finite paths into *True* and *False* test outcomes.

Test Selection

We have described Kripke structures in terms of the the values of variables within a program—the semantics of the program—rather than the program's syntactic information. Many test coverage criteria, however, are defined in terms of the syntax of the program, which is not represented within the Kripke structure. We can easily remedy this by extending $\mathcal{L}$ to include the information necessary to determine coverage. Let $C = \{c_1, c_2, \ldots, c_n\}$ be a set of Boolean variables representing the syntactic obligations of the criteria, and let $\mathcal{L}_V$ be the set of atomic propositions containing predicates representing the system state. Then $\mathcal{L} = L_V \cup C$, and we assign each variable in C *True* in the state(s) in which the obligation is satisfied.

Consider the problem of test selection for the ASW. We make two observations concerning the ASW's corresponding Kripke structure. First, even if we assume the range of integers is finite but large, e.g., 2^{64}, the number of system states is too large to reasonably explore them all. Second, the *while(true)* loop implies that infinite (or at least unbounded) paths are possible. Thus the tester must select a small number of tests from an infinite set of possibilities.

Suppose we wish to discuss branch coverage. We can extend AP with a set of coverage predicates representing each branch evaluating to *True* and *False*, $C = \{B_9, \neg B_9, B_{10}, \neg B_{10}, \ldots\}$, where B_x represents the branch on line X in the ASW program. Consider the snippet of Kripke structure in Figure 3.2 (a) representing execution of line 9. The top two states differ in the value of *inhibit*, and thus lead to



(a) ASW Kripke Structure (b) ASW Kripke Structure w/ Branch Metrics

Figure 3.2: Snippet of Kripke Structure Representing ASW Example

different states after executing line 9. Extending $\mathcal{L}$ with C, we can then label states where each branch is taken, yielding Figure 3.2 (b). To satisfy branch coverage, we must select a set of paths P such that each $c \in C$ is assigned to a state in at least one $p \in P$.

Oracle Selection

While in principle simply partitioning the set of finite paths into *True* and *False* test outcomes, the problem of oracle selection is much more complex in practice, as noted in Section 3.2.1. Many factors in oracle selection are dependent on the method underlying the construction of the oracle. For example, several sources may be used to determine correctness, including formal specifications, informal design documents

or even human inputs. These factors cannot be described using Kripke structures (or our functional framework).

One factor present in all oracles, however, is the portion of the system state considered by the oracle, which we term the *oracle data* [57]. Formally, the oracle data is $OD \subseteq \mathcal{L}$, i.e., the subset of variables considered by the oracle. This can range from oracles which consider a single variable to oracles which consider every variable (i.e., consider the entire system state).

In both our theoretical and empirical work, we will primarily consider oracles that operate by comparing values produced by the program for some test against expected values described in the test case. We will refer to such oracles as *expected value oracles*. When presenting or evaluating several oracles for the same system, these oracles will typically differ in their oracle data.

To illustrate the concept of oracle data, assume we are testing the example system and plan to use an expected value oracle. Now suppose we have narrowed our oracle data choices down to two possible sets, OO and IV. The oracle using the OO oracle data considers only the output variables, while the oracle using the IV oracle data considers both outputs and internal variables. Formally, the oracle data for OO would be OO = {alarm, doiOn} and for IV would be $IV = OO \cup$ { badTime, prevAlt }. Both oracles will, for a given test (path), examine each state in the path and determine if the predicates in the state match the predicates expected by the oracle, ignoring predicates not in the oracle data.

The decision as to which oracle data to use depends on several factors, including the observability of the internal variables, the cost of considering more of the system state, and the expected fault finding gains in the testing process. We explore these issues in Section 3.5.3, and empirically study the influence of varying the oracle data in later chapters.

3.4 Oracle Comparisons

By extending Gourlay's framework with a set $\mathcal{O}$ of oracles, we have introduced the problem of oracle selection: the problem of selecting an oracle o from a set of possible oracles. Just as with the problem of test selection, we desire some method of estimating the relative usefulness of oracles. Unfortunately, we are unaware of any comparison relations specific to oracles (though mutation testing represents a method of comparing combinations of test inputs and test oracles; see Section 3.5.2). To facilitate such comparisons, we present several possible oracle comparison relations, based on the test coverage criteria comparison relations explored by several authors [27, 74] and discussed in the following section.

Our oracle comparisons, like test coverage criteria comparisons, are based on the ability of the oracles to detect faults. Recall that an oracle is *complete* with respect to p and s if for all tests t:

$$corr_t(t, p, s) \implies o(t, p)$$

In other words, when a fault is detected by o, the fault is real, and therefore represents an error in the program. As most oracles discussed in testing research are designed to be complete, the oracle comparisons we present assume complete oracles. Comparisons between non-complete oracles would require a different approach accounting for oracles signaling faults when none have occurred.

3.4.1 Power Comparison

Our first relation is based on Gourlay's definition of *power* [27]. We state an oracle o_1 has a power greater than or equal to oracle o_2 with respect to a test set TS (written $o_1 \geq_{TS} o_2$) for program p and specification s if:

$$\forall t \in TS, o_1(t, p) \Rightarrow o_2(t, p)$$

In other words, if o_1 fails to detect a fault for some test, then so does o_2. If $o_1 \geq_{TS} o_2$, it is possible that o_1 and o_2 are equally powerful, i.e.,

$$\forall t \in TS, o_1(t,p) \Leftrightarrow o_2(t,p).$$

We may wish to state that some oracle o_1 is strictly better than an oracle o_2. We state that o_1 is more powerful than o_2 for test set TS ($o_1 >_{TS} o_2$) if:

$$\begin{aligned} \forall t \in TS, o_1(t,p) \quad &\Rightarrow \quad o_2(t,p) \wedge \\ \exists t' \in TS, \neg o_1(t',p) \quad &\wedge \quad o_2(t',p) \end{aligned}$$

In other words, $o_1 \geq_{TS} o_2$ and for some test $t \in TS$, o_1 detects a fault where o_2 fails to detect a fault.

```
0:   var: x : int = 0
1:   var: y : int = 0
2:   if input() = true:
3:     x = 1
4:   else:
5:     y = 1
```

Figure 3.3: Oracle Comparison Example Program

Note that power is relative to a fixed test set TS. Given different test sets, the relative power of oracles may vary. Consider the sample program p in Figure 3.3. Consider two oracles, o_x and o_y, with both oracles being simple input/output oracles, and with o_x having oracle data x and o_y having oracle data y. Consider two test sets T_t and T_f, each with exactly one test, such that T_t sets *input*() to true and T_f sets *input*() to false. Assume both lines 3 and 5 are incorrect (e.g., wrong constant is assigned). Then:

$$\begin{aligned} o_x \quad &>_{T_t} \quad o_y \\ o_y \quad &>_{T_f} \quad o_x \end{aligned}$$

For some pairs of of oracles, it may be the case that:

$$\forall TS \subseteq 2^{\mathcal{T}}, o_1 \geq_{TS} o_2$$

In other words, o_1 has power greater than or equal to o_2 for all possible test sets TS. In such a case we state that oracle o_1 has power *universally* greater than or equal to oracle o_2 (written $o_1 \geq o_2$). For example, consider an oracle o_a defined in terms of a set of assertions A, where $\neg o_a(t, p)$ indicates that test t violates an assertion $a \in A$. Let A' be an additional set of assertions, and let oracle o_{a2} be an oracle defined in terms of a set of assertions $A \cup A'$. As the set of assertions used by o_{a2} is a superset of the set of assertions used by o_a, for any test set TS, $o_{a2} \geq_{TS} o_a$ and thus $o_{a2} \geq o_a$. A similar situation occurs when an oracle o_1 is observing a superset of the oracle data observed by an oracle o_2, for example, o_2 observes the outputs from the program and o_1 observes additional internal state information. Given that both o_1 and o_2 are complete, $o_1 \geq o_2$.

3.4.2 Probabilistic Comparison

The power relation is a fairly restrictive relation between oracles: if $o_1 \geq_{TS} o_2$, then not only does o_1 detect more faults, it must detect every fault detected by o_2. While this relationship will often hold for oracles constructed using the same basic principle (e.g., sets of assertions), we desire a method of comparing the effectiveness of *all* oracles, i.e., a total comparison relation. This includes oracles constructed using different principles, e.g., assertion-based oracles versus expected value oracles.

Weyuker et al. recognized this problem with respect to test coverage criteria and defined a more useful probabilistic comparison between test criteria called *PROBBETTER* [74]. (We will hereafter refer to *PROBBETTER* as *PB*.) A criterion C_1 is *PB* than C_2 with respect to program p and specification s if a randomly selected test set satisfying C_1 is probabilistically more likely to detect a failure in p than a

randomly selected test set satisfying C_2 (written as C_1 PB C_2). A 'randomly selected test set' refers to a test set drawn from the set of all possible test sets satisfying a criterion C. As most criteria are monotonic, the number of test sets satisfying C is often very large or infinite [70]. Consequently, it can be difficult to *prove* that C_1 is PB than C_2; nevertheless, empirical studies of test coverage criteria effectiveness can be used to approximate this relationship (indeed, this is arguably one of the primary contributions of such studies), thus, rendering this criterion comparison and other similar probabilistic comparisons useful.

We base our total oracle comparison on the Weyuker et al. PB relation. We state an oracle o_1 is PB than oracle o_2 with respect to a test set TS

$$o_1 \ PB_{TS} \ o_2$$

for program p if for a randomly selected test $t \in TS$, o_1 is more likely to detect a fault than o_2. We state o_1 is universally PB than o_2 if $o_1 \ PB_{\mathcal{T}} \ o_2$, where $\mathcal{T}$ is the entire set of tests that can be run against p. (This is conceptually different from the definition of universally greater power outlined above.)

We can show power is a strictly stronger relation than PB when applied to oracles, i.e., for test set TS and program p,

$$o_1 >_{TS} o_2 \Rightarrow o_1 \ PB_{TS} \ o_2.$$

Assume we have oracle o_1 and o_2 such that for test set TS and program p, $o_1 >_{TS}$ o_2. For any test $t \in TS$, one of the following is true:

$$o_1(t,p) \quad \wedge \quad o_2(t,p) \tag{3.1}$$

$$\neg o_1(t,p) \quad \wedge \quad \neg o_2(t,p) \tag{3.2}$$

$$\neg o_1(t,p) \quad \wedge \quad o_2(t,p) \tag{3.3}$$

In other words, for all $t \in TS$ it must be true that (1) neither oracle detects a fault, (2) both oracles detect a fault or (3) o_1 detects a fault while o_2 does not. Based on the definition of oracle power, it cannot be the case that o_2 detects a fault if o_1 does not detect a fault. Clearly, given an randomly selected $t \in TS$, o_1 is at least as likely to detect a fault as o_2. Furthermore, we know there exists at least one $t \in TS$ such that $\neg o_1(t,p) \wedge o_2(t,p)$ (note the use $>_{TS}$) and thus for at least one $t \in TS$, o_1 detects a fault that o_2 does not. Therefore o_1 PB_{TS} o_2.

3.4.3 Oracle Metrics

Arguably, one of the core contributions of testing research is evaluating how testing approaches relate to one another. Unsurprisingly, then, a number of metrics have been proposed for discussing the set of programs $\mathcal{P}$ and the set of tests $\mathcal{T}$, including software testability [64], various test coverage criteria, and the test coverage criteria comparison relations of power, *PROBBETTER*, subsumes, etc. [74].

However, we are unaware of any metrics specific to test oracles. In this section, we have proposed two basic oracle comparison metrics, and have shown that the more restrictive (but non-total) comparison, *power*, implies the less powerful, but total comparison, *PB*. These metrics allow us to compare oracles in terms of fault finding ability, and highlight a potential (albeit in retrospect rather obvious) avenue for research into oracles—analytically and empirically comparing different oracles, as is commonly done for test coverage criteria. Future work in oracles may yield more metrics not explicitly based on fault finding ability, such as test oracle cost. In the remainder of the chapter, we will explore how our extended framework explicitly considering test oracles influences the testing process.

3.5 Implication of Functional Model of Testing to Previous Work

In this section, we revisit some earlier influential work in testing, exploring how explicitly considering a test oracle affects the results. We also explore how existing research can be used to discuss problems related to test oracles.

3.5.1 Comparing Coverage Criteria

A significant portion of the theoretical and empirical testing research is concerned with methods of comparing coverage criteria. While several methods have been proposed, they implicitly assume the presence of an oracle. This can lead to conclusions relying on key assumptions that are either unstated or minimally discussed. If we instead consider that the oracle may vary, we can arrive at conclusions that are different from published results in subtle, but important ways.

Power Comparison

We first illustrate why oracles are relevant using the *power* relation, first proposed by Gourlay for test methods and subsequently adjusted by Weiss [68] for use with test coverage criteria. Weiss states a criterion C_1 is at least as powerful as C_2, written as $C_1 \geq C_2$, if for any program p and specification s, if all test sets satisfying C_2 exposes an error in p then so do all test sets satisfying C_1. Note here that the definition requires *all* test sets satisfying the criterion reveal the fault—an unlikely occurrence in practice. Weiss's discussion completely omits the notion of an oracle; we assume a constant complete oracle o is used.

We restate the definition of the power of a test coverage criterion using our framework. A criterion C_1 is at least as powerful as a criterion C_2 with respect to a complete

oracle o (written $C_1 \geq_o C_2$) if:

$$\forall p \in \mathcal{P}, s \in \mathcal{S}, C_1(p, s, T_1), C_2(p, s, T_2) :$$
$$(\exists t_2 \in T_2, \neg o(p, t_2) \Rightarrow \exists t_1 \in T_1 \neg o(p, t_1))$$

In other words, if all test sets satisfying C_2 are guaranteed to find a fault for p when using oracle o, then so are all test sets satisfying C_1. This formulation makes the role of the oracle explicit—the relative power of a test coverage criterion is defined with respect to a constant oracle.

It is easy to show that the oracle is relevant in the *power* relation. Consider the statement (ST) and branch (BR) coverage criteria. As subsumption between criteria implies power [74], we know that $BR \geq ST$. Generally speaking, this is a vacuous relationship, as neither coverage is guaranteed to find faults for most programs p. Nevertheless, assume there exists a program p which has some fault f revealed by every test set satisfying statement coverage (and thus every test set satisfying branch coverage), but does not always propagate to the output (e.g., an incorrect constant is used).

Let o_{out} be an input/expected value oracle considering only the outputs, and let o_{all} be an input/expected value oracle considering both the outputs and the internal variables. If a test set T_{BR} satisfying branch coverage is paired with o_{out}, and a test set T_{ST} satisfying statement coverage is paired with o_{all}, then T_{ST} , o_{all} is guaranteed to detect f, but T_{BR} , o_{out} is not. Thus, the *power* relation for test coverage criteria requires the same oracle to be used with both coverage criteria.

Probabilistic Comparison

The *power* relation between test coverage criteria is known to be vacuous for most criteria and is thus of limited value [74]. We extend the Weyuker et al. PB relation [74]

to consider oracles explicitly using the same notation and terminology used for *power* above.

```
0:   var: temp : int = 0
1:   var: out : int = 0
2:   forever:
3:     out = temp
4:     temp = input() * 2
```

Figure 3.4: PROBBETTER Example Program

To demonstrate the effect of oracles on *PB*, consider the following example. Let p be the program in Figure 3.4, and let the specification s state that *out* at iteration i should be equal to the input at iteration $i-1$, or 0 if $i = 0$. Assume the number of possible inputs is bounded at 100, with a range of -49 to 50. Let $TL1_{all}$ and $TL2_{sin}$ be coverage criteria such that for a test set TS, $TL1_{all}$ is satisfied if TS contains every test of length 1 and no other tests, and $TL2_{sin}$ is satisfied if TS contains exactly one test of length 2. (We define length as the number of loop iterations.) Let o_{out} be an oracle with oracle data *out*, and let o_{all} be an oracle with oracle data *out* and *temp*. Both oracles are input/expected value oracles, signaling a fault when the value of a variable is incorrect. *out* is considered incorrect when s is violated and *temp* is considered incorrect when the value is not equal to the prior input.

p contains a fault, as line 4 should not double the input. Consequently, the value of *out* at iteration i is only equal to the input at iteration $i-1$ if the input at iteration $i-1$ was 0. We make two observations about detecting this fault. First, to detect the fault we must use an input other than 0, as p satisfies s for 0. Second, the fault requires at least least two inputs to reach the output, and thus *out* will be correct for all tests of length one. Consequently, no test set satisfying $TL1_{all}$ will detect the fault using oracle o_{out}, while all test sets satisfying $TL1_{all}$ will detect the fault using o_{all}. Furthermore, most test sets satisfying $TL2_{sin}$ will detect the fault using either

oracle; when using o_{out}, only tests in which the first input is 0 will fail to detect the fault, while when using o_{all} only the test in which both inputs are 0 will fail to detect the fault.

Let $Prob(C, O)$ be the probability of detecting a fault when using oracle O and a randomly selected test set satisfying criteria C. We can state:

$$Prob(TL1_{all}, o_{out}) = 0.0$$

$$Prob(TL1_{all}, o_{all}) = 1.0$$

$$Prob(TL2_{sin}, o_{out}) = 0.99$$

$$Prob(TL2_{sin}, o_{all}) = 0.9999$$

Thus, for program p and specification s:

$$TL2_{sin} \; PB_{o_{out}} \; TL1_{all}$$

$$TL1_{all} \; PB_{o_{all}} \; TL2_{sin}$$

Implications

These results highlight the relationship between oracles, tests, and programs on the efficacy of the testing process. The relationship between oracles and tests can easily be seen from these results. Both the *power* and PB relation were defined to compare the efficacy of the test coverage criteria (with respect to a program and specification in the case of PB). However, it is clear that oracles cannot be ignored when discussing test selection; to do so may yield misleading or incorrect conclusions.

While less obvious, these results also highlight the relationship between oracles and programs. The construction of the program in Figure 3.4 is such that the error on line 4 produces incorrect internal state for 99% of the inputs, but cannot affect the

output unless a test length of at least 2 is used. If instead the error had occurred on line 3 (i.e., *out* was doubled and *temp* was not), the program would be semantically equivalent in terms of input/output behavior, but *PB* would be unaffected by the oracle used. That such a subtle change can completely negate the benefit of using a more powerful oracle indicates that as with test selection and oracles, we cannot ignore program characteristics when discussing oracle selection. We further explore the relationship between programs and test oracles later in this section.

3.5.2 Mutation Testing

Mutation testing is a test selection method based on selecting a set of tests to detect small (usually syntactic) changes in the program [19]. Briefly, to select a set of tests satisfying mutation coverage for a program p, we first produce a set of mutants M that differ from p in small ways (e.g., change arithmetic operators, swap variable names, etc.). We then select a set of tests T such that each semantically different mutant $m \in M$ is distinguished from p.

Several types of mutation testing have been proposed. In strong mutation testing [19], we must find a set of tests T such that $\forall m \in M, \exists t \in T,\ p(t) \neq m(t)$, i.e., the output of each faulty program m differs from p's output for some test t. In weak mutation testing [38], we need only find a set of tests T such that for each $m \in M$, the internal state of the p and m differs for some test t. In [78], Woodward and Haywood note that mutation testing exists on a spectrum, with strong and weak mutation on opposite ends of the spectrum.

This spectrum of approaches is primarily differentiated by the method used to determine if a mutant has been detected, i.e., the test oracle. Recall that in Section 3.2.2 we defined a *complete adequacy criterion* to be an adequacy criterion defined in terms of both the test set and the oracle used. If we view the method used to distinguish

the mutants M from the program p as an oracle, we can reformulate the spectrum of mutation testing approaches as a single, complete adequacy criterion.

For the set of mutants M, mutation adequacy Mut_M is satisfied for program p, specification s, test set TS, and oracle o if:

$$Mut_M(p \times s \times TS \times o) \Rightarrow \forall m \in M, \exists t \in TS : \neg o(t, m)$$

In other words, for each mutant $m \in M$, there exists a test t such that the oracle o signals a fault.

This formulation of mutation testing differs slightly from the usual approaches to mutation testing, as the oracle is part of the actual testing process, whereas generally the method used to distinguish M from p is only used to select tests. Nevertheless, this formulation captures the core of mutation testing—constructing a testing process that is guaranteed to detect a set of pre-specified faults—without focusing on *how* the faults are detected. A very strong oracle can be used with a small number of simple tests; conversely, a weak output-only oracle can be used with a tests that ensure the mutant faults propagate to the output.

The relationship between the program being tested, the tests selected, and the oracle used is clear in this formulation of mutation testing. From the program p, a set of mutants M are generated. Using the set of mutants, an oracle o and a set of tests TS are selected such that each mutant is detected. If we change one testing factor, the other factors must also change accordingly to satisfy the criterion—a different program yields different mutants, thus requiring different tests and/or a different oracle; a weaker oracle may require more or different tests; simpler tests may require a more powerful oracle. We believe the close relationship between factors in mutation testing to be worth considering—mutation testing is based on detecting faults, and detecting faults is the goal of any testing process. Insights related to mutation testing seem likely to hold in many testing processes.

3.5.3 Testability

The testability of a software system, as defined by Voas et al., is the probability that the system will fail *if faulty* [65]. Generally, methods of computing software testability estimate the probability of a fault in a specific program location (e.g., a statement) propagating to the output, usually with respect to an input distribution or specific input. By computing the testability of each program location, it is argued, we can focus testing resources on program locations that have low probabilities of propagating errors. As a representative technique, we explore only work led by Voas related to the PIE (Propagation, Infection, and Execution) method [64, 65, 67].

Voas et al. define several testability metrics [64]. Consider the *propagation estimate* metric, denoted $\psi_{l,a}$. The propagation estimate is the estimated probability that a perturbed value of a at location l will affect the output. In practice, measuring testability is about estimating failure probabilities, and $\psi_{l,a}$ is therefore used to estimate the probability that a, if incorrect, will cause the program to fail. Consequently, $\psi_{l,a}$ only makes sense if we assume the presence of an oracle defined in terms of the outputs. If we instead consider that the oracle may not be defined in terms of the outputs, the propagation estimate above becomes less informative—we are not interested in the probability of a fault propagating to *any* output, we are interested in the probability of a fault being detectable by the oracle used.

To account for the oracle, we redefine propagation estimate with respect to an oracle o, denoting it $\psi_{l,a,o}$. This redefined propagation estimate is the estimated probability that a perturbed value of a at location l will affect a variable in the oracle data of o. This metric thus estimates the probability that a fault at a will be detectable by oracle o. We can show that given an arbitrary $o \in O$, $\psi_{l,a,o}$ is not necessarily equal to $\psi_{l,a}$. Suppose we have a program p, a set of tests t, and three oracles, o_o and o_v and o_a. Let o_o be an oracle with oracle data containing every

output and no other variables (i.e., the oracle assumed by $\psi_{l,a}$), let o_v be an oracle considering the single internal variable v, and let o_a be an oracle considering the single variable a. Let $1.0 > \psi_{l,a,o_o} > 0.0$ and let a be some variable defined after the last assignment of v (thus a cannot propagate to v). Therefore:

$$\begin{aligned} 1.0 &> \psi_{l,a,o_o} > 0.0 \\ \psi_{l,a,o_v} &= 0.0 \\ \psi_{l,a,o_a} &= 1.0 \\ \psi_{l,a,o_a} &> \psi_{l,a,o_o} > \psi_{l,a,o_v} \\ \psi_{l,a,o_a} &> \psi_{l,a} > \psi_{l,a,o_v} \end{aligned}$$

We can see that in order to accurately use testability information to guide software testing, we must account for the oracle used. If we do not, we may select tests likely to propagate errors to variables in which we are not interested, or we may direct resources to increase testing of parts of the program unlikely to propagate errors to the output, ignoring the fact that these parts of the program may already be covered by the oracle data.

Effect on Oracle Selection

Software testability is often proposed as a method of directing test selection; by determining which parts of the program are and are not likely to hide faults, we can select tests proportionally. However, software testability can *also* be used to guide oracle selection.

Consider the previous example, assume the variable a is unlikely to propagate to the output. If we wish to improve fault finding, we can select tests aimed at improving the probability of a propagating to the output, *or* we can use a stronger oracle with oracle data containing a variable to which an error in a is likely to propagate.

This leads to the observation that *variables with low propagation estimates represent opportunities for increasing the oracle power*. It then naturally follows that *if all variables in the program have high propagation estimates, increasing the oracle data is unlikely to significantly improve the oracle power*. Note here that the former observation has been alluded to by Voas and Miller, who proposed using testability to guide the creation of assertions [67].

Implications

Like mutation testing, testability metrics highlight the close interrelationship between programs, test sets, and oracles. Certain faults may be difficult to uncover in a program p through testing. By computing the testability of p, we can determine where these faults are likely to hide, and then direct testing resources—both in terms of tests and oracles—to finding them. Voas suggests adding tests to better exercise parts of the code likely to hide faults [64], thus using testability information to improve the testing process. As noted above (and by Voas [67]) we can also use testability information to select better oracles. Clearly, doing both may be unnecessary; if we use testability information to select a better oracle and thus increase the testability, we may no longer need additional tests. Similarly, given a large number of tests compensating for a low propagation estimate, selecting a better oracle may provide little improvement.

3.6 Chapter Conclusion

In this chapter, we have aimed to provide a foundation for software testing research—in particular empirical testing research—that is better suited for the task than previous attempts. In particular, we have attempted to provide a foundation more capable of exploring the interrelationship between tests, programs, and oracles. To accomplish

this, we explored both functional and computational formalisms for describing testing, with a focus on being able to describe both test inputs and test oracles. We then formalized several previously undescribed or informally described testing concepts, in particular concepts related to test oracles. Finally, we explored the implications of our work on previous work in the theory of testing, demonstrating the influence of test oracles on the effectiveness of the testing process.

Specific contributions of this chapter include:

- We extended Gourlay's functional framework for describing testing to account for test oracles
- We illustrated how Kripke structures can be used to discuss problems of testing
- We defined the following testing concepts, using either the functional or computational formalisms:
  - Properties relating test oracles to their ability to determine correctness: *complete*, *sound*, and *perfect*
  - *Oracle data*, the set of variables considered by a test oracle
  - *Oracle adequacy criterion*, a method of selecting test oracle (similar to test adequacy criterion), defined as $O_C \subseteq P \times S \times O$
  - *Complete adequacy criterion*, a method of selecting both a test suite and a test oracle together, defined as $TO_C \subseteq P \times S \times 2^T \times O$
  - Several methods of comparing test oracle based on existing methods of comparing test inputs
- Explored the implications of explicitly considering test oracles on existing work in the theory of testing, demonstrating that, in theory, test oracles can influence the effectiveness of the testing process

The work in this chapter raises a key question—given that our results indicate that test oracles and test inputs both influence the effectiveness of the testing process *in theory*, how much can we expect these artifacts to influence the effectiveness of the testing process *in practice*? In the next few chapters, we perform an empirical study exploring this (and other) questions. We will use our work in this chapter to describe several problems of interest, demonstrating that in practice the interaction between artifacts exists and quantifying the impact of this relationship.

Chapter 4

Empirical Study Overview

In the previous chapter, we explored, theoretically, how testing artifacts can interact to influence the effectiveness of the testing process, finding that previous conclusions in testing research are sensitive to variations in testing artifacts used ; notably, varying the *test oracle* selected may significantly alter conclusions on the effectiveness of test input selection methods.

This work raises several questions. First, to what degree do the results shown to hold in theory also hold in practice? Software engineering, and software verification in particular, contains many examples of techniques whose theoretical and practical efficacy vary. For example, formal verification methods such as model checking are often undecidable or intractable in the worst case, but can (with clever heuristics) yield good results for real systems in practice. Thus it is quite possible that our theoretical results illustrating the interaction between testing artifacts may not hold in practice, or may hold but not to any practically significant degree. We therefore must both demonstrate and quantify the practical impact of these interactions.

Second, how do testing artifacts not considered in Chapter 3, or properties of testing artifacts not considered, impact the effectiveness of the testing process? Specifically, how do practically relevant considerations such as the *program structure*, *test coverage criterion*, or percentage of *system state* considered by the test oracle—each an aspect of the testing process difficult to formally reason about—interact to influence the effectiveness of the testing process?

Finally, assuming these interactions between artifacts exist and that they are practically relevant, what can be done to improve the effectiveness of the testing process? In particular, do there exist unexplored avenues of research in software testing that might be used improve software testing in the future?

These questions are broad and—as noted in the introduction—there is a large amount of work that must be done to thoroughly answer them. Such work must encompass many domains of software engineering and many aspects of the testing process. Nevertheless, to begin to answer these broad questions, we have formulated several research questions exploring specific interactions between testing artifacts within the domain of critical avionics systems, and have conducted an empirical study capable of answering these questions.

In this chapter, we will provide the details of how our study was conducted. In Section 4.1, we discuss testing artifacts relevant to our study. In Section 4.2, we describe how these artifacts are varied in our study, stating the dependent and independent variables. Finally, in Section 4.3, we outline our experimental design, detailing how each step is conducted in Sections 4.4 through 4.9.

In the following three chapters, we present the specific research questions explored, along with motivation, analysis and discussion. The questions in each chapter explore similar types of interactions, and were selected because we believe they are particularly noteworthy or relevant in our domain of choice, critical avionics systems. The specific research questions require motivation which is best left to subsequent chapters; however, to better understand our study design, we list the high level interactions each chapter explores:

Program Structure – Structural Coverage Criteria: How does the structure of the system under test influence the cost and effectiveness of structural coverage criteria?

Test Oracles – Test Inputs: How do test oracles and test inputs interact to influence the effectiveness of software testing?

Program Structure – Test Oracles: How does the structure of the system under test influence the ability to test oracles to improve the effectiveness of software testing?

Final note: the research questions were proposed *prior* to designing the experiment; we did not simply collect large amounts of data and mine it to discover these interactions. The separation of the experimental design from the research questions and analysis has been done only to simplify the organization of this work, and avoid needless repetition of implementation details.

4.1 Overview of Testing Synchronous Reactive Systems

In Chapter 3, we discussed testing using a functional notation. This notation is intentionally very high level, and does not capture all the details relevant to testing systems in our domain of interest, critical avionics systems. Therefore, in this section, we provide an overview of testing artifacts relevant to testing our case examples, describing the format and relevant properties of each of the four artifacts shown in Figure 4.1.

4.1.1 Program

Each case example in our study is expressed in a synchronous reactive language called Lustre [12]. A synchronous reactive systems operates using the following steps: (1) receives input from the environment, (2) recomputes the internal state, and (3) produces output. These steps are performed sequentially, and repeated until the system terminates.

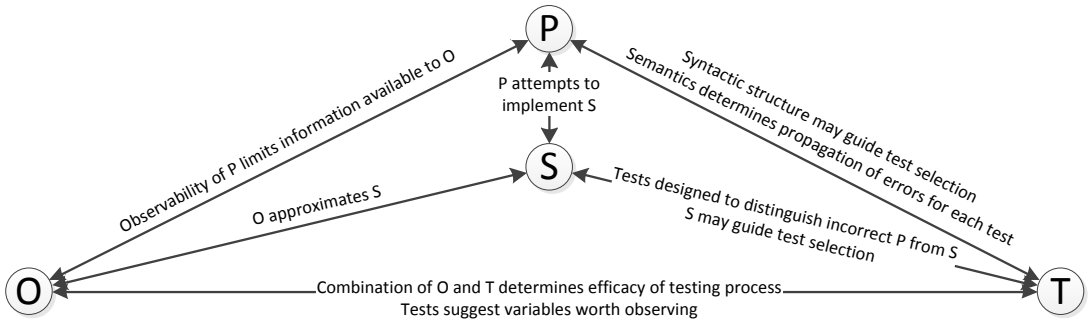


Figure 4.1: Example Relationships Between Testing Artifacts

The number of programming constructs available in Lustre is considerably smaller than those available in more traditional imperative programming languages such as C or Java. Notable constructs that are generally available in imperative languages but do not exist in Lustre include:

- Loop constructs, e.g. the for-loop. Conditional branches exist, but jumps backward cannot occur.

- Dynamic memory, i.e., a heap. All variables are predefined, with a type specified.

- Side effects. Each system is a effectively list of assignments of the form `varName = <equation>`. Each element in an equation must be computed before it used (i.e., cyclic dependencies in which the value of A depends on the value of B and vice-versa are not allowed).

- Complex data types. This follows from the restriction on dynamic memory. While records/tuples can be specified, these are merely syntactic sugar for simple data types such as integers, Booleans, and floating point numbers.

```
node exampleProgramNode(input_1: bool;
      input_2: bool;
      input_3: int;
      input_4: bool)
   returns (output_1: bool);
var
   internal_1: bool;
   internal_2: bool;
   internal_3: bool;
   internal_4: bool;
let
   internal_1 = (input_1 OR input_2) OR input_4;
   internal_2 = input_3 <= 100;
   internal_3 = input_3 > 100;
   internal_4 = IF internal_1 THEN internal_2 ELSE internal_3;
   output_1 = internal_4;
tel;
```

Figure 4.2: Example Lustre Program

We present a sample Lustre program in Figure 4.2

In this study, there are three aspects of these systems that are particularly relevant: the potential for *masking*, the *complexity of expressions*, and the *observability* of the system.

Masking occurs during execution when a value is computed, but cannot possibly influence the system's output. For example, in Figure 4.2, the value of `internal_3` is only externally visible when `internal_1` is `false`. If we were testing this system with a test input where `input_4` was always `true`, the computation of the value of `internal_3`—even if wrong—would not be detectable when examining the output variables. As we saw in the previous chapter, the presence of masking (or lack thereof) has implications in the construction of test oracles. If masking exists, it may be beneficial to our testing effectiveness to consider the internal state of the system; if masking does not exist, considering internal state is likely less useful. Futhermore,

the presence of masking is important when selecting test inputs. If we wish to both detect faults and cause them to propagate to the output, our selection of test inputs must be careful designed to avoid masking. Note that the presence of masking is also sometimes referred to as a lack of *fault propagation*—erroneous values in the program have not propagated to an observable output.

The *expression complexity* refers to the size of expressions in the source code. Intuitively, an expression is more complex if has more operators and/or references more variables. For example, in Figure 4.2, the expression assigned to `internal_1` is more complex than the expression assigned to `internal_3`, as the former contains more Boolean operators and references more variables. The practical implications of expression complexity is chiefly its impact on structural coverage criteria—when using such criteria, more complex expressions often require more tests to satisfy the criteria. Given semantically equivalent formulations of the same program, the effectiveness of such a criterion may varying depending on the complexity of expressions in each program.

Observability simply refers to our ability to observe program state. In this study, we will assume that only the values of variables (internal state and output) can be observed. Thus given two semantically equivalent programs, one which defines more variables than the other, we view the former as more observable. The practical implications of observability chiefly relate to test oracle construction—value or computation which is not observable cannot be considered in the oracle data. For example, the computation of `internal_1` involves two `OR` expressions, but only the final computation is assigned to `internal_1`; we cannot consider the value of `input_1 OR input_2` in the oracle data.

We recognize that, given a sufficiently advanced test harness, we can observe any computed value, even those not assigned to a variable. We will assume in this study,

```
         input_1  input_2  input_3  input_4
step 1:     True     True       42    False
step 2:     False    True       13     True
step 3:     False   False        9     True
```

Figure 4.3: Example Lustre Program Test Input (Length of Three)

however, that such a harness does not exist. In any case, the construction of such a test harness would likely not occur unless evidence existed that it would be beneficial, and this evidence does not appear to exist. (Though in Chapter 7 we provide it.)

4.1.2 Test Inputs

Due to the simple *input* → *compute state* → *output* nature of synchronous reactive systems, as well as the lack of dynamic memory and the restriction to simple scalar datatypes, the construction of test inputs is straightforward in our study. Each *test input* (also termed a *test case* or simply *test*) consists of a list of one or more steps. Each *step* defines a value for every input in the system; the number of steps is the *length* of the test. A set of test inputs is called a *test suite*; the number of inputs is equal to the *size* of the test suite. An example test input for the system in Figure 4.2 is given in Figure 4.3.

As discussed in the introduction, developing methods of selecting test inputs is a longstanding research objective. In this study, the coverage criterion each test suite satisfies, along with the size of the test suite, will be the primary properties distinguishing test suites. We will explore the use of two broad types of test coverage criteria: *structural coverage criteria*, and *requirements coverage criteria*. We will also explore the use of *random* testing.

Structural coverage criteria define the adequacy of a test suite in terms of syntactic program elements. An example of such a criterion would be statement coverage, in

which all statements must execute. Such coverage criteria are also known as *white box* coverage criteria, due to their use of source code information. In this study, we explore the use of the following structural coverage criteria:

Branch: Every branch must evaluate to true and false. [83]

Condition: Every atomic Boolean element (e.g., variable, relational comparison) in a branch condition must evaluate to true and false. Note that this does *not* imply that every branch must evaluate to true and false [83].

MCDC (Modified Condition Decision Coverage): The independence of every atomic Boolean element in every Boolean expression (e.g. branch conditions, assignments of the form `varName = a AND b`, etc.) must be shown.

Demonstrating independence of an atomic Boolean element is surprisingly complex; a good tutorial can be found in [34]. At a high level, to demonstrate independence of an atomic Boolean element, we must show the value of the element can cause the root Boolean expression evaluate to both true and false. Several variations of MCDC exist—for this study, we use Masking MCDC, as it is the method of computing MCDC used by the avionics community [17].

Requirements coverage criteria define the adequacy of a test suite in terms of some set of requirements. For example, a requirements coverage criteria based on use cases may require each use case be expressed as a test input. Such coverage criteria are also known as *black box* coverage criteria, due to their lack of use of source code information—they view testing as input/output only, treating the program internals as an opaque black box. In this study, each of our case examples has an associated set of Linear Temporal Logic (LTL) requirements [53]. In previous work, we have defined several coverage criteria over such properties [75]. We use these requirements coverage criteria in this study. They are:

Naive Requirements: Each LTL requirement must evaluate to true.

Antecedent: Each LTL requirement must evaluate to true. For requirements of the form $A \rightarrow B$ the antecedent A—the portion to the left of the implication operator—must also evaluate to true, thus preventing vacuous satisfaction.

UFC (Unique First Cause): UFC is adopted from the MCDC criterion. However, as LTL properties define paths, to satisfy UFC we must demonstrate for each LTL formula the independence of each atomic Boolean element along a path. In other words, each element must be shown to be the *unique first cause* of the formula becoming true.

UFC defines independence in terms of the shortest satisfying path for the formula. Thus, for a formula A and a path π, an atomic condition α in A is the unique first cause if, in the first state along π in which A is satisfied, it is satisfied because of atomic condition α. Note that in the context of testing, a path is represented by a test. A detailed description of UFC can be found in [75].

In random testing, we simply randomly select test inputs, a straightforward task given the nature of our test inputs. Random testing is, like requirements coverage criteria, considered to be a black box test method.

4.1.3 Test Oracles

As noted previously in Chapters 2 and 3, there exist many possible methods of constructing test oracles. In this study, we will focus on test oracles defined in terms of concrete expected values. In other words, we are interested in test oracles that, for each test input, specify concrete values the system is expected to produce for one or more variables (internal state and/or output). During testing, such an oracle compares the produced values against the expected values, failing the test if a difference

occurs. We term such oracles *expected value oracles*. In our experience with industrial parters, such test oracles are commonly used when testing synchronous reactive systems.

In this study, we will only examine complete expected value oracles, i.e., oracles such that when the program is executing correctly, the test will pass. (See Chapter 3.2.1.) As each expected value oracles operates by comparing a subset of internal state and output variables against concrete, expected values, any two complete expected value oracles for the same system must have, for a test input t, the same expected value for any variable v in their oracle data.

Complete expected value oracles therefore differ only in the set of variables for which expected values are defined, which we defined as the *oracle data* in Chapter 3. We loosely classify expected value oracles based on the composition of their oracle data as follows:

Unrestricted: Oracle data contains any combination of output variables internal state variables.

Output-Base: Oracle data contains all the output variables, and one or more internal state variables.

We refer to these classifications as subtypes, and refer to oracles as being of that subtype, e.g., oracles belonging to the unrestricted classification are unrestricted oracles. Also of note are two specific test oracles:

Output-Only: Oracle data contains all the output variables, and no internal state variables.

Maximum: Oracle data contains all output variables and all internal state variables.

These represent two oracles of particular interest: the oracle monitoring every output, and only the outputs, and the oracle monitoring all observable state information. The former represents the current practice in industry, while the latter represents the most effective test oracle for any system.

We will refer to the *size* of an oracle (e.g. "an oracle of size 25", "a large oracle"), where *size* refers to the number of variables used as oracle data. For discussions where the distinction between internal state variables and outputs is irrelevant we will simply refer to *variables* rather than *internal variables and outputs*. Finally, we will refer to an oracle where the oracle data is a set V of variables or a superset of V as *containing* or *using* V.

4.1.4 Specifications

In this study, we will not be making significant use of the system specification. Our study, as we outline below, uses mutation testing to estimate fault finding effectiveness, and thus no system specification is required to evaluate the correctness of test runs.

As we described above, however, each system has an accompanying set of Linear Temporal Logic (LTL) properties. Each set of LTL properties were judged to be "good" by the system developers. These properties are used to measure coverage of the requirements coverage metrics.

4.2 Independent and Dependent Variables

In this study we have three independent variables—each corresponding to either the program, test, or oracle set from our revised testing system from Chapter 3 and illustrated again in Figure 4.4—and one key dependent variable, the fault finding effectiveness of the testing process. Our independent variables are:

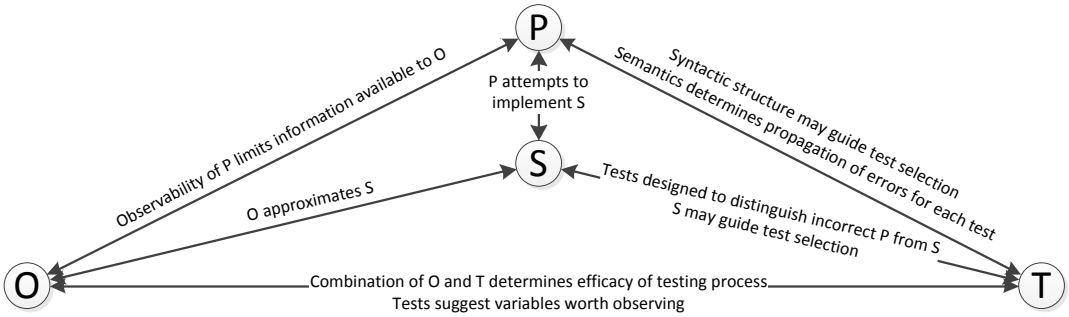


Figure 4.4: Example Relationships Between Testing Artifacts

Program Structure: For a given program, several syntactically different, but semantically equivalent programs are often possible. By restructuring programs we may influence other artifacts, such as test inputs derived based on the syntactic structure. In this study, we explore the impact of two program structures: *inlined*, in which the program has relatively complex expressions; and *noninlined*, in which the program has relatively simple expressions. This is detailed in Section 4.4.

Test Inputs: In this work, we explore six test coverage criteria—three structural coverage criteria, and three requirements coverage criteria—as well as random test suites of varying size. These criteria are described in Section 4.1.2.

Test Oracles: As mentioned in Chapter 2, methods of selecting test oracles, unlike methods of selecting test inputs, are less common in testing literature. We formulate two methods of selecting test oracles—which we term subtypes—based on the composition of the test oracles (i.e., based on the usage of output and internal state variables), varying the number of variables considered by the test oracle between the minimum and maximum oracle sizes. We also consider every oracle of size one.

Dependent variables include:

Fault Finding Effectiveness: The fault finding effectiveness (often called simply the *effectiveness*) of the testing process is the ability of our test inputs and test oracle to detect faults in our program under test. To compute the fault finding effectiveness of the testing process, we automatically generate a large number of faulty systems (termed *mutants*) from our original system, and then measure the number of mutants our testing process can detect.

Note that we use the definitions of Avizienis and Laprie [3], in which a fault is an arbitrary defect in a program that can lead to an erroneous state and possibly to a *failure*, which is an externally visible deviation from expected behavior. In our case, a failure is evident when the output trace of the mutated model deviates from the expected output trace.

This dependent variable is our primary dependent variable, and is the core of most of our research questions and analyses.

(Reduced) Test Suite Size: When generating test suites to satisfy a coverage criterion, we are chiefly interested in the effectiveness of the resulting test suite. However, in some scenarios, we are also interested in the number of tests required to satisfy the coverage criterion.

Number of Variables: In addition to the complexity of the program expressions, the inlined and noninlined programs differ in the number of (internal state) variables they contain. This is relevant when considering issues of observability.

4.3 Experimental Design

In this section, we outline the systems used in our study and described in detail the steps taken to perform the study. We begin by outlining the basic steps taken. For each case example, we performed the following:

Generated inlined and noninlined program versions. We transformed the program structure, creating one version with a high degree of expression complexity (inlined version) and one version with a low degree of expression complexity (noninlined version). This is detailed in Section 4.4. Note that unless specified, the remaining steps are applied to both the inlined and noninlined version separately (e.g., separate mutant sets are generated for the inlined and noninlined versions).

Generated mutants. We randomly generated 250 mutants, each containing a single fault, and then removed functionally equivalent mutants. (Section 4.8.)

Generated random test inputs. We generated random inputs for 1,000 tests of test lengths between 2 and 10 steps. This set of random test inputs is used for both the inlined and noninlined versions of each case example. (Section 4.5.)

Generated tests satisfying coverage criteria. For each coverage criterion considered in our experiment, we generated one set of tests satisfying said coverage criterion. The following criteria are considered: branch, condition, Modified Condition Decision Coverage (MCDC), naive requirements, antecedent, and Unique First Cause (UFC). NuSMV was used to generate each set of tests, resulting a test set with one test input for each obligation. (Section 4.6.)

Ran tests on mutants. We ran each mutant and the original case example using every test generated, and collected the internal state and output variable values

produced at every step. This yields raw data used for the remaining steps in our study. (Section 4.9.)

Generated oracles. We randomly generated three sets of oracles—one set for both the unrestricted and output-only oracle subtypes (50 oracles each) and one set containing every oracle of size 1 (only observing one variable). We randomly selected which variables were to be included in an oracle. (Section 4.7.)

Generated reduced test suites. For each coverage criterion, we generated 50 reduced test suites using a greedy reduction algorithm. Each reduced test suite maintains the coverage provided by the full test suite. (Section 4.6.)

Generated subsets of random tests. We randomly generated 200 subsets of the randomized test suite, with subsets containing 5 to 1,000 tests from the original test suite. (Section 4.5.)

Assessed fault finding ability of each oracle and test suite combination. We determined how many mutants were detected by every oracle and a reduced test suite combination. (Section 4.9.)

These steps produce the necessary fault finding data required to evaluate our questions of interest. In the remainder of this chapter, we explore the details of these steps.

4.3.1 Case Examples

We used four industrial synchronous reactive systems developed by Rockwell Collins Inc. in our experiment. All four systems were modeled using the Simulink notation from Mathworks Inc. [45] and were translated to the Lustre synchronous programming language [30] to take advantage of existing automation. Two systems, *DWM_1*

and *DWM_2*, represent portions of a Display Window Manager (DWM) for a commercial display system. Two systems represent various aspects of a Flight Guidance System (FGS), which is a component of the overall Flight Control System (FCS) in a commercial aircraft. The two FGS systems in this paper focus on the mode logic of the FGS. The *Vertmax_Batch* and *Latctl_Batch* systems describe the vertical and lateral mode logic for a flight guidance system. All four systems represent sizable, operational systems intended for real-world use. Each system has a reasonably complete (as judged by the developer) set of requirements expressed as Linear Temporal Logic (LTL) properties associated with it. Information related to these systems is provided in Table 4.1.

	# Simulink Subsystems	**# Blocks**	**# LTL Properties**
DWM_1	3,109	11,439	170
DWM_2	128	429	41
Vertmax_Batch	396	1,453	294
Latctl_Batch	120	718	103

Table 4.1: Case Example Information

4.4 Program Structure

For each case example, we generate two versions that are semantically equivalent, but syntactically different. We terms these the *inlined* and *noninlined* versions of the program, described below. Examples of an inlined and noninlined version of the program are given in Figures 4.5 and 4.6.

4.4.1 Noninlined Program Structure

In a noninlined program, the structure of the program is similar to the structure of the original Simulink model. Each signal from the Simulink model has been preserved,

```
node exampleProgramNode(input_1: bool;
      input_2: bool;
      input_3: int;
      input_4: bool)
   returns (output_1: bool);
var
   internal_1: bool;
   internal_2: bool;
   internal_3: bool;
   internal_4: bool;
   internal_5: bool;
   internal_6: bool;
let
   internal_1 = input_1 AND input_2;
   internal_2 = input_3 > 100;
   internal_3 = internal_1 OR input_4
   internal_4 = IF (internal_3) THEN internal_2 ELSE input_1;
   internal_5 = input_3 > 50;
   internal_6 = IF (internal_5) THEN internal_4 ELSE input_2;
   output_1 = internal_6;
tel;
```

Figure 4.5: Example Noninlined Program

resulting in a very large number of internal state variables, with each internal state variable corresponding to a relatively simple expression. A small example noninlined program is given in Figure 4.5.

Note that unlike our previous study exploring program structure [55], in this study the noninlined program hierarchy is flattened such that the implementation has only one node (this facilitates our study of test oracles).

4.4.2 Inlined Program Structure

In an inlined program, the program is first flattened such that the implementation (as with the noninlined program) has only one node. Once this has been completed, we then inline most, but not all, of the intermediate variables into the model. When

```
node exampleProgramNode(input_1: bool;
      input_2: bool;
      input_3: int;
      input_4: bool)
   returns (output_1: bool);
let
   output_1 =
       IF (input_3 > 50)
       THEN
           IF ((input_1 AND input_2) OR input_4)
           THEN input_3 > 100
           ELSE input_1 ELSE input_2;
tel;
```

Figure 4.6: Example Inlined Program

inlining a variable, we substitute the expression corresponding to the variable wherever the variable is referenced, and then remove the variable from the program (as it is no longer referenced). This has the effect of (1) reducing the number of internal state variables, as inlined variables are removed from the model, (2) increasing the complexity of expressions, as inlined variables are substituted wherever they are references, and (3) increasing the number of nested `if-then-else` expressions, as such expressions are often substituted into `then` and `else` branches.

Some of the structural coverage criteria used in our experiment are defined exclusively over traditional imperative structures (such as those found in C, Java, etc.); we therefore restrict our inlining to prevent syntactic constructs impossible in imperative programs from arising. In particular, we do not place `if-then-else` expressions inside `if` conditions—while this is valid in Lustre programs, it is impossible in imperative programs and would prevent us from accurately measuring coverage over structural coverage criteria.

In Figure 4.6, we present an inlined version of the program from Figure 4.5. As we can see, the inlined version has been reduced from five internal state variables

to zero, with the set of expressions condensed to one, considerably more complex, expression. This expression illustrates how inlining can increase complexity both in terms of Boolean/relational expressions and nesting of `if-then-else` statements. For example, we see that the condition formerly represented by `internal_3` has been inlined, and the `if-then-else` expression formerly represented by `internal_4` has been nested inside another `if-then-else` statement. As we will see shortly, while these transformations result in semantically equivalent programs, their impact on testing is significant.

4.5 Random Test Input Generation

We generated a single set of 1,000 random tests for each case example. Each individual test in these sets contains between 2 and 10 steps with the number of tests of each test length distributed evenly. We then generated test suites of various sizes for our experiment by randomly selecting a subset of the full random test suite. For each case example, we generated 200 test suites, with the size of each test suite evenly distributed from size 5 to size 1,000. These reduced test suites are then used in our evaluation.

4.6 Coverage Directed Test Input Generation

There exist several methods of generating tests to satisfy coverage criteria. We adopt counterexample-based test generation approaches based on existing model checking approaches to generate tests satisfying our coverage criteria of interest [23, 56]. We used the NuSMV model checker in our experiments [50].

We performed the following steps for each coverage criterion:

1. We processed the Lustre source code to generate *coverage obligations* for our

coverage obligations of interest. Each coverage obligation represents a property that should be satisfied by some test (e.g., some branch covered, some condition evaluates to true). These obligations were inserted into the Lustre source code as *trap properties* [23].

2. We translated the Lustre source code, with trap properties, to SMV syntax.

3. We ran the SMV model through NuSMV. This produces a list of counterexamples, each of which corresponds to the satisfaction of some coverage obligation.

4. Each counterexample is translated into an executable test input.

Tests satisfying requirements coverage criteria are generated using the original case example, as the LTL properties are formulated in terms of the original system. These test inputs are therefore the same across the inlined and noninlined systems. Tests satisfying structural coverage criteria are generated separately using the inlined and noninlined case examples.

Note that some coverage obligations are unsatisfiable, i.e., there does not exist a test which satisfies the coverage obligation. (This can occur, for example, if infeasible combinations of conditions are required by some coverage obligation.) Nevertheless, by using this approach, we ensure that the maximum number of satisfiable obligations are satisfied.

This approach generates a separate test for each coverage obligation. While a simple method of generating tests, in practice this results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. For example, in branch coverage, a test satisfying an obligation for a nested branch will also satisfy obligations for outer branches. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to satisfy the coverage criterion. Given the correlation between test suite size and fault finding

effectiveness [49], this has the potential to yield misleading results—unnecessarily large test suites may lead us to conclude that a coverage criterion yields an effective test suites, when in reality it is the size of the test suite that is responsible for the effectiveness.

To avoid this, we reduce each naive test suite generated while maintaining the coverage achieved. This reduction is done using a simple greedy algorithm following these steps:

1. Each test is executed, and the coverage obligations satisfied by each test are recorded. This provides the coverage information used in subsequent steps.

2. We initialize two empty sets: *obs* contains satisfied obligations and *reduced* contains our reduced test suite. We initalize a set *tests* containing all tests from our naive test suite.

3. We randomly select and remove a test t from *tests*. If the test satisfies an obligation not currently in *obs*, we add it to *reduced*, and add the obligations satisfied by t to *obs*. Otherwise we discard the test.

4. We repeat the previous step until *tests* is empty.

Due to the randomization, many reduced test suites can be generated using this process. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 randomly reduced test suites using this process.

4.6.1 A Note on Counterexample-Based Test Generation

The use of counterexample-based test generation is one of several options for generating tests to satisfy coverage criteria. As the method of test generation selected

can (and—as we will see in Chapter 5—does) impact our results, it is worth discussing why we have elected to use counterexample-based test generation and what the implications of this decision are.

Consider the problem of evaluating the effectiveness of a test suite generated to satisfy a coverage criterion. We wish to select, from (in this case) an infinite set of test inputs T, a test suite satisfying our coverage criterion. For each coverage obligation, there will generally be a very large number (possibly an infinite number) of tests satisfying the coverage obligation. Accordingly, we must select a very small fraction of tests from T to satisfy our coverage obligations.

One possible method of selecting this test suite is to randomly generate tests until each coverage obligation has been satisfied, and then, as with counterexample-based test generation, generate one or more reduced test suites. Indeed, the use of undirected random test generation is also commonly used in empirical software engineering studies, and was the primary altenative considered when selecting our method of test generation. This approach has the benefit of being somewhat unbiased—each test input is randomly drawn from T, and thus effectiveness cannot be linked to the test input generation tool, the skill of tester (when manually generating tests), etc. However, there are two potential problems with this approach: the unlikelihood of satisfying complex coverage criteria by chance, and the inherent effectiveness of random test input selection.

First, while for an arbitrary coverage obligation CO there often exists a large, potentially unbounded number of test inputs satisfying the obligation, the percentage of T satisfying CO can be very small. Thus, when randomly generating tests to satsify CO, the number of test inputs we must generate to satisfy CO with reasonable odds is potentially infeasibly high. In our own work, we have noticed that this is often true for the complex coverage obligations required by the MCDC and UFC coverage criteria,

with some satisfiable obligations remaining unsatisfied after hundreds of hours of random test input generation. Given the large number of test suites we generate in this study, we feel the resources required by this approach make it impractical.

Second, random test selection has been shown to be (somewhat surprisingly) a very effective method of test input generation [49, 25, 72, 32]. Given that we are measuring the effectiveness of test input selection methods, one fear when using random test input selection to satisfy a coverage criterion is that the effectiveness of the resulting test inputs will be largely unrelated to the coverage criterion—that is, the test inputs will be effective because random test input selection is used.

The use of counterexample-based test generation, and the NuSMV model checker in particular, avoids both of these issues. With respect to the former issue, NuSMV is capable of generating test inputs satisfying all achievable coverage obligations in a reasonable time frame. With respect to the latter issue, NuSMV—like most model checkers—generates simple, straightforward test inputs. These test inputs are generated using heuristic search, manipulating only input variables required to satisfy the obligation and leaving all other input variables set to "default" variables (0 or `false` for integer/float or Boolean values, respectively). Test suites generated to satisfy a coverage criterion using NuSMV thus tend to have short tests that do only enough to satisfy the criterion. Concerns that a test suite is particularly good because it was generated using NuSMV are, in our experience, rarely warranted.

4.7 Test Oracles

We create each oracle by selecting a subset of the internal state variables and/or outputs for use as the oracle data for the oracle. We use an oracle procedure that verifies the expected values exactly match the actual results.

For each case example, we generated 50 *unrestricted* oracles, 50 *output-base* ora-

cles, the *output-only* oracle, and the *maximum* oracle. For the output-base oracles, we evenly distributed the number of internal state variables included from 1 to the maximum number of internal state variables. For examples where generating 50 oracles of different sizes is impossible, some sizes were repeated (again evenly distributed). Furthermore, we also generated every possible oracle of size 1 to explore the variability in fault finding across individual variables (output and internal).

4.8 Mutant Generation

We created 250 *mutants* (or faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was introduced by either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. Note these mutation operators are similar to the mutation operators used in [2], in which the authors conclude mutation testing is an adequate proxy for real faults.

We seed the following class of faults:

Arithmetic: Changes an arithmetic operator (+, -, /, *, mod, exp).

Relational: Changes a relational operator ($=, \neq, <, >, \leq, \geq$).

Boolean: Changes a boolean operator ($\vee, \wedge$, XOR).

Negation: Introduces the boolean $\neg$ operator.

Delay: Introduces the delay operator on a variable reference (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value).

Constant: Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating boolean constants.

Variable Replacement: Substitutes a variable occurring in an equation with another variable of the same type.

We generated mutants so that the *fault ratio* for each fault class was approximately uniform. The term fault ratio refers to the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class. For example, assume for some example there are R possible Relational faults and B possible Boolean faults. For a uniform fault ratio, we would seed x relational faults and y boolean faults in the implementation so that $x/R = y/B$.

One risk of mutation testing is *functionally equivalent* mutants, in which faults exist but these faults cannot cause a *failure*, which is an externally visible deviation from correct behavior. For our study, we used model checking to detect and remove functionally equivalent mutants. This is made possible due to our use of synchronous reactive systems as case examples—each system is finite, and thus determining equivalence is decidable. (Equivalence checking is fairly routine in the hardware side of the synchronous reactive system community; a good introduction can be found in [63].) Thus for every mutant used in our study, there exists at least one trace that can lead to a user-visible failure, and all fault finding measurements indeed measure actual faults detected.

4.9 Data Collection

After generating the full test suites and mutant set for a given case example, we ran each tests suite against every mutant and the original case example. For each run of the test suite, we recorded the value of every internal variable and output at each step of every test using a Lustre simulator developed in a joint University of Minnesota/Rockwell Collins project. This process yielded a complete set of values

produced by running the test suite against every mutant and the correct implementation for each case example.

To determine the fault finding of a test suite t and oracle o for a case example we simply compare the values produced by the original case example against every mutant using (1) the subset of the full test suite corresponding to the test suite t and (2) the subset of variables corresponding to the oracle data for oracle o. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed") divided by the total number of non-equivalent mutants created. We perform this analysis for each oracle and test suite for every case example yielding a very large number of measurements per case example. We use the information produced by this analysis in later chapters to explore our research questions.

4.10 Threats to Validity

Note that some of the threats discussed in this section apply only to specific to chapters; nevertheless, most of these threats are common to all chapters and conclusions.

4.10.1 External Validity

We have conducted our study on four synchronous reactive critical systems. Nevertheless, we believe these systems are representative of the class of systems in which we are interested, and our results are thus generalizable to other systems in the domain.

We have used programs written in Lustre as our implementation language rather than a more common language such as C or C++. Nevertheless, systems written in the Lustre language are similar in style to traditional imperative code produced by code generators used in embedded systems development. We therefore believe that testing Lustre code is sufficiently similar to testing reactive systems written in

traditional imperative languages.

For each case example, we used a single set of LTL properties. It is possible these sets are unique in some sense, for example particularly good, particularly bad, constructed in an unusual property, etc. However, these sets of properties were judged to be "good" by Rockwell Collins engineers and are written in a style typical for these types of systems. We therefore believe our results related to requirements coverage criteria are generalizable.

When using tests generated to satisfy a coverage criterion, we have generated these tests using a model checker. Other possible options would be to use tests manually created by testers or randomly generated tests. It is possible these other options would yield results that are significantly different. However, in our experience, tests generated using a model checker are relatively less effective than other options; we therefore are effectively studying worst-case (or at least not exceptionally positive) behavior of these coverage criterion. Given that we are often evaluating the effectiveness of these coverage criterion and/or attempting to demonstrate that an interaction can occur, this worst-case behavior is desirable in our study.

For all coverage criteria, we have examined 50 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [55, 62].

For random tests, we have examined 200 test suites of sizes between 5 and 1,000 tests, using test lengths of 2 to 10 steps. It is possible that test suites of differing sizes, or tests of different lengths, may yield different results. However, tests of longer length yield little difference in our pilot studies [60]. Furthermore, in our analysis, we find that beyond roughly 250 tests, the test suites all perform roughly the same.

We have varied program structure using an in-house tool for transforming pro-

grams. It is possible that one or both structures used do not correspond to a program structure we are likely to see in actual programs. Nevertheless, the noninlined program structure is very similar to the original structure of the system before it is translated from Simulink, and we thus believe it is representative of systems likely to be considered. The inlined program structure is similar to the code generated from tools such as Real-Time workbench. It is designed to highlight the potential impact of varying program structure, and (as shown in subsequent chapters) does so very well.

When selecting test oracles, we have used expected-value test oracles. We have adopted two approaches, the unrestricted and output-base approaches, both of which rely on random selection of oracle data. It is possible that neither approach accurately captures how oracles are constructed; however, based on our knowledge of industrial practice, we believe our method of oracle construction is similar to how oracles are varied in practice.

4.10.2 Internal Validity

This study, being a study of software engineering artifacts, is highly controlled. Typical internal threats to validity such as selection biases between control and experimental groups, or a lack of a controlled environment (e.g., studies of natural phenomenon, studies conducted while under the influence of external physical forces) do not exist.

One potential threat is the extensive use of automation in our experiment. It is possible that some effects observed are due to errors in the automation of the experiment. However, much of this automation is part of an industrial framework (courtesy of Rockwell-Collins), particularly the more complex automation for program transformation, and thus been extensively verified. Other automation for running tests, producing oracles, etc., developed at the University of Minnesota has also undergone

verification, and is relatively simple. We therefore believe it is very unlikely that errors which could lead to erroneous conclusions exist in the automation.

4.10.3 Construct Validity

In our study, we measure fault finding over seeded faults, rather than real faults encountered during development of the software. It is possible using real faults would lead to different results. However, Andrews et al. have shown the use of seeded faults leads to conclusions similar to those obtained using real faults in fault finding experiments [2].

In our study, we often discuss the cost of testing, with cost being linked to test suite size and oracle size. In practice, running tests does appear to be expensive, with cost correlating with test suite size [41, 35, 59, 20] (this observation is the foundation of test suite reduction and prioritization work). In practice, oracle size also appears to be related to the cost of testing, with larger oracles requiring more runtime [79] (due to instrumentation) and requiring more effort on the part of the tester (as expected values must be defined).

4.10.4 Conclusion Validity

In our study, we perform both quantitative statistical analyses and more qualitative interpretation of patterns. In the case of statistical analyses, we have attempted to ensure the base assumptions for these analyses are met, and thus have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. (Notably, we have avoided statistical inference across case examples, as we have not randomly sampled these examples from the larger population.) Furthermore, we have avoided "fishing" for results, i.e., we have not conducted any type of data mining. We have instead designed our

experiment around a handful of questions, opting to verify that key observations are statistically significant.

In the case of qualitative interpretation, we have focused on broad, easily visible patterns, and have attempted to back up our observations with discussions of why they occur, i.e., the mechanism causing each observation. We have chosen to not highlight subtle patterns whose mechanism is unclear, as such patterns may be chance observations or not meaningful.

Chapter 5

Impact of Program Structure on the Effectiveness of Structural Coverage Criteria

In this chapter, we explore interactions between the program structure and the effectiveness of structural test coverage criteria, with a specific focus on the MCDC criterion. We begin by outlining the motivations behind our interest. We then state specific research questions, and address these questions using several analyses. We conclude with a discussion of the practical implications of this work, and how it may be used to inform the testing process in the future.

Note that the work here is a direct extension of previous work [55]. The work discussed here differs in (1) the use of coverage criteria other than MCDC, and (2) the use of fault finding effectiveness numbers.

5.1 Motivation

In Chapter 3, we presented the definition of test coverage criteria:

$$T_C \subseteq P \times S \times 2^T$$

Of interest in this chapter are *structural coverage criteria*, such as branch coverage, condition coverage, and MCDC. These criteria are often used to determine the adequacy of test inputs used in the testing process. When using such criteria, we measure how well our test inputs exercise syntactic elements in the code, in an at-

tempt to give us confidence that, should errors in the program exist, our test inputs are capable of detecting them. However, for a given program p, there often exist many syntactically different, but semantically equivalent programs p', p'', etc.. Such programs contain different syntactic elements, and thus satisfying a structural coverage criterion may require a different set of test inputs, despite that each program is functionally equivalent. In other words, for semantically equivalent programs p and p', structural coverage criteria SCC, and test suite TS, it is possible that:

$$SCC \subseteq p \times S \times TS \bigwedge SCC \not\subseteq p' \times S \times TS$$

This presents a potential problem—depending on the structure of the program, the effectiveness of structural coverage criteria may vary significantly. In this case, the confidence obtained in program correctness from using such criteria to judge the adequacy of test inputs would be low. If we understood how the structure of the program influences the effectiveness of the criteria—knowledge which is not currently available—we could determine a priori if the structure of a program was acceptable for use with the criteria. Barring that knowledge, however, we would have no way of knowing if the structure of the program was suitable for use with the criteria.

This is particularly a concern in the domain of avionics systems, as the use of the MCDC criterion is required by regulation DO-178B when certifying the most critical flight software [34]. If this criterion is sensitive to the structure of the program, the effectiveness of this certification would be questionable, as software developers are free to choose whatever structure they like when developing avionics software. Given the potentially very high costs of satisfying the MCDC coverage criterion, of particular concern is the possibility that "cheating" MCDC by structuring the program to reduce costs may be desirable, but negatively impact the effectiveness of the resulting test suites.

5.2 Research Questions

We ask the following questions about each of the three structural coverage criteria (branch, condition, and MCDC):

Question 1 (Q1): How does the program structure impact the cost, as measured by the number of test inputs required, of satisfying the coverage criteria?

Question 2 (Q2): How does the program structure impact the effectiveness, as measured by the number of faults detected, of test suites satisfying the coverage criteria?

We explore the following factors in this chapter:

Program Structure: We examine programs which are *inlined* and *noninlined*.

Test Inputs: We examine test inputs derived to satisfy the *branch*, *condition*, and *MCDC* coverage criteria. Random test inputs and test inputs derived to satisfy requirements coverage metrics are not explored, as they are not derived from program structure and thus cannot be affected by it.

Test Oracle: We use only the *output-only* oracle.

We have fixed the test oracle because of the different internal structures of the inlined and noninlined programs—the inlined program has only a few internal state variables, while the noninlined program has many such variables. Thus the output-base and maximum test oracles for the noninlined program observe a far greater percentage of the internal state than those of the inlined program. Consequently, comparisons made with oracles other than the output-only oracle are difficult to interpret in the context of $Q1$ and $Q2$.

5.3 Experimental Results

Recall that for each test coverage criterion, we began with a large test suite that achieved the maximum achievable fault finding for the criterion. We then subsequently used a randomized test suite reduction algorithm to generate 50 randomly reduced test suites. Thus, within each case example, for each combination of coverage criterion and program structure we have a set of fault finding effectiveness numbers.

Using these numbers, for each case example, program structure, and coverage criterion, we measured the following:

Average Fault Finding: Mean fault finding across reduced test suites.

Average Test Suite Size: Mean test suite size across reduced test suites.

For each case example and coverage criterion, we also computed:

Relative Change in Average Test Suite Size: We measure the relative change (as a percentage) in average test suite size when generating tests using inlined program as compared to when generating tests using the noninlined program. Positive percentages indicate average test suite size is larger for the inlined program.

Relative Change in Average Fault Finding: We measure the relative change (as a percentage) in average test suite fault finding effectiveness when generating tests using the inlined program as compared to when generating tests using the noninlined program. Positive percentages indicate average effectiveness is larger when generating test suites using the inlined program.

These measurements are given in Table 5.1.

DWM_1 System						
	NI μ Size	**I μ Size**	**% Size Cng**	**NI μ FF**	**I μ FF**	**% FF Chg**
Branch	13.7	18.48	34.89%	54.27%	48.53%	-15.36%
Condition	21.36	25.62	19.94%	75.01%	84.71%	6.89%
MCDC	32.72	56.58	72.92%	80.74%	90.2%	5.73%
DWM_2 System						
	NI μ Size	**I μ Size**	**% Size Cng**	**NI μ FF**	**I μ FF**	**% FF Chg**
Branch	19.12	420.46	2099.05%	13.58%	92.74%	556.70%
Condition	13.5	403.66	2890.07%	19.34%	87.73%	336.18%
MCDC	23.86	423.24	1673.84%	18.89%	93.33%	375.02%
Latctl_Batch System						
	NI μ Size	**I μ Size**	**% Size Cng**	**NI μ FF**	**I μ FF**	**% FF Chg**
Branch	5.62	19.78	251.95%	33.81%	55.41%	61.83%
Condition	25.1	17.34	-30.91%	48.65%	53.99%	9.59%
MCDC	56.02	75.5	34.77%	79.84%	78.33%	-3.10%
Vertmax_Batch System						
	NI μ Size	**I μ Size**	**% Size Cng**	**NI μ FF**	**I μ FF**	**% FF Chg**
Branch	12.24	35.48	189.86%	31.09%	48.0%	55.60%
Condition	193.2	62.24	-67.78%	45.58%	50.24%	11.12%
MCDC	237.38	259.8	9.44%	59.27%	90.68%	54.21%

Table 5.1: Measurements Across Structural Coverage Criteria
μ = Mean, FF = Fault Finding

5.3.1 Demonstration of Statistical Significance

To answer *Q1* and *Q2*, we began by proposing and evaluating the following hypotheses:

Hypothesis (H_1) : A test suite generated to satisfy a structural coverage criterion C over the noninlined program will require a different number of tests relative to a test suite generated to satisfy C over the inlined program.

Hypothesis (H_2) : A test suite generated to satisfy a structural coverage criterion C over the noninlined program will achieve a different level of fault finding relative to a test suite generated to satisfy C over the inlined program.

To evaluate our hypotheses, we first formed null hypotheses as follows:

$H0_1$: A test suite generated to satisfy a structural coverage criterion C over the non-inlined program will contain the same number of tests as a test suite generated to satisfy C over the inlined program.

$H0_2$: A test suite generated to satisfy a structural coverage criterion C over the noninlined program will achieve the same level of fault finding relative to a test suite generated to satisfy C over the inlined program.

To accept H_1 and H_2, we must reject $H0_1$ and $H0_2$. To evaluate these hypothesis, we use the two-tailed bootstrap permutation test, a non-parametric statistical test with no distributional assumptions [43]. 250,000 samples were made using the bootstrap permutation test. Note the use of a non-parametric test—we have little knowledge of the underlying distribution of fault finding effectiveness numbers, and therefore make as few assumptions as possible (e.g., no assumption of normality is given).

We perform this statistical test using every combination of case example and test coverage criteria. Our resulting p-values are each very small, less than 0.0001. Given a traditional $\alpha = 0.05$ (or even much smaller), we reject our null hypotheses and accept H_1 and H_2.

Note that we have avoided statistically generalizing *across* case examples and coverage criterion; rather, we have evaluated H for each pairing of coverage criterion and case example. This avoidance is due the fact that both the case examples and coverage criteria are not randomly selected from their respective populations. Statistical generalization is therefore inappropriate, as the base assumption of all statistical hypothesis testing methods—random sampling from a larger population—is

clearly violated. Nevertheless, in the remainder of this chapter we will argue several conclusions based on this data.

5.4 Discussion

As we can see in Table 5.1 and as demonstrated in the previous section, for every case example and coverage criteria, the structure of the program has a significant impact on the effectiveness of our test coverage criteria. In this section, we discuss theses results, highlighting trends observed and discussing their causes and implications.

5.4.1 Cost of Satisfying Structural Coverage Criteria

For each of the three structural coverage criteria, the structure of the program can be manipulated to alter the cost of satisfying the coverage criterion. As seen in Table 5.1, the number of tests required to satisfy each of the coverage criteria changes when varying the program structure, sometime dramatically. For nearly every case example and criterion pairing, more tests are required when using an inlined program structure versus a noninlined program structure. An extreme example of this is the *DWM_2* system: when using condition coverage, 29.7 times more tests are required on average when satisfying said criterion on the inlined system versus the noninlined system (403.6 versus 13.5 tests).

This increase in tests can be attributed to the increased complexity of the coverage obligations generated when using the inlined program as opposed to the noninlined program. Recall that when transforming the noninlined program into the inlined program, two important changes occur: (1) the number of branches in the code increases, as `if-then-else` expressions are substituted for intermediate values, and (2) the complexity of conditional expressions increases, as intermediate variables formerly used as atomic variables are instead replaced with more complex subexpressions. We

```
[1] internal42 = IF (cond1) THEN x ELSE y
[2]
[3] output1 = IF (cond2) THEN internal42 ELSE z
[4] output2 = IF (cond3) THEN z ELSE internal42
```

Noninlined Program

```
[1] output1 = IF (cond2) THEN (IF (cond1) THEN x ELSE y) ELSE z
[2] output2 = IF (cond3) THEN z ELSE (IF (cond1) THEN x ELSE y)
```

Inlined Program

Figure 5.1: Nesting of Branches During Inlining Transformation

illustrate examples of (1) and (2) in Figures 5.1 and 5.2, respectively.

These two changes account for the increase in test suite size. In the case of branch coverage, the inlining transformation increases the number of branches. This, coupled with the increased nesting of branches, contributes to the need for more tests. As seen in Figure 5.1, the branch on line 1 in the noninlined program has been substituted to become branches on line 1 and 2 in the inlined program, thus increasing the number of coverage obligations we must satisfy. Furthermore, these two branches are now nested in another branch, and therefore each coverage obligation is more rigorous—when exercising the inner branches, the outer branch must evaluate specifically to `true` and `false` (for line 1 and 2, respectively) to allow us to reach these branches. The presence of these additional constraints—depending on the relationship between `cond2` and `cond3`—may prevent the coverage obligations related to these inner branches from being covered by the same test. This increase in the number of coverage obligations, along with the likelihood that these new coverage obligations will be covered by different test inputs, results in a larger number of tests being required to satisfy branch coverage over the inlined program as compared to the noninlined program.

In the case of MCDC coverage, the increase in the complexity of each expression

has a strong influence of the number of tests required. This is illustrated in Figure 5.2, in which we can see the number of atomic conditions on line 4 grows from 2 to 4 when line 1 and 2 are inlined. As the number of tests required to satisfy MCDC is correlated with the complexity of the expressions [55], inlining the program structure results in a significantly larger number of required tests as compared to the noninlined structure. Coupled with the effect of nesting and replication outlined in our discussion of branch coverage, the number of tests required to satisfy MCDC will likely increase.

In the case of condition coverage, our results are less consistent. The *Latctl_Batch* and *Vertmax_Batch* systems exhibit a *reduction* in the number of tests required to satisfy said criterion when moving from a noninlined to an inlined program structure, while the *DWM_1* and *DWM_2* systems exhibit an increase as observed for other coverage criteria. This result was initially perplexing, given the consistent relative changes shown by the other coverage criteria.

Recall, however, that both of the *Latctl_Batch* and *Vertmax_Batch* systems are part of the mode logic for an avionics system. Upon examination, we found that many atomic conditions (e.g., `a`, `v == 0`, etc.) in these systems relate to this mode logic, and the number of these atomic conditions is relatively low. Consequently,

```
[1] internal42 = x AND y
[2] internal13 = z AND y
[3]
[4] output1 = IF (cond1) THEN (internal42 OR internal13) else z
```

Noninlined Program

```
[1] output1 =   IF (cond1) THEN ((x AND y) OR (z AND y)) ELSE z
```

Inlined Program

Figure 5.2: Increasing Complexity of Boolean Decisions During Inlining Transformation

across most of the expressions, the same small set of atomic conditions are used. When inlining such a system, the complexity of expressions will increase, but the number of basic atomic conditions within each expression may increase by very little. Thus when inlining a variable `v`, we remove coverage obligations related to `v` as it is removed, but—unlike, for example, branch coverage—we often do not increase the number of coverage obligations when we substitute the expression for `v` into an expression `e`, as the atomic conditions present in `v`'s expression and `e` are often the same. Consequently, the number of coverage obligations for condition coverage can actually decrease.

For example, consider Figure 5.2. When generating coverage obligations for condition coverage, obligations to cover `x`, `y` and `z` will be generated, along with obligations for `internal42` and `internal13`. In the inlined version, however, only obligations for `x`, `y` and `z` are generated. This phenomenon occurs frequently when inlining the *Latctl_Batch* and *Vertmax_Batch* systems, though the nesting of expressions does somewhat counter-balance it, as we will see in Section 5.4.2.

A final note: the reader may recognize that for condition coverage, each coverage obligation need not be satisfied by a different test. Depending on the structure of the system, only two tests may be needed—one in which all conditions are `true`, and one in which all conditions are `false`. Indeed, the ease with which condition coverage can be satisfied is one of the motivating factors for the more robust MCDC coverage criterion. However, recall that each test set satisfying a coverage criterion is generated using the NuSMV model checker. As noted in Section 4.6, tests generated using this method tend to be simple, leaving as many inputs to their default values (i.e., 0 or `false`), and thus tests in which multiple condition obligations are satisfied are not common.

In the context of condition coverage, this often results in tests that are only

capable of satisfying a single obligation for a given expression—while NuSMV could satisfy multiple obligations simultaneously, it processes each obligation separately, producing inputs targeted for each obligation. This behavior results in a surprisingly high number of test inputs in the reduced test suites, closer to the number of test inputs required to satisfy MCDC than the number of test inputs required to satisfy branch coverage.

5.4.2 Effectiveness of Test Suites Satisfying Structural Coverage Criteria

For each of the three structural coverage criteria, changing the structure of the program impacts the effectiveness of tests satisfying the coverage criterion. As seen in Table 5.1, this effect generally correlates with test suite size, with larger test sets—generated over inlined program structures—usually finding more faults than smaller test suites.

The improvements in fault finding appear to occur for two reasons. First, and perhaps most obvious, is the increase in test suite size. Recent work on modeling the effectiveness of testing has indicated that effectiveness is highly dependent on test suite size [49]. Thus, simply by increasing the number of test inputs used, we can improve the effectiveness of our testing process.

Second, by changing the program structure, we have increased the complexity of the constraints on our coverage obligations. Recall again the example in Figure 5.1 in the context of branch coverage. The noninlined program has only two branches, with no nesting, while the inlined program has three branches, two of which are nested inside the `then` and `else` branches. Previously, we discussed how this nesting increases the complexity of the constraints for branch coverage obligations. In this case, *two* constraints must be satisfied to satisfy the coverage obligations for the inner branches: the outer constraint (`cond2` or `NOT cond3`) plus the inner branch

constraints. These outer constraints make generating tests to satisfy both inner `if-then-else` statements using the same test less likely, contributing to the increase in test suite size.

However, these additional constraints also force specific paths to be explored during testing. For errors that are more easily detected when exploring certain paths, this can result in more effective tests. For example, assume that line 1 in the noninlined version of the program should instead read:

```
[1] internal42 = IF (cond1) THEN x ELSE (NOT x)
```

For the `else` branch in line 1, the value of `NOT x` should be assigned when `cond1` is `false`. To detect this error using an output-only oracle, we require a test in which (1) `cond1` is false and (2) `cond2` is `true` or `cond3` is `false`. Such a test will assign (incorrectly) `output1` or `output2` the value of `y`.

In the noninlined version of the program, branch coverage does not require such a test be selected—we will always have a test in which `cond1` is `false`, but for this test there is no constraint on `cond2` or `cond3`. In the inlined program, however, `cond1` is nested, and thus the branch coverage obligations for *cond1* place additional constraints on `cond2` and `cond3`. Of interest here is line 2, where `cond3` must evaluate to `false` while `cond1` evaluated to `true` and `false`. Consequently, the error will be detected given an output-only oracle. A similar phenomenon is exhibited with the MCDC criterion, as more complex expressions in the inlined programs require more specific combinations of Boolean expressions to be exercised.

As with test suite size, the results for condition coverage deviate from the trends observed for branch and MCDC coverage, and require investigation. Again of note are the results for the *Vertmax_Batch* and *Latctl_Batch* systems. Like branch and MCDC coverage, tests satisfying condition coverage over the inlined programs outperform tests satisfying condition coverage over the noninlined programs. However, recall that

fewer tests are required to satisfy condition coverage over the inlined program. This presents an unusual situation—in moving from a noninlined to an inlined program, we have both reduced the test suite size *and* increased the fault finding effectiveness of test suites satisfying condition coverage.

To understand why this occurs, recall that while the test suites are large, many of the generated test inputs are very simple (due to the use of NuSMV). The size of the test suite is therefore misleading—it consists of a large number of tests generated to satisfy simple coverage obligations, and are thus very poor in terms of effectiveness. This stands in contrast to the tests generated using the inlined program, which benefit from the more complex coverage obligations resulting from the more deeply nested expressions.

Two other observations deviate from the general trend: branch coverage for the *DWM_1* system and MCDC coverage for the *Latctl_Batch* systems. Both of these deviations appear to be chance occurrences. With respect to the former, the number of tests satisfying branch coverage over either the inlined and noninlined program differs little (approximately 5 on average), and the resulting fault finding is also low, approximately 50% in both examples. It simply appears that NuSMV has generated a test from the noninlined version that is relatively effective compared to those generated for the inlined version, improving the effectiveness of the small test suite.

With respect to the latter observation, note that the fault finding effectiveness of tests sets satisfying MCDC for the *Latctl_Batch* system is very close, roughly 79% for both the inlined and noninlined version. It appears that the effectiveness of MCDC tops out at about 79%, and inlining the program—despite the increase in test suite size—does not significantly improve the effectiveness of the test suite. This may occur if, for example, the 20% of faults both suites fail to detect are a specific type of fault MCDC is not designed to detect (e.g., arithmetic, relational). In such a

scenario, generating tests satisfying MCDC over the inlined program will only result in additional tests targeted at the wrong type of faults (faults in Boolean expressions).

5.4.3 Implications

By varying the structure of the program, we can influence—positively or negatively—the cost of satisfying structural coverage criteria. For two of these coverage criteria, it is clear how to structure our code to minimize cost: limit the number of branches in the code, in the case of branch coverage, and limit both the number of branches and the complexity of program expressions in the case of MCDC coverage. In the context of avionics systems, this observation is potentially useful, as satisfying MCDC during testing is required for certain types of systems and can be extremely expensive. By applying syntactic transformations similar to those used here to avionics systems, developers can generate systems for which MCDC coverage is significantly cheaper to achieve. For our examples, noninlined systems required test suites of size 91% to a mere 3% of the size of the test suites generated over the semantically equivalent inlined systems—a potentially dramatic saving in testing costs.

Unfortunately, for each case example and coverage criterion, test suites generated to satisfy structural coverage criteria over the inlined program generally outperformed those test suites generated to satisfy structural coverage criteria over the noninlined program, with relative improvements of up to 556.7%. Thus we see a potential tradeoff—we can restructure our program to reduce the cost of satisfying coverage criteria, but we incur the risk of reducing the effectiveness of the testing process. In the context of the avionics domain, the widespread adoption of model-based development techniques has led to the recent suggestion that MCDC should be measured over simpler artifacts. This proposal is worrisome, particularly when coupled with the strong financial incentive to restructure the program.

Both of these implications hint at a deeper issue: the sensitivity of these coverage criteria, each of which represent well known, commonly employed methods of measuring test adequacy, to the structure of the program. If we assume the role of a coverage criterion is to determine if our test inputs are adequate—as is commonly suggested, and indeed, implied in the synonymous term *test adequacy metric*—it is worth questioning if these metrics are *themselves* adequate. Given that their effectiveness can vary considerably depending on the structure, ranging from very high to—in the case of the *DWM_2* system—very low, when asking the question: *is this set of test inputs adequate?* it is difficult to recommend the use of these criteria. Accordingly, we believe that additional work must be done in this area to improve upon existing criteria. We outline possible directions in the next section.

5.5 Chapter Conclusion and Future Work

In this chapter, we have explored how program structure influences the effectiveness of three structural coverage criteria using inlined and noninlined versions of four real-world avionics systems. Our results demonstrate that within this domain, program structure has a potentially strong impact on both (1) the number of tests required to satisfy a structural coverage criterion, with larger tests suites (sometimes dramatically so) required when using inlined programs and (2) the fault finding effectiveness of test suites satisfying structural coverage criteria, with more effective test suites being generated when using inlined programs. Note that the latter result generally holds true even when the results for test suite size do not.

These results point to serious questions concerning the adequacy of these structural coverage criteria—given the variability in fault finding present when using the criteria, what level of confidence do we have that satisfying these criteria implies an effective set of test inputs? This is particularly concerning in the context of avion-

ics of systems,where the MCDC criterion is required when testing the most critical systems.

From these results, it is clear that additional work needs to be done to either improve or replace these structural coverage criteria. We therefore propose the following questions for future work:

- For the structural coverage criteria explored, can we develop a canonical method (or methods) of structuring programs such that tests generated to satisfy these criterion will be effective?
- Alternatively, can we improve the existing structural coverage criterion to be program structure-agnostic, i.e., can we reformulate these coverage criteria such that their effectiveness does not depend on program structure?
- To what extent do these results generalize to other domains, e.g., desktop applications?
- To what extent do these results generalize to other structural coverage criteria, e.g., data-flow criteria?

Chapter 6

Influence of Test Oracles and Test Input Selection on Testing Effectiveness

In this chapter, we explore how test oracles and test inputs jointly influence the effectiveness of software testing. We focus on how changing the size of the test oracle impacts testing effectiveness, and how this varies across different methods of selecting test inputs. As in the previous chapter, we begin by outlining our motivations behind this, state several research questions, and present several analyses aimed at answering these questions. We conclude with a discussion of the practical implications of this work, and how it may be used to inform the testing process in the future.

Some of the work presented here has been published [60]. The work presented in this chapter differs primarily in that (1) multiple coverage criteria are presented (only random is explored in [60]) (2) the work is considerably more thorough, presenting analyses in greater detail and also presenting analyses beyond those in [60].

6.1 Motivation

The test oracle, like the test inputs, is a necessary part of the testing process. In Chapter 3, we presented the concept of an oracle adequacy criterion:

$$O_C \subseteq P \times S \times O.$$

This relates to the problem of *oracle selection*: from the many test oracles we can

potentially create, which oracle should be used in the testing process? Previously, we theoretically demonstrated the importance of this problem, noting, for example, that the selection of the test oracle impacts the relative effectiveness of test suites. This problem has also been empirically explored in a handful of recent works [10, 79], with results indicating the selection of the test oracle can impact the efficacy of the testing process. In our own previous work within the domain of avionics systems, we have noted that the fault finding effectiveness of the output-only and maximum test oracles sometimes varies dramatically (up to 2.5x) [54]. We therefore believe the problem of *oracle selection* is, at least potentially, of similar importance as the problem of test input selection.

Unfortunately, few techniques exist for selecting test oracles. Furthermore, there exists little empirical data rigorously exploring how test oracles impact the effectiveness of the testing process. Instead, we have mostly intuition to draw upon to select test oracles, as we lack the necessary empirical foundation to understand the impact of test oracles on the testing process. As an initial step towards understanding how test oracles influence the effectiveness of testing, and, in the longer term, developing methods of selecting test oracles, this foundation must be established.

In this chapter, we work towards establishing an empirical foundation in the domain of avionics systems. We explore how testing effectiveness is influenced by varying two key aspects of the oracle data for complete expected value oracles: the type of variables in the oracle data (internal or output variables), and the size of the oracle data. We have focused on complete expected value test oracles as they are, in our experience, commonly used within the avionics domain for testing software systems, and their influence on the testing process is therefore of practical significance.

By better understanding how these two aspects impact testing effectiveness, we intend to (1) quantify the influence of oracle selection on the testing process, thus

moving away from intuition and towards empirical evidence, and (2) use this evidence to formulate ideas for future work concerning oracle selection methods.

6.2 Research Questions

We ask the following questions:

Question 1 (Q1): How does oracle size influence the effectiveness of the testing process given a fixed test suite?

Question 2 (Q2): How much do individual variables in the oracle data contribute to the effectiveness of the testing process?

Question 3 (Q3): In the case of tests generated to satisfy a coverage criterion, how does the coverage criterion used and oracle data chosen jointly influence the effectiveness of the the testing process?

Question 4 (Q4): In the case of random testing, how does the size of the test suite and oracle data chosen jointly influence the effectiveness of the the testing process?

We explore the following factors in this chapter:

Program Structure: We examine only programs which are *noninlined*. Inlined programs contain few variables, limiting our ability to vary the test oracle. The impact of this is explored in Chapter 7.

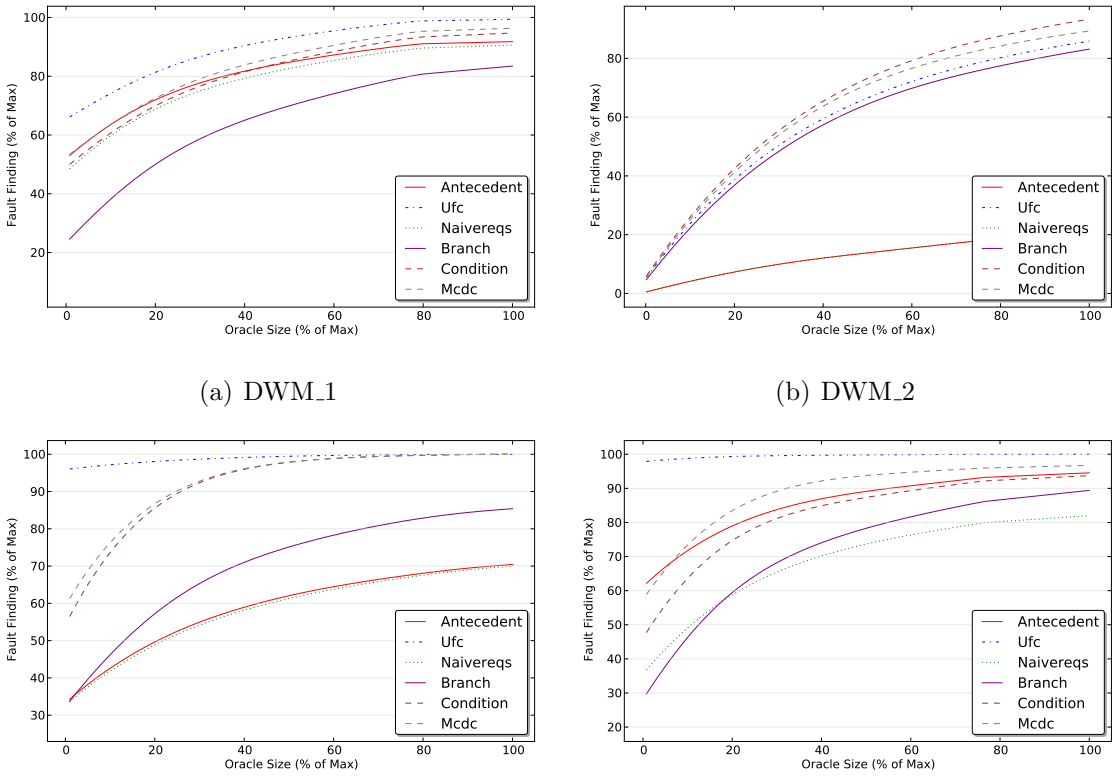
Test Inputs: We examine test suites derived to satisfy every coverage criterion and randomly generated test suites of various sizes.

Test Oracle: We explore the effectiveness of many test oracles, varying both the size and composition of the oracle data.

Recall from Chapter 4 the terms *unrestricted*, referring to oracle data containing any combination of outputs and internal state variables, and *output-base*, referring to oracle data containing all outputs and one or more internal state variables. When considering the impact of oracle size, our analysis will be divided between unrestricted and output-base oracles.

Also recall the terms *output-only* oracle, referring to oracles containing all (and only) the outputs, and *maximum* oracle, referring to oracles containing all the outputs and all the internal state variables. When considering if internal state is useful in the oracle data, our analysis will compare these two oracles.

We now perform several analyses relevant to the questions posed above. These analyses include visualizations of the data, statistical hypothesis testing, and application of regression models. Due to the differences between using randomly generated tests versus tests generated to satisfy a coverage criterion—namely, our ability to arbitrarily vary the number of randomly generated test inputs considered, something not possible for tests generated to satisfy a coverage criterion—we divide our analyses between test suites satisfying coverage criteria and randomly generated test suites. In Section 6.3 we present analyses using test inputs generated to satisfy coverage criteria, and in Section 6.4 we present analyses using randomly generated test inputs. In Section 6.5 we explore how individual variables impact the effectiveness of the testing process. When presenting our analyses, we focus on the observations made with each analysis individually, deferring a broader discussion, including the implications of our observations, to Section 6.6.



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

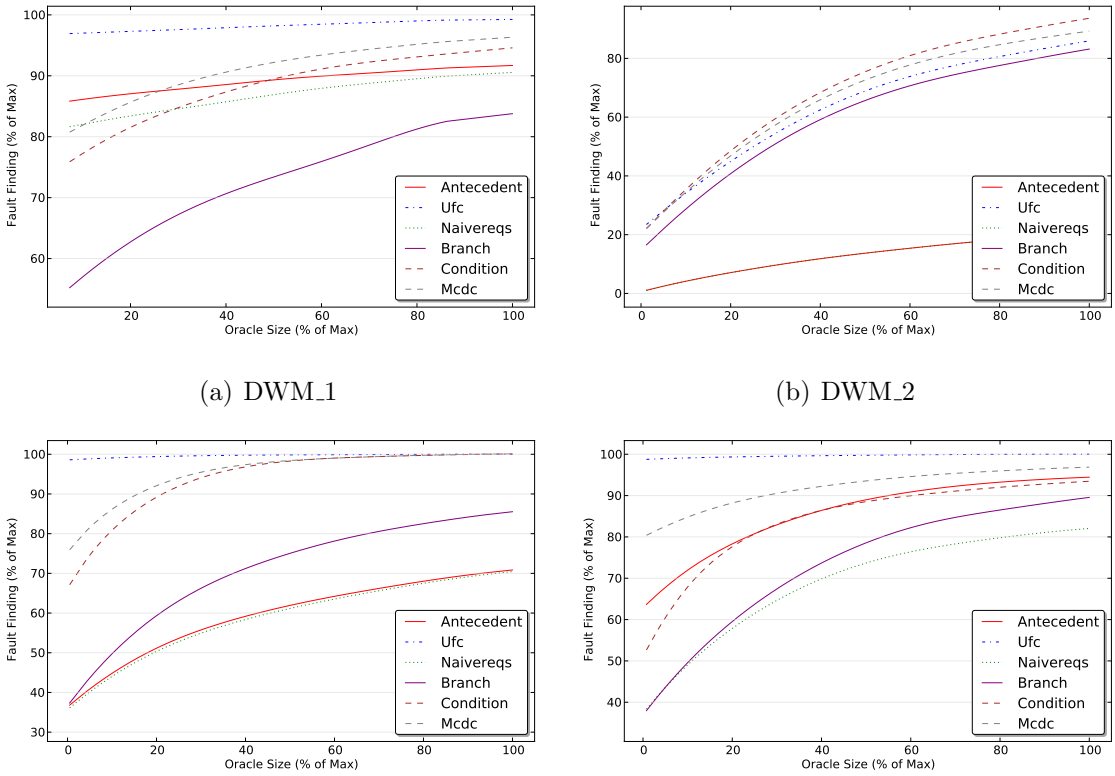
(d) Latctl_Batch

Figure 6.1: Oracle Size vs Fault Finding, Unrestricted Subtype

6.3 Influence of Oracle Data for Test Suites Satisfying Coverage Criteria

6.3.1 Influence of Oracle Size Given Fixed Test Data

We begin by visualizing the influence of oracle size via oracles generated using both the unrestricted and output-base approaches. In Figures 6.1 and 6.2, we illustrate the relationship between oracle size and fault finding using all six coverage criteria and all four case examples. Recall that for each coverage criterion, we generated 50 reduced test suites satisfying each criteria. For each criterion, we plot the average



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.2: Oracle Size vs Fault Finding, Output-base Subtype

fault finding across these reduced test suites for each test oracle. Presenting the fault finding measurements for every criterion as a scatter-plot yields a figure that is difficult to interpret; we therefore generate and draw a line for each case example from the analysis data collected using locally weighted scatter-plot smoothing—or LOESS [43]—with a bandwidth factor of 0.1. (Note we chose 0.1 as it effectively highlights the general trends present.) Note that in both figures, the right hand side of the figures always represents the maximum oracle (100% of the variables).

We draw two observations from these figures. First, the relationship between fault finding and oracle size can often be characterized as logarithmic—as oracle size

increases, fault finding initially increases rapidly, but the rate of increase tends to level off as oracle size approaches the maximum. This indicates that in most scenarios, there exists significant overlap in the faults detected by each variable within the system. For example, consider the use of the MCDC coverage criterion with the *Vertmax_Batch* system in the context of output-base oracles (Figure 6.2). When using the output-only oracle, reduced test suites satisfying MCDC detect on average roughly 60% of the possible mutants. The effectiveness of the testing process rises rapidly as internal state variables are added, until roughly 30% of the internal state variables have been added, after which effectiveness grows slowly, peaking roughly when 60% internal state variables have been added.

Note, however, that while this logarithmic relationship is generally present, there are several instances where this relationship is subdued. This leads to our second observation: test suites generated to satisfy requirements coverage metrics appear to be, in some circumstances, less influenced by the selection of test oracles than test suites generated to satisfy structural coverage criteria. In particular, test suites satisfying requirements coverage criteria appear to be less influenced by the addition of internal state variables to an output-only oracle.

As an example, consider the use of antecedent and condition coverage for the *Latctl_Batch* system in the context of unconstrained oracles (Figure 6.1). Initially, test inputs satisfying antecedent coverage outperform test inputs satisfying condition coverage, finding roughly 15% more faults. However, as test oracle size increases, this gap narrows, with the difference becoming negligible once the maximum oracle is in use. Similar examples can be found in each case example, though the degree differs.

Note that in several instances tests satisfying UFC are highly effective, even with small oracles (e.g., a single randomly selected variable). The lack of logarithmic behavior does not necessarily relate to the impact of test oracles; the test inputs,

irrespective of the test oracle, find nearly all the faults, and thus any improvements are necessarily small. This occurs in part because very large test suites required to satisfy UFC coverage for some systems (over 1,000 in some cases), and in part due to simple chance from random oracle data selection. For example, for the *Latctl_Batch* system, the unrestricted smallest oracle finds 5% more faults than the next few oracles, yielding a figure in which small oracles appear to be exceptionally effective.

6.3.2 Relative Improvement using Maximum Oracle

The previous section demonstrated that the use of larger oracles generally leads to improved fault finding, and illustrated roughly what form this relationship takes. Of particular interest to us is the difference in effectiveness between an output-only and a maximum oracle; given that the output-only oracle appears to be current industrial practice, we are interested in how the use of internal state can impact the effectiveness of the testing process. Accordingly, we would like to, for each case example and coverage criterion, (1) quantify this relationship in terms of relative improvement between an output-only and a maximum oracle, and (2) demonstrate this improvement (when it exists) is statistically significant.

In Table 6.1, for each case example and coverage criterion, we list (1) the average fault finding using the output-only oracle and maximum oracles as a percent of maximum fault finding for each case example, and (2) the relative improvement when moving from an output-only oracle to a maximum oracle. In Table 6.2, for each case example we list (1) the oracle sizes using the output-only and maximum oracles, and (2) the relative increase in oracle size when moving from output-only to a maximum oracle.

As shown, the improvement when moving from the output-only to the maximum oracle is often quite substantial, particularly when using test suites which have rel-

	Branch	Condition	MCDC	Naive Req	Antecedent	UFC
DWM_1 System						
μ **OO FF**	54.27%	75.01%	80.74%	81.97%	85.76%	96.91%
μ **MX FF**	83.17%	94.52%	96.17%	90.35%	91.55%	99.17%
% FF Imp	53.26%	26.01%	19.11%	10.22%	6.74%	2.33%
DWM_2 System						
μ **OO FF**	13.58%	19.34%	18.89%	0.0%	0.0%	20.33%
μ **MX FF**	82.65%	93.09%	88.99%	21.18%	21.18%	85.55%
% FF Imp	508.42%	381.16%	370.89%	∞	∞	320.66%
Latctl_Batch System						
μ **OO FF**	33.81%	48.65%	79.84%	34.77%	61.52%	98.73%
μ **MX FF**	89.30%	93.60%	96.70%	81.81%	94.41%	100.0%
% FF Imp	164.05%	92.37%	21.12%	135.26%	53.45%	1.28%
Vertmax_Batch System						
μ **OO FF**	31.09%	45.58%	59.27%	31.89%	32.46%	98.07%
μ **MX FF**	85.16%	100.0%	100.0%	70.19%	70.58%	100.0%
% FF Imp	173.88%	119.39%	68.70%	120.07%	117.41%	1.96%

Table 6.1: Fault Finding Effectiveness, Output-Only vs Maximum Oracle
μ = Mean, OO = Output-Only, MX = Maximum, FF = Fault Finding

atively low fault finding. In the most dramatic case, test suites that find no faults when paired with the output-only oracle (as occurs for the *DWM_2* system for antecedent and naive requirements coverage), find 20% of the detectable faults when instead paired with the maximum oracle.

We note three general trends from Table 6.1. First, when using test inputs satisfying the strongest structural or requirements test coverage criteria (i.e., MCDC or UFC), the use of the maximum oracle ensures that testing effectiveness is high (85+%) for all case examples, even for systems where said test inputs perform poorly with the output-only test oracle. For example, for the *DWM_2* system, we see that test inputs satisfying MCDC and UFC perform poorly when paired with the output-only oracle, while performance improves over threefold when pairing said test inputs with the maximum oracle. Indeed, for several systems, the fault finding effectiveness

	OO Oracle Size	MX Oracle Size	% Relative Increase
DMW_1	9	124	1277.8%
DMW2_	7	576	8128.6%
Latctl_Batch	1	129	12800%
Vertmax_Batch	2	417	20750%

Table 6.2: Oracle Size, Output-Only vs Maximum Oracle
OO = Output-Only, MX = Maximum

when pairing MCDC or UFC coverage with the maximum oracle is very high, 95+%.

Second, we note that when using test inputs satisfying structural coverage criteria, fault finding effectiveness tends to be relatively high when using the maximum oracle, even for test suites that perform poorly (less than 20%) when paired with the output-only oracle. This indicates that these test suites are effective at uncovering faults, as erroneous states do occur when they are executed, but are ineffective at propagating these faults to the outputs. Given this, the use of internal state in the test oracle becomes an effective method of improving testing effectiveness when using said test inputs.

Third, we note that when using the maximum oracle over the output-only oracle in conjunction with test inputs generated to satisfy requirements coverage metrics, the relative improvement is often slightly smaller than when using test inputs generated to satisfy structural coverage criteria. For example, for the *DWM_1* system, we can see the relative improvements for the requirements coverage metrics are less than those for the structural coverage metrics. Less obviously, for the *Latctl_Batch* and *Vertmax_Batch* systems, we see that the progression of relative (and absolute) improvement as we move from weaker to strong coverage criteria (i.e., from branch to MCDC and from naive requirements coverage to UFC) indicates that the test suites satisfying structural coverage criteria are more strongly influenced by the test oracle than those satisfying requirements coverage criteria.

Unfortunately, as shown in Table 6.2, we can also see the maximum oracle is between 12 and 207 times larger than the output-only oracle. This is discouraging; clearly, the fault finding improvements are desirable, particularly in the domain of critical systems. However, using the maximum oracle is likely more expensive than using the output-only oracle, as well as potentially infeasible given the likely unavailability of an oracle with knowledge of all internal variables.

6.3.3 Statistical Significance of Maximum Oracle Improvement

Table 6.1, along with Figure 6.2, provide evidence that considering internal state in test oracle may lead to improvements in testing effectiveness. We verify this evidence by testing the following hypotheses:

> **(H_1):** *The maximum oracle will have better fault finding than the output-only oracle.*

We evaluate H_1 for each system and coverage criterion combination using the paired permutation test, a non-parametric statistical test that calculates the probability p that two paired sets of data come from the same population [43]. We formulate the null hypothesis $H0_1$ for H_1 as follows:

> $H0_1$ *The maximum oracle will detect the same number of faults as the output-only oracle.*

To accept H_1, we must reject $H0_1$. Note that the output-only oracle cannot outperform the maximum oracle,as the set of variables used by the output-only oracle is always a subset of the variables used by the maximum oracle. We therefore equate rejecting $H0_1$ with accepting H_1 and use a one-tailed bootstrapped paired permutation test, exploring 250,000 permutations for each p-value [43].

To perform the one-tailed bootstrapped paired permutation test for a given system and coverage criterion, we pair the number of faults found by the output-only oracle with the number of faults found by the maximum oracle for all 50 reduced test suites, resulting in 50 pairs per coverage criterion and case example. We then run the permutation test and find that for all combinations of coverage criterion and case examples, the p-value is very small, less that 0.001. We therefore reject $H0_1$ with an $\alpha < 0.001$ and accept H_1 for each case example and coverage criterion.

Note that as in Chapter 5, we have not statistically generalized *across* case examples and coverage criterion, due the fact that both the case examples and coverage criteria are not randomly selected from their respective populations. This lack of generalization holds throughout this chapter.

6.3.4 Correlation of Oracle Size with Fault Finding

We have established that the maximum oracle is more effective than the output only oracle and quantified how test suite size influences the improvement, but the middle ground—oracles containing some, but not all internal state variables—has thus far been ignored. We begin by exploring the the strength of the correlation between oracle size and fault finding. We suspected it was strong and therefore explored the following hypothesis:

> **(H_2):** *There exists a strong and positive correlation between oracle size and fault finding.*

We evaluate H_2 by measuring the correlation between oracle size and fault finding for each case example and coverage criteria using each of the 50 reduced test suites. Correlation is measured using Spearman's rank correlation coefficient [43], a non-parametric measure of correlation. This yields 50 coefficients for each case example and coverage criteria combination. We list the minimum, mean, and maximum

	Branch	Condition	MCDC	Naive Req	Antecedent	UFC
DWM_1 System						
Min Corr	0.96	0.96	0.95	0.94	0.94	0.93
Mean Corr	0.96	0.96	0.96	0.94	0.94	0.93
Max Corr	0.97	0.97	0.96	0.95	0.94	0.93
DWM_2 System						
Min Corr	0.98	0.98	0.98	0.97	0.97	0.99
Mean Corr	0.99	0.99	0.99	0.97	0.97	0.99
Max Corr	0.99	0.99	0.99	0.97	0.97	0.99
Latctl_Batch System						
Min Corr	0.95	0.93	0.93	0.95	0.93	0.66
Mean Corr	0.95	0.94	0.94	0.96	0.95	0.67
Max Corr	0.95	0.94	0.94	0.97	0.96	0.70
Vertmax_Batch System						
Min Corr	0.98	0.93	0.93	0.97	0.97	0.82
Mean Corr	0.98	0.93	0.94	0.98	0.98	0.82
Max Corr	0.98	0.94	0.94	0.98	0.98	0.82

Table 6.3: Correlation of Oracle Size vs Fault Finding (Unrestricted Subtype)

observed coefficients for each combination, in Tables 6.3 and 6.4 for both the unrestricted and output-base subtypes, respectively. For every calculated coefficient, the p-value was less than 0.001 and thus statistically significant.

As shown, the observed correlation is consistently high, using both the unconstrained and output-base oracles, for all coverage criteria. (Note 0.7 or above is considered high correlation, with 0.9 or above being considered very high [28].) The exception to this are the correlations observed when using test suites generated to satisfy the UFC coverage criterion, which exhibit moderate to high correlations depending on the set of oracles used and the case example. This observation is consistent with the observations made in Sections 6.3.1 and 6.3.2—for these case examples, the test oracle has only a small influence of the effectiveness of test inputs generated to satisfy UFC coverage. Also note this is consistent with the overall high levels of fault finding observed for UFC. Even at small oracle sizes, test inputs satisfying UFC cov-

	Branch	Condition	MCDC	Naive Req	Antecedent	UFC
DWM_1 System						
Min Corr	0.93	0.91	0.89	0.91	0.88	0.83
Mean Corr	0.94	0.92	0.91	0.92	0.91	0.83
Max Corr	0.95	0.93	0.94	0.93	0.92	0.83
DWM_2 System						
Min Corr	0.98	0.98	0.98	0.96	0.96	0.98
Mean Corr	0.98	0.99	0.98	0.96	0.96	0.98
Max Corr	0.99	0.99	0.99	0.96	0.96	0.99
Latctl_Batch System						
Min Corr	0.97	0.97	0.96	0.96	0.97	0.79
Mean Corr	0.97	0.97	0.96	0.97	0.97	0.82
Max Corr	0.97	0.97	0.97	0.98	0.98	0.87
Vertmax_Batch System						
Min Corr	0.98	0.91	0.95	0.97	0.97	0.75
Mean Corr	0.98	0.93	0.95	0.98	0.98	0.75
Max Corr	0.98	0.93	0.95	0.98	0.98	0.75

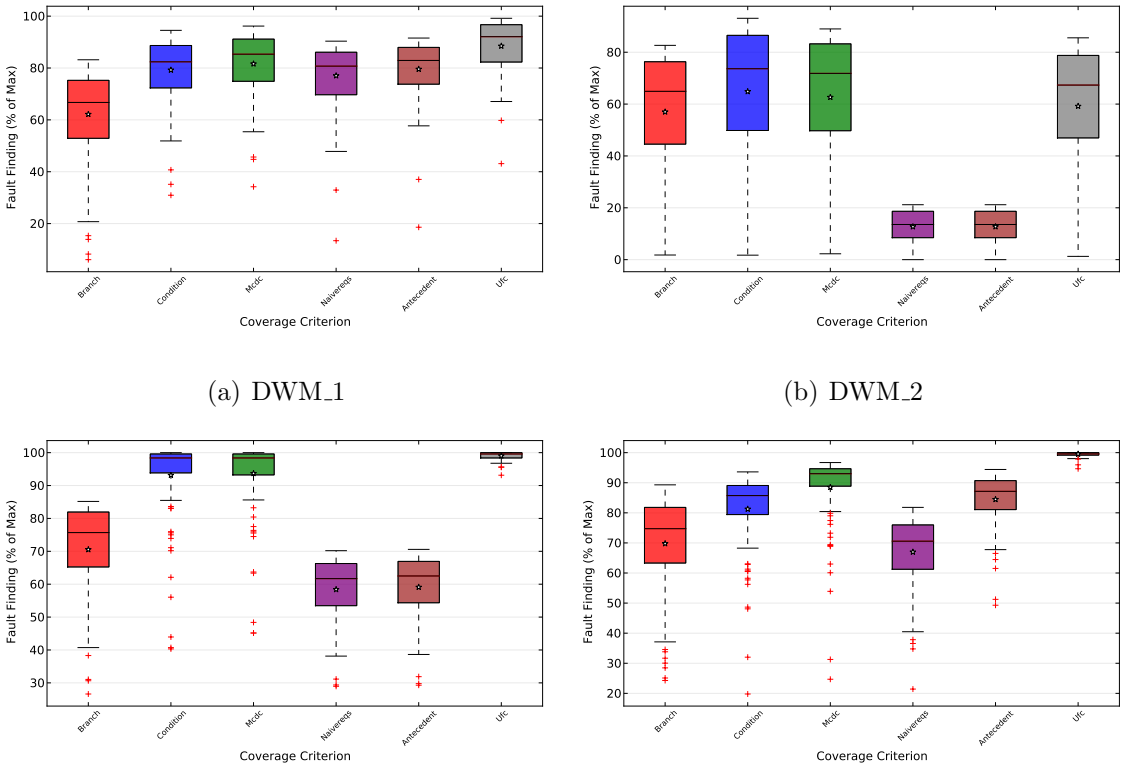
Table 6.4: Correlation of Oracle Size vs Fault Finding (Output-Base Subtype)

erage achieve high levels of fault finding; consequently, less improvement is possible when using larger oracle sizes, leading to large and small oracles achieving similar levels of fault finding, and thus relatively lower correlations.

Nevertheless, given the high correlation observed for most case examples and coverage criteria, and the statistical significance of our results, we *accept* H_2 and conclude that—as we suspected—oracle size is strongly correlated with fault finding for expected value oracles.

6.3.5 Relationship of Coverage Criteria, Oracle Size, and Fault Finding

In the previous sections, we have explored how oracle size impacts the effectiveness of the testing process given a fixed criterion, characterizing the shape of the the relationship between test oracle size and fault finding, and quantifying the potential improvements available when using internal state variables. In this section, we illus-



(a) DWM_1

(b) DWM_2

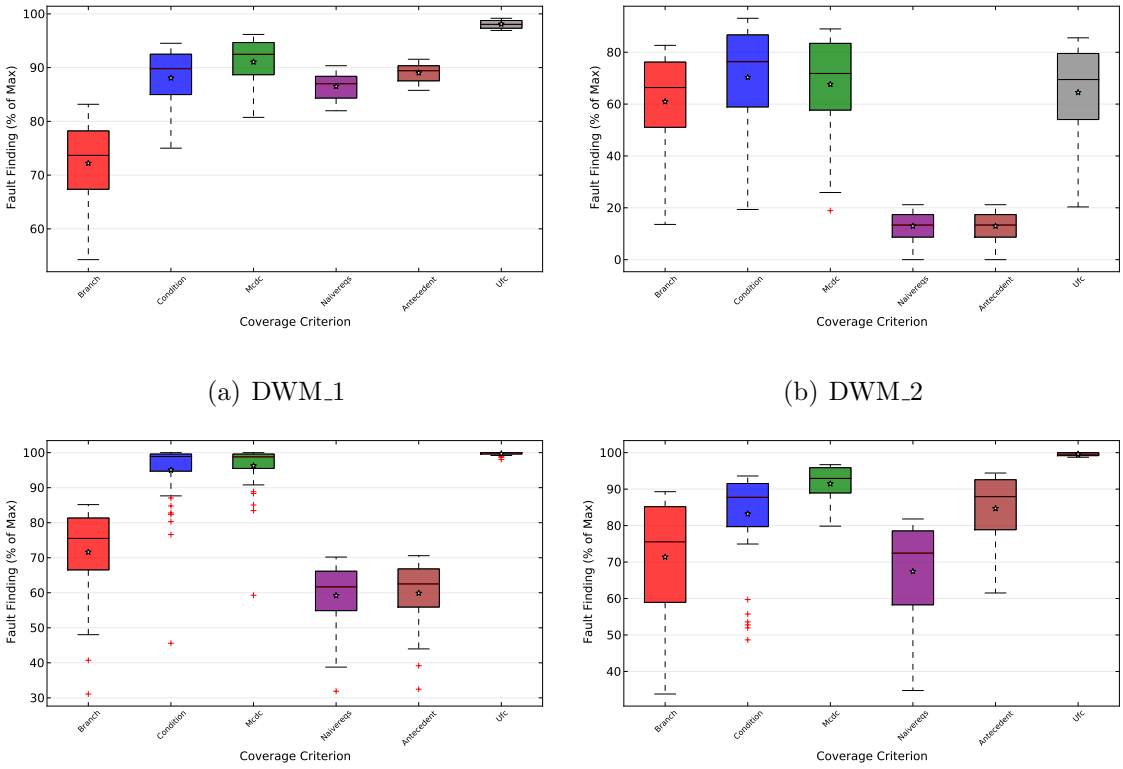
(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.3: Fault Finding Effectiveness Across All Oracle Sizes, Unrestricted Subtype

trate the potential interaction between the selection of a test coverage criteria and the selection of a test oracle.

In Figures 6.3 and 6.4, for each case example we present a boxplot illustrating, for each coverage criterion, the fault finding effectiveness of every combination of a reduced test suite satisfying the criterion with every test oracle (for both the unrestricted and output-base subtypes, respectively). In other words, we present *all* the fault finding numbers computed for each coverage criterion, across all test oracles. In these plots, box represents the *interquartile range*, or the *IQR*, of the data set, while the line within the box represents the median. The star represents the mean of



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.4: Fault Finding Effectiveness Across All Oracle Sizes, Output-base Subtype

the data, and the "whiskers", i.e. the lines extending from the box, capture all data points either 1.5×IQR above the 75% quartile mark or below the 25% quartile mark. Red + marks represent outliers, i.e., data values outside the whiskers Also note that, in general, the data points comprising the top of the box plot, i.e., the data points with high fault finding, are those achieved using large oracles, while those near the bottom are those achieved using smaller oracles.

We know from previous analyses that the choice of test coverage criteria and test oracle impacts the effectiveness of the testing process, with more rigorous criteria and larger test oracles leading to improved fault finding effectiveness. These figures,

however, illustrate the interaction between these artifacts, as both the criteria and oracle size are varied.

We draw two observations here. First, in many cases, there exists significant overlap in terms of fault finding effectiveness between criteria. In other words, for a given case example, there often several combinations of test coverage criteria and test oracles capable of achieving similar levels of fault finding. For example, in Figure 6.4 we see that for the *Latctl_Batch* system 80% fault finding can be achieved using five of the six coverage criteria when considering output-base oracles. For stronger coverage criteria, this can likely be achieved using weaker test oracles; for weaker coverage criteria, this can be achieved using stronger test oracles. Similar observations can be made for each case example when considering both unrestricted and output-base test oracles. In particular, we note the structural coverage criteria tend to overlap substantially, with branch coverage and MCDC coverage overlapping in every case example and oracle subtype (albeit sometimes only with respect to the very best data point in branch coverage and the very worst data points in MCDC coverage).

Second, more rigorous coverage criteria often exhibit a smaller range of fault finding values across oracle sizes as measured by the inter-quartile range (IQR). For example, for each of the case examples, as we progress from branch, to condition, to MCDC coverage, the range of fault finding values generally drops, particularly when considering the output-base subtype. For example, for the *DWM_1* system using output-base oracles, branch coverage has an IQR of roughly 10%, condition coverage has an IQR of roughly 7%, and MCDC coverage has an IQR of roughly 5%. This reduction in ranges occurs despite the fact that none of the structural coverage criteria achieve 100% fault finding, and thus larger IQRs are possible. Similar observations can be made concerning requirements coverage criteria, with the exception of naive requirements and antecedent coverage for the *DWM_2* system; these coverage criteria

perform poorly with respect to this system, achieving less than 20% fault finding at best.

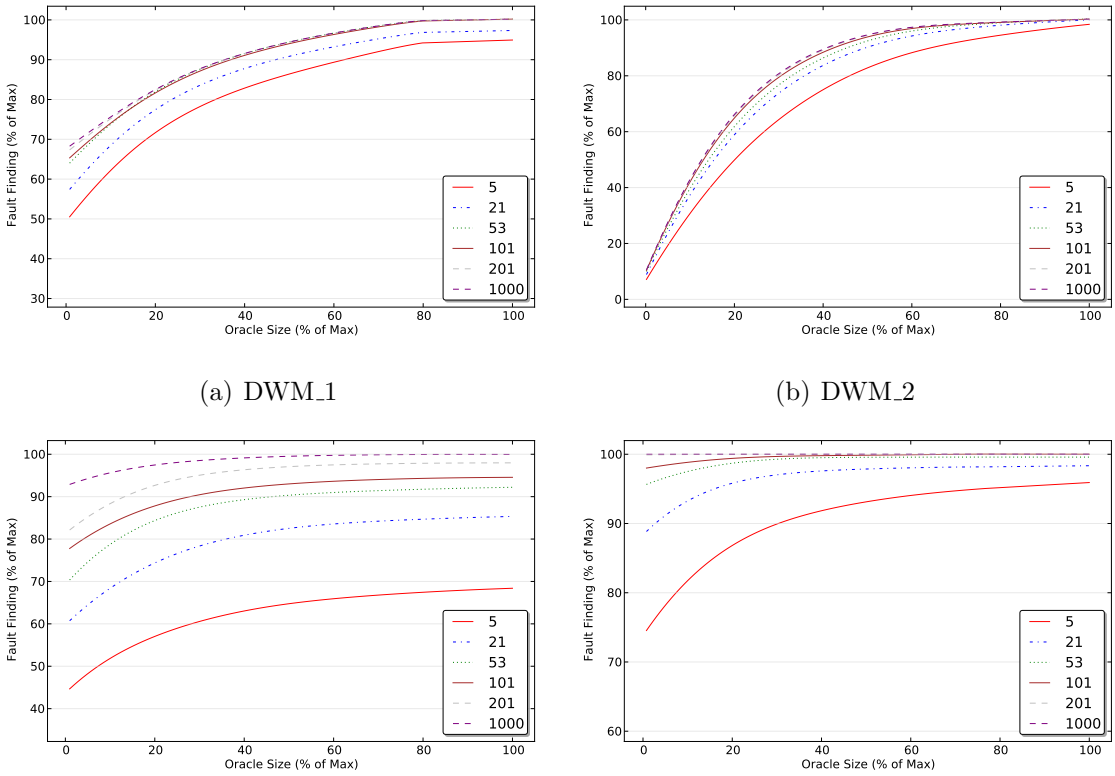
These observations highlight the potential tradeoffs that can be made in software testing. Depending on the resources available, and the nature of costs in our testing process, we may wish to use a relatively weaker coverage criterion with a strong test oracle, or a stronger coverage criterion with a weaker test oracle, or some combination in-between. We discuss the implications of this in Section 6.6.

6.4 Influence of Oracle Data for Randomly Generated Test Suites

In the previous section, we explored how the test oracle influences the effectiveness of the testing process using a number of coverage criteria. However, we also interested in how the number of tests and the size of the test oracle interact to influence the effectiveness of testing (per Question 4). When using coverage criteria, our ability to directly vary the test suite is limited—we can vary the criteria our test suite satisfies, but we cannot, for example, directly vary the size of the test suite (assuming we use nonreducible test suites). In this section, we explore the influence of test oracles in the context of random testing, thus allowing us to directly explore how test suite size and oracle size jointly influence the effectiveness of the testing process.

6.4.1 Influence of Oracle Size Given Fixed Test Data

In Figures 6.5 and 6.6, we illustrate the relationship between oracle size and fault finding using randomly generated test suites and all four case examples. Recall that we generated 200 random test suites of sizes varying between 5 and 1,000 random tests. Rather than plot how every test suite is influenced by oracle size (which would



(a) DWM_1

(b) DWM_2

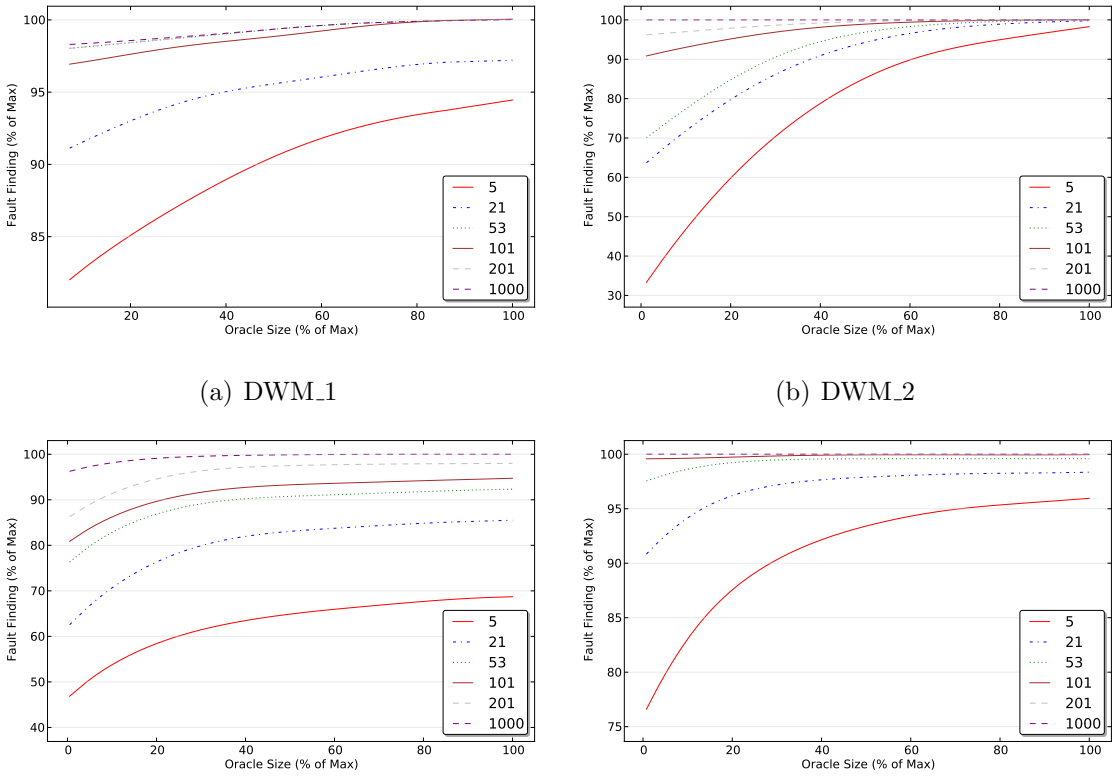
(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.5: Random Tests, Oracle Size vs Fault Finding, Unrestricted Subtype

require a prohibitive number of figures), we have selected a set of test suite sizes we have found to be representative. Note that we focus on smaller test suite sizes as larger test suite sizes (> 200 tests) tend to exhibit similar levels of fault finding in our case examples. As in Subsection 6.4.1, we have smoothed the plots using LOESS smoothing [43] with a bandwidth factor of 0.1.

We make two observations from these figures. First, as with test suites generated to satisfy coverage criteria, the relationship between fault finding and oracle size is generally logarithmic, with a rapid increase in fault finding for small oracle sizes coupled with a gradual drop in the rate of increase for larger oracle sizes. Consequently,



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.6: Random Tests, Oracle Size vs Fault Finding, Output-base Subtype

as before, there exists substantial overlap in the faults detected by variables within the system.

Second, we note that the potential improvements available from using larger test oracles diminishes as test suite size grows. This is perhaps best illustrated by the *DWM_2* system when using output-base oracles—every test suite generated is capable of achieving close to the maximum level of fault finding, but while the largest test suite achieves this with the output-only oracle, the smallest test suite achieves this only when every variable in the system is part of the oracle data.

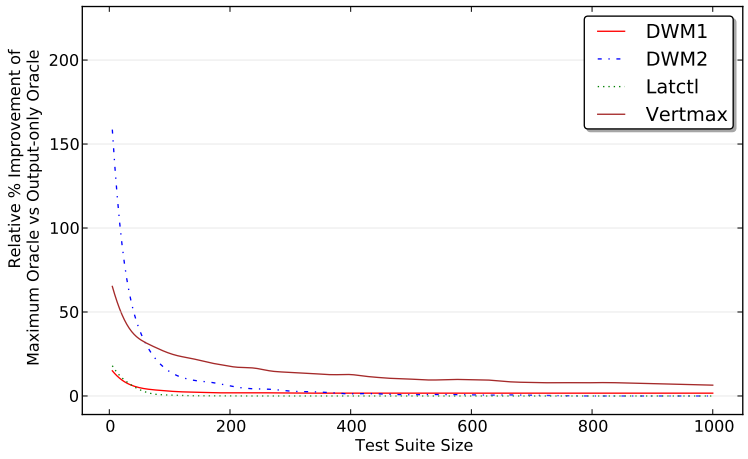
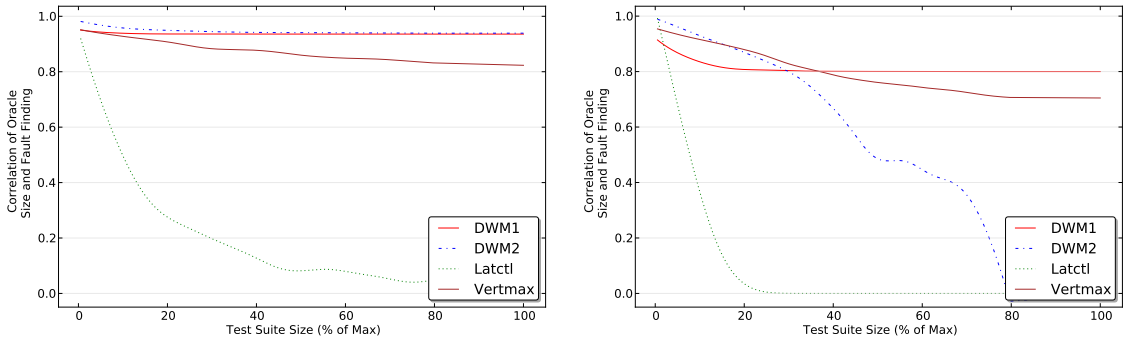


Figure 6.7: Relative Improvement of Maximum Oracle Over Output-only Oracle vs Test Suite Size

6.4.2 Relative Improvement using Maximum Oracle

Concerning Figure 6.6, we noted that test suite size and the potential improvements available from using larger test oracles appear to be linked. Of particular interest is how the relative improvement when using the maximum oracle over the output-only oracle varies with test suite size—we would like to know how the effectiveness of considering internal state changes as test suite size varies. Accordingly, in Figure 6.7, we plot the relative improvement in fault finding that occurs when using the maximum oracle over the output-only oracle for randomly generated test suites of size 5 to 1,000. We have again smoothed the plots using LOESS smoothing [43] with a bandwidth factor of 0.1.

For each case example, we can see that the relative improvement when using a maximum oracle starts fairly high, over 150% for the *DWM_2* system and over 20% for all other systems. As test suite size increases, however, the relative improvement drops quickly as we approach 100 tests (very quickly for *DWM_1* and *Latctl_Batch*),



(a) Unrestricted Subtype

(b) Output-base Subtype

Figure 6.8: Correlation of Oracle Size vs Fault Finding as Test Suite Size Varies

dropping below 10% for all case examples at 1,000 tests. This indicates that when using small test suites, the choice of test oracle is potentially highly influential, but less so when using larger test suites. This is in keeping with our observations concerning coverage criteria from Section 6.3.2, in which we noted test suites satisfying stronger coverage criteria are less influenced by the selection of the test oracle than tests suites satisfying weaker coverage criteria (e.g., MCDC versus branch coverage).

6.4.3 Correlation of Oracle Size and Fault Finding

As we have shown, the improvements in testing effectiveness available from using larger test oracles tend to diminish as test suite size increases. This can be further illustrated by exploring the correlation between fault finding as oracle size for each random test suite. In Figure 6.8, we plot the correlation of oracle size and fault finding (again using Spearman's rank correlation coefficient [43]) for each case example using both the unconstrained and output-base oracle subtypes. For every calculated coefficient, the p-value was less than 0.001 and thus statistically significant.

As shown, for both subtypes, correlation is very high for small test suite sizes (as was the case for test suites generated to satisfy coverage criteria). However,

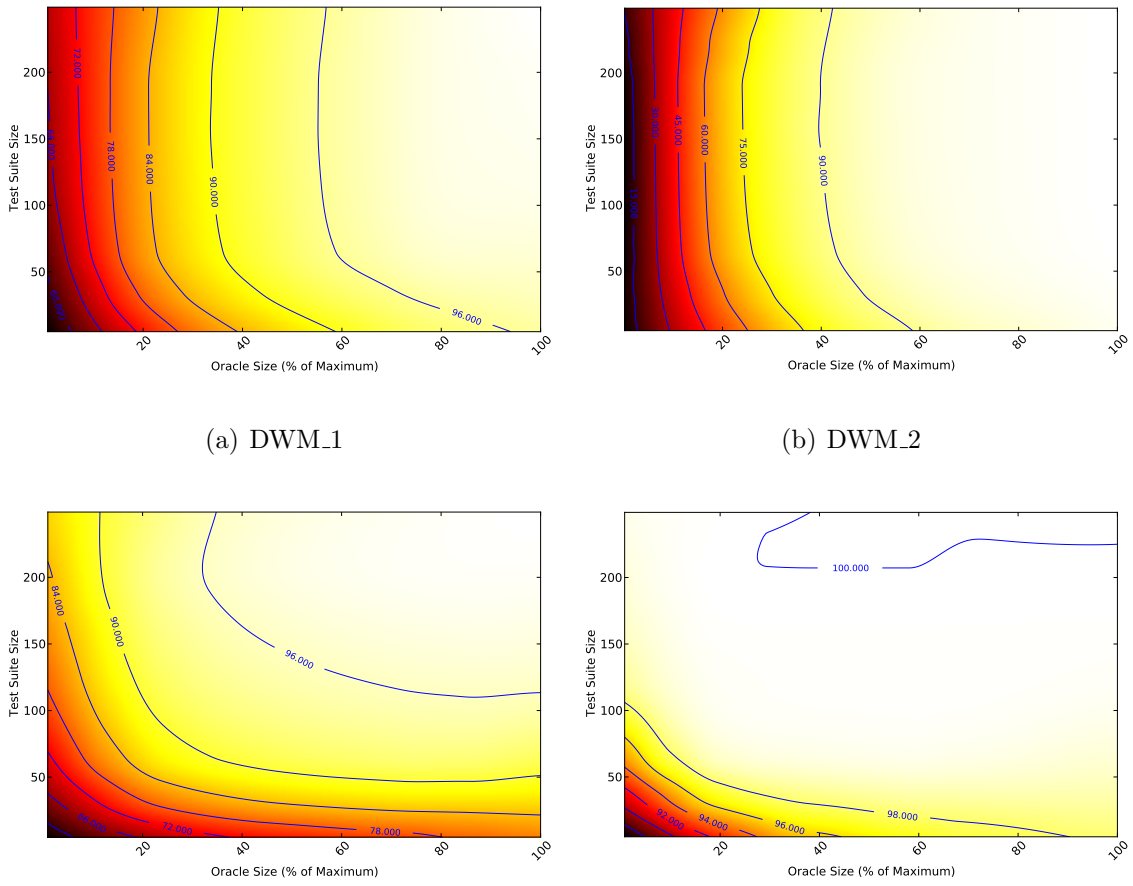
as test suite size increases, correlation drops for each case example. *Latctl_Batch* aptly demonstrates this in both subtypes—even with relatively small test suite sizes (10-20% of the largest test suite size) correlation drops to below 0.6. Other case examples demonstrate less dramatic drops in fault finding, but nevertheless, we see the effectiveness of larger test oracles drops as test suite size increases.

6.4.4 Relationship of Random Test Suite Size, Oracle Size, and Fault Finding

In Figures 6.9 and 6.10, we present the relationship between test suite size, oracle size and fault finding as a contour map for the unrestricted and output-base oracles, respectively. For these figures, we chose to focus on smaller test suite sizes (< 250) as the more interesting observations apply mostly to smaller test suite sizes. Oracle sizes and fault finding measurements are given as a percentage of their respective maximums.

As with several previous figures, these figures have been smoothed using LOESS (with a smoothing factor of 0.20). In these figures, the light areas represent combinations of test suites and oracles that reveal a relatively high number of faults, and dark areas represent combinations of test suites and oracles that reveal a relatively low number of faults. Each contour line represents a constant level of fault finding. For example, we can see that for the *DWM_1* system in Figure 6.10, 50 tests and the output-only oracle reveals roughly the same number of faults as 25 tests and an output-base oracle containing 60% of the possible variables (94% of the faults).

We make three observations concerning these figures. First, the tradeoffs implied in previous results are easily seen here. We have now clearly illustrated how the selection of test suite size and test oracle size influences the effectiveness of the testing process. For example, for the *DWM_2* system, we can see that in order to achieve



(a) DWM_1

(b) DWM_2

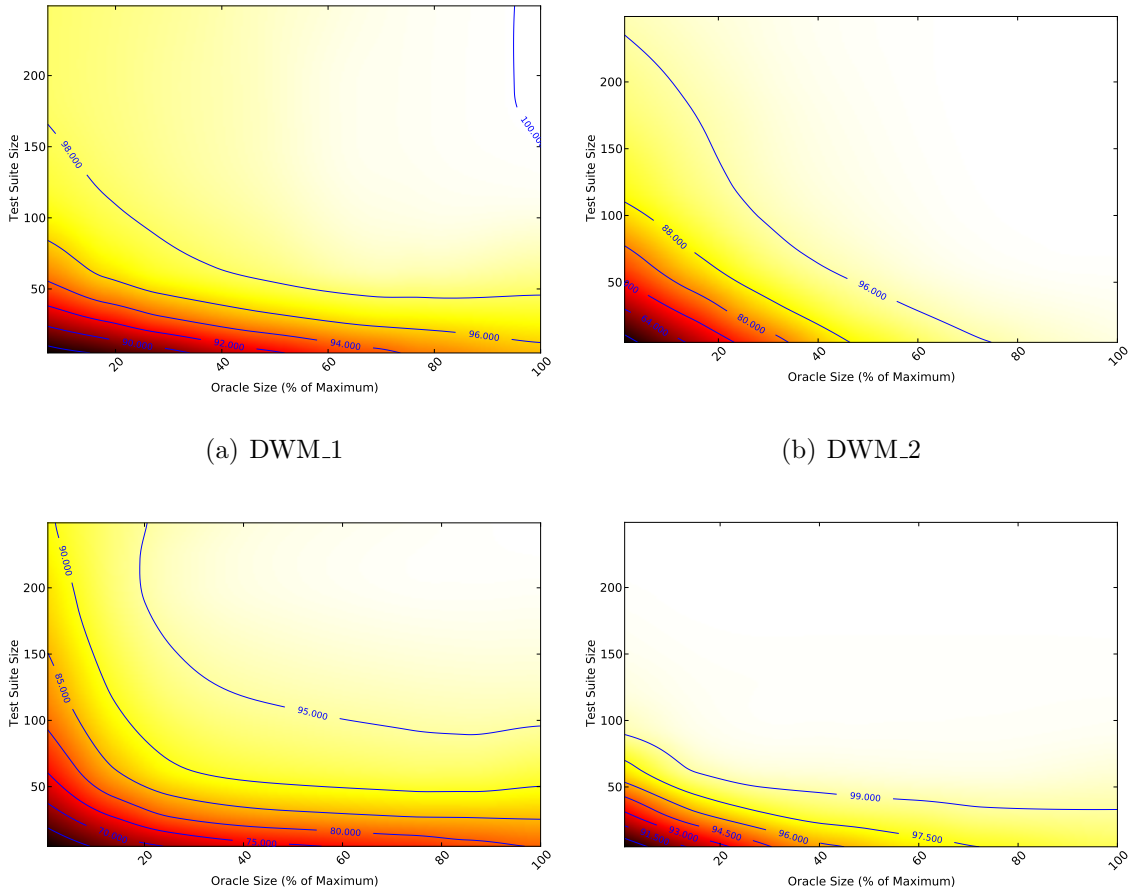
(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.9: Relationship of Test Suite Size, Oracle Size and Fault Finding, Unrestricted Subtype

96% fault finding when considering output-base oracles, we could use (1) a relatively large test suite ($>$ 225 tests) and the output-only test oracle, or (2) a relatively small test suite ($<$ 25 tests) and the maximum test oracle, or (3) some intermediate combination.

Second, given a set cost function and finite resources, the most effective test suite and oracle combination can vary depending on how test suite size and oracle size



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.10: Relationship of Test Suite Size, Oracle Size and Fault Finding, Output-base Subtype

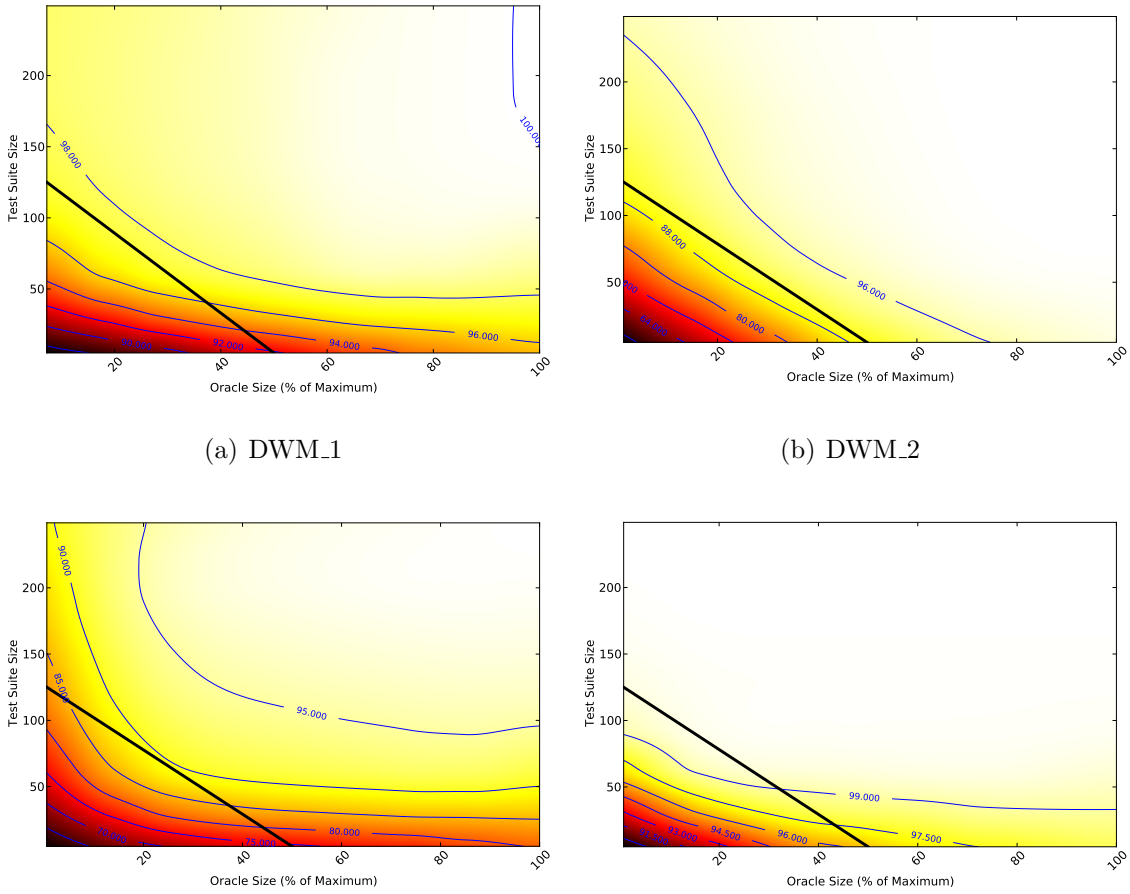
jointly influence the number of faults revealed. In other words, the potential for a tradeoff is determined by the relative power of the test suite and the test oracle. For example, assume that our cost budget is such that we can use an output-base oracle considering roughly 50% of the internal state while running a single test, or use the output-only oracle while running 125 random tests, or some linearly related combination thereof. We plot this cost function as a thick black line for each system

in Figure 6.11. Note that these contour plots use output-base oracles.

Examining these lines, we can see that the most cost effective choice can vary substantially depending on the specific system under consideration. For example, we can see that for the *Latctl_Batch* and *DWM_1* systems, the best choice for our given cost function is the output-only oracle with 125 tests, yielding 99% and 97% fault finding, respectively. In contrast, for the *Vertmax_Batch* system, the best choice is an output-base oracle with roughly 30% of the internal state and roughly 75 tests, yielding 90% fault finding. If we were to change our choice for *Vertmax_Batch*—using an output-only oracle with 125 tests—fault-finding would drop to less than 85%, a reduction of 5% points.

Third, these figures highlight the importance of composition of test oracles, i.e., how outputs and internal state variables impact the effectiveness of the testing process. This can be seen in the differences between Figures 6.9 and 6.10, representing the unrestricted and output-only test oracles—specifically, the differences between the *DWM_1* and *DWM_2* case examples across oracle subtypes. In the case of the unrestricted oracle subtype, the contour maps for these case examples consist largely of vertical lines, indicating that improvements in fault finding are primarily available by increasing oracle size. Conversely, in the case of the output-base oracle subtype, the contour maps for these case examples consist of more horizontal (though not completely horizontal) lines, indicating that increased test suite size is (depending on the cost function) more likely to lead to improvements in fault finding.

The difference here is the use of output variables in the oracle data. In the case of the unrestricted oracle subtype, the use of outputs is not guaranteed and small oracles tend to perform poorly. Therefore, regardless of the number of tests used, fault finding will be on average low unless a sufficiently large oracle is used. When using output-base oracles, however, we achieve good (if not ideal) fault finding merely



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.11: Cost Function w/ Relationship of Test Suite Size, Oracle Size and Fault Finding, Output-base Subtype

by the inclusion of outputs in the oracle data, and test suite size is generally more important than oracle size with respect to effectiveness.

6.4.5 Regression Modelling of Random Test Suite Size, Oracle Size, and Fault Finding

The previous visualizations and analyses highlight the interrelationship between test suite size and oracle size, and hint at the relationship between these two factors, but their exact nature is unclear. Here we explore the use of linear regression to attempt to model how random test suite size and oracle size jointly influence the effectiveness of the testing process.

In linear regression, we model the data as a linear equation $y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon_i$ in which the variables x_i correspond to factors (independent variables) and the variable y denotes the dependent variable. The goal is to determine the coefficients β_i that best fit the data.

Following the approach used by Namin and Andrews for modeling effectiveness in terms of test suite size and branch coverage [49], we fitted several linear models of fault finding (FF) to the collected data, with the goal of (1) determining if the models considering both test suite size (TS) and oracle size (OS) outperformed models considering either factor alone and (2) determining which model transformation best fit the data.

We explored several linear regression models, trying each possible combination of *TS*, *log(TS)*, *OS*, and *log(OS)* with respect to fault finding (FF). We then computed the adjusted R^2 to determine the goodness of fit for each model. In Tables 6.5 and 6.6, we list the adjusted R^2 values for all models explored for the unrestricted and output-base subtypes, respectively.

Our results indicate that with respect to adjusted R^2 and when considering only output-base oracles, models of the form $FF = \beta_1 \cdot TS$ or $FF = \beta_1 \cdot log(TS)$ outperform models of the form $FF = \beta_2 \cdot OS$ or $FF = \beta_2 \cdot logt(OS)$. Thus if we assume an output-base oracle, test suite size is a better predictor of testing effectiveness than oracle size.

	DWM_1	**DWM_2**	**Latctl_Batch**	**Vertmax_Batch**
TS	0.00	0.00	0.07	0.26
log(TS)	0.00	0.00	0.25	0.50
OS	0.72	0.63	0.01	0.17
log(OS)	0.76	0.90	0.02	0.28
TS + OS	0.72	0.63	0.09	0.43
TS + log(OS)	0.76	0.91	0.10	0.54
log(TS) + OS	0.72	0.63	0.27	0.67
log(TS) + log(OS)	0.77	0.91	0.28	0.78

Table 6.5: Adjusted R^2 Goodness of Fit for Models of Fault Finding, Unrestricted Subtype

TS = Test Suite Size, OS = Oracle Size

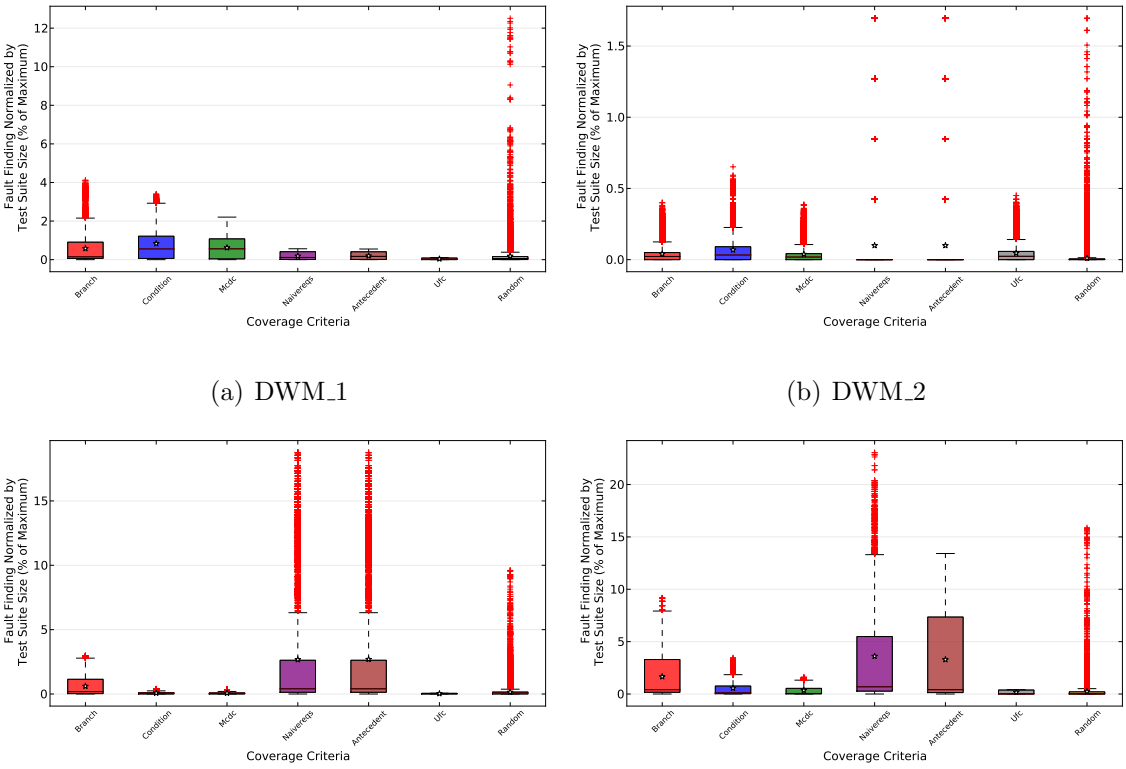
	DWM_1	**DWM_2**	**Latctl_Batch**	**Vertmax_Batch**
TS	0.09	0.14	0.07	0.32
log(TS)	0.33	0.32	0.28	0.62
OS	0.17	0.10	0.01	0.08
log(OS)	0.17	0.12	0.01	0.15
TS + OS	0.27	0.24	0.08	0.40
TS + log(OS)	0.27	0.27	0.09	0.47
log(TS) + OS	0.50	0.42	0.29	0.71
log(TS) + log(OS)	0.50	0.44	0.29	0.78

Table 6.6: Adjusted R^2 Goodness of Fit for Models of Fault Finding, Output-base Subtype

TS = Test Suite Size, OS = Oracle Size

For the unrestricted subtype, no consistent pattern for single factor models can be seen—depending on the case example, the use of either oracle size or test suite size may be a better predictor than the alternative.

However, for both oracle subtypes, we found models of the form $FF = \beta_1 \cdot log(TS) + \beta_2 \cdot log(OS)$ performed better than or equal to all other models explored. This provides additional evidence that both test suite size and oracle size influence the effectiveness of the testing process. Note the R^2 value observed for this model varies considerably across all case examples, indicating in turn that the relationship



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 6.12: Fault Finding Effectiveness of Individual Variables for Each Coverage Criterion

between test suite size, test oracles, and fault finding also varies across case examples. This is in keeping with our observations from Section 6.4.4 (and indeed, throughout this chapter).

6.5 Effectiveness of Individual Variables in Oracle Data

In previous sections, we have examined the effectiveness of test oracles of various sizes. Also of interest (per Question 2) is how individual variables contribute to the effectiveness of the testing process. We therefore examined the fault finding ability

of each individual variable for every case example using every test suite, i.e., we investigated the fault finding ability of every oracle of size 1.

Once we had computed these fault finding measurements, we then normalized each measurement using the size of the test suite corresponding to the measurement. In other words, we divided each effectiveness measurement by the size of the corresponding test suite used. This was done to control for varying test suite sizes across coverage criterion. In the absence of this normalization, a given variable will nearly always detect more faults when using stronger coverage criteria (or larger random test suites) simply because more test inputs have been run.

We plot these results for each case example in Figure 6.12. These boxplots are similar to those in Figures 6.3 and 6.4 (except for the normalization). Also note that for this analysis, we use both random test suites and test suites generated to satisfy coverage criteria.

We make two observations concerning the effectiveness of individual variables. First, we note that for each case example, the median normalized effectiveness of an individual variable tends to be nearly the same, and very low—close to zero in all case examples and coverage criteria. This indicates that in general, variables tend to be marginally effective individually, with many variables finding few, if any faults, irrespective of the coverage criterion used.

Second, we note that for each case example, there often exist a large number of outlier variables finding far more faults than the median or average fault finding for a variable. Coupled with previous observation, we can see that for each case example, while many variables tend to be very ineffective as part of the oracle data, some variables are exceptionally effective, finding several times (up to 10 times more) faults per test on average. We term these variables *critical-variables*, and define them to be variables whose fault finding is more than 2 standard deviations better than

	Branch	Condition	MCDC	Naive Req	Antecedent	UFC	Random
DWM_1 System							
μ	0.57	0.83	0.61	0.18	0.19	0.04	0.18
σ	0.85	0.91	0.63	0.19	0.19	0.03	0.58
Median	0.14	0.55	0.56	0.11	0.16	0.03	0.05
Avg % Out <= 0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Avg % In <= 0	4.86%	4.34%	6.08%	6.90%	6.53%	4.34%	4.34%
Avg % Out $> \mu + \sigma$	44.44%	44.44%	44.44%	66.66%	51.77%	66.66%	66.66%
Avg % In $> \mu + \sigma$	12.83%	17.28%	19.07%	23.09%	28.05%	28.57%	32.75%
Avg % Out $> \mu + 2\sigma$	37.77%	11.11%	2.0%	0.0%	0.0%	0.0%	0.05%
Avg % In $> \mu + 2\sigma$	8.17%	4.33%	1.21%	0.0%	0.0%	0.0%	0.0%
DWM_2 System							
μ	0.04	0.06	0.03	0.09	0.09	0.04	0.00
σ	0.05	0.09	0.05	0.24	0.24	0.06	0.04
Median	0.02	0.03	0.01	0.0	0.0	0.02	0.00
Avg % Out <= 0	0.0%	0.0%	0.28%	100.0%	100.0%	0.0%	0.0%
Avg % In <= 0	37.97%	34.09%	36.73%	82.24%	82.24%	35.13%	32.17%
Avg % Out $> \mu + \sigma$	68.57%	95.42%	69.42%	0.0%	0.0%	100.0%	100.0%
Avg % In $> \mu + \sigma$	10.97%	11.72%	11.84%	17.75%	17.75%	10.65%	6.31%
Avg % Out $> \mu + 2\sigma$	33.42%	76.28%	59.42%	0.0%	0.0%	89.71%	99.85%
Avg % In $> \mu + 2\sigma$	6.17%	5.83%	6.19%	4.74%	4.74%	5.18%	2.94%
Latctl_Batch System							
μ	1.65	0.55	0.34	3.60	3.28	0.16	0.24
σ	2.05	0.81	0.45	5.05	4.08	0.18	0.82
Median	0.41	0.10	0.05	0.68	0.41	0.01	0.01
Avg % Out <= 0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Avg % In <= 0	6.25%	6.25%	6.25%	7.68%	6.25%	6.25%	6.25%
Avg % Out $> \mu + \sigma$	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Avg % In $> \mu + \sigma$	25.0%	17.18%	15.62%	19.84%	26.70%	33.40%	37.62%
Avg % Out $> \mu + 2\sigma$	100.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
Avg % In $> \mu + 2\sigma$	4.68%	8.73%	8.85%	5.95%	0.09%	0.0%	0.0%
Vertmax_Batch System							
μ	0.60	0.07	0.06	2.67	2.69	0.03	0.14
σ	0.70	0.08	0.07	4.49	4.51	0.02	0.36
Median	0.18	0.06	0.05	0.40	0.40	0.03	0.08
Avg % Out <= 0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Avg % In <= 0	21.87%	14.21%	13.97%	22.55%	21.88%	13.97%	14.06%
Avg % Out $> \mu + \sigma$	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Avg % In $> \mu + \sigma$	14.93%	9.88%	10.60%	15.58%	15.81%	18.69%	15.19%
Avg % Out $> \mu + 2\sigma$	50.0%	0.0%	57.0%	97.0%	96.0%	0.0%	9.5%
Avg % In $> \mu + 2\sigma$	5.54%	7.18%	7.56%	10.60%	10.52%	0.0%	2.57%

Table 6.7: Fault Finding Effectiveness of Individual Variables by Coverage Criterion
Out = Output Variables, In = Internal State Variables

the median.

In Table 6.7, we quantify our observations, listing for each case example and coverage criterion: (1) the number of variables which detect no faults on average (2) the mean (μ) and standard deviation (σ) in fault finding effectiveness for individual variables, and (3) the percentage of internal state (In) and output variables (Out)

which find 1 or 2 standard deviations more faults than the mean on average. As we can see, under our definition of a critical-variable, for these case examples up to 100% of output variables and 10.6% of internal state variables, respectively, are critical-variables for a given test suite (on average). For each case example, we can see these critical-variables reveal roughly 3 times as many faults as the average variable, though for some case example and coverage criterion combinations—random test suites used in conjunction with the *Latctl_Batch* system, for example—the mean and standard deviation are such that critical-variables find far more faults than the average variable,.

Note that many of these critical variables are output variables, which provides evidence that current testing practice in the avionics domain—which relies on output-only oracles—is reasonable. However, for each case example there exists several internal state variables that are also critical variables. These variables represent an opportunity to improve fault finding through methods of efficiently detecting these variables and adding them to the oracle data.

6.6 Discussion

We now discuss the questions posed at the start of this chapter, and highlight the implications of our results:

Question 1 (Q1): How does oracle size influence the effectiveness of the testing process given a fixed test suite?

Question 2 (Q2): How much do individual variables in the oracle data contribute to the effectiveness of the testing process?

Question 3 (Q3): In the case of tests generated to satisfy a coverage criterion, how

does the coverage criterion used and oracle data chosen jointly influence the effectiveness of the the testing process?

Question 4 (Q4): In the case of random testing, how does the size of the test suite and oracle data chosen jointly influence the effectiveness of the the testing process?

At this point, it is worth re-iterating that while we achieve consistent results across case examples for some analyses, we are only using four models from a single domain to explore these questions. Generalizability is thus limited to this domain.

6.6.1 Influence of Oracle Size Given a Fixed Test Suite

Our results indicate that given a fixed test suite, an increase in oracle size corresponds to an increase in fault finding, often with very high correlation (> 0.9). This increase, shown in Figures 6.1, 6.2, 6.5, and 6.6 can be characterized as logarithmic, with an initial rapid increase in testing effectiveness as oracle size grows from smaller oracles to larger oracles, followed by a more gradual climb in oracle effectiveness. This relationship can be observed largely irrespective of the oracle subtype, with only minor differences noted across subtypes (explored later in this section). Note that for some coverage criteria and case example pairings, fault finding is initially very high for small oracles; accordingly, fault finding cannot increase as oracle size increases, and thus oracle size does not impact effectiveness.

While this relationship retains the same basic shape across coverage criteria and case examples, we note that subtle differences are apparent depending on the case example. For example, the relationship for the *DWM_1* system is generally flatter, with less of a rapid increase and "peaking" behavior, while the relationship for the *Vermatx_Batch* system is just the opposite, with an exceptionally sharp rise that quickly flattens out.

As previously noted, the use output-only oracles appears to be standard practice in industry, and we are thus interested in how the addition of internal state to output-only oracles can impact the effectiveness of the testing process. Our results in Table 6.1 indicate that for a fixed test suite, the relative improvement in considering internal state varies substantially depending on the case example and coverage criterion considered, with improvements as high as 370% for MCDC test suites and the *DWM_2* system, and as low as 0% when using very large randomly generated test suites for all systems except *Vertmax_Batch*. Thus, while the addition of internal state to an output-only oracle can be a powerful method of improving the effectiveness of the testing process, it may also provide little or no benefit.

Nevertheless, these results provide strong evidence that, in practice, larger oracle size often leads to improvements in fault finding effectiveness, supporting the theoretical results from Chapter 3. These results imply that, given a set of tests, we can often improve the effectiveness of our testing process by increasing oracle size. Furthermore, the logarithmic shape of the relationship between oracle size and fault finding indicates that, in general, smaller oracle sizes may be more cost effective than larger oracle size (assuming cost scales linearly with the size of the oracle data), thus providing a general guideline for oracle selection: use smaller oracles, as larger oracles are often only marginally more effective. Given the variation in the results, however, it is clear more sophisticated methods of selecting test oracles are needed.

6.6.2 Impact of Individual Variables of Effectiveness of Oracle Data

For all case examples, the median fault finding effectiveness for an oracle of size 1 (i.e., for an individual variable) was uniformly low, with a fault finding effectiveness of less than 1.0 per test case. In other words, at least half of variables, when paired with an arbitrary test suite, average less than 1 fault detected per test run. Furthermore, a

significant percentage of variables find no faults at all—in particular, internal state variables perform poorly individually, with 4.34% to up to 82.24% finding no faults (depending on the case example and coverage criterion). Thus, our analysis indicates that many variables are of only marginal use in the oracle data.

However, for each system, we also noted outlier variables which find far more faults than the average variable. We termed such variables *critical-variables*, and define them to be variables finding greater than two standard deviations more faults than the average variable. For each case example, the critical-variables comprise only a small percentage of the overall set of variables, tend to be output variables rather than internal state variables, and find between 3 to 10 times as many variables as the average variables.

Clearly, the inclusion—or exclusion—of these critical-variables has a potentially significant impact on the effectiveness of the test oracle. If it were possible to select these critical-variables quickly (i.e., without conducting a large scale empirical study such as this one) this might lead to a method of constructing small—but highly effective—oracle data for expected value oracles. Such a method might allow us to obtain most of the benefits of large oracles, without the considerable expense.

6.6.3 Joint Influence of Coverage Criteria and Oracle Data

As the coverage criteria varies, the influence of the oracle data, with few exceptions, remains fairly consistent. The relationship between oracle size remains largely logarithmic, the correlation remains high (excepting those cases in which small oracles are very effective), and relative improvement in using the output-base and maximum oracle remains high (again, excepting when small oracles are very effective).

However, there do exist subtle changes in how oracle selection influences effectiveness as the coverage criteria varies. First, as the rigor of the coverage criterion

increases, the potential relative improvement in considering internal state (i.e., the difference between the output-base and maximum oracle) decreases. This can be seen by considering the progression of rigor for the structural and requirements coverage metrics, e.g., branch → MCDC and naive requirements coverage → UFC—as more rigorous coverage criterion are employed, the relative improvement universally drops.

This drop can likely be attributed at least in part to the effectiveness of test suites satisfying more rigorous coverage criteria. When using the output-only oracle, fault finding is higher for test suites satisfying, for example, MCDC versus those satisfying branch coverage; consequently, the potential increase when using the maximum oracle is lessened. It is also possible that the increased rigor plays a part—stronger test inputs leave less room for larger oracles to influence effectiveness.

In either case, we can see that when using test suites satisfying more rigorous coverage criterion, the potential benefits of careful oracle selection is reduced. The implications of this are twofold. First, it implies that when testing, testers may be able to place less effort in oracle selection provided a rigorous coverage criteria is used. Second, it highlights the importance of considering multiple factors in software engineering studies—had we conducted this study using only one type of coverage criterion (for example, UFC or branch coverage), our results might have led us to conclude that the influence of test oracles on testing effectiveness is very weak or strong, when it is in fact quite variable.

Potential Tradeoffs in Oracle Size / Test Suite Power

The interaction between test suites and test oracles is best illustrated in Figures 6.3 and 6.4, which plot the fault finding effectiveness of all test suites using all test oracles. Here we can see the significant overlap between coverage criteria, with similar levels of fault finding achievable across different coverage criteria. These results indicate a

potential tradeoff in testing—we can achieve similar levels of fault finding using test suites satisfying strong coverage criteria with small oracles, or large oracles with test suites satisfying weaker coverage criteria, or some intermediate combination.

Thus, depending on the testing resources available, we can select the most cost effective combination of test inputs and test oracle from several potential options. For example, if the large test oracles were viewed as cheap the use (as might occur when a model suitable for use as a test oracle and hardware facilitating efficient monitoring during testing were both available), we might opt to use a weaker, cheaper coverage criterion. In a different scenario in which test oracle size was capped at relatively small size (as might occur when testing embedded systems), we might opt to use a stronger coverage criterion. We further discuss these potential tradeoffs later in this section.

Differences in Structural and Requirements Coverage Criteria

As illustrated in Figure 6.2 and detailed in Table 6.1, the potential benefit of considering internal state is less when using test suites generated to satisfy a requirements coverage criterion as compared to when using test suites generated to satisfy structural coverage criteria. This can be seen by comparing the relative improvements across structural and requirements coverage metrics for each case example. Even when fault finding effectiveness is similar for two structural and requirements coverage criteria–for example, for naive requirements and branch for the *Latctl_Batch* system, and naive requirements and MCDC for the *DWM_1* system—the relative improvement is always less for the requirements coverage metric.

This can be attributed to the nature of these different types of coverage criteria. Tests satisfying structural coverage criteria are focused on exercising syntactic elements of the source code, and, accordingly, consist of short tests covering some branch

or combination of conditions. These test inputs may be effective at uncovering faults, but may not be effective—due to length and their focus on internal expressions—at propagating these errors to the outputs. Consequently the use of internal state can significantly improve their effectiveness.

In contrast, tests satisfying requirements coverage criteria are formulated in terms of LTL requirements, which in turn tend to relate input variables to output variables. These tests tend to be longer, as they must cause changes in the output to occur. In turn, they are more likely to allow faults to propagate to the outputs. While their construction is not foolproof—faults may still occur and not propagate to the output—the effectiveness of internal state is diminished in relation to tests satisfying structural coverage criteria.

We now have evidence that, as we have previously suspected, test suites designed to propagate changes to the output—like those satisfying these requirements coverage metrics—are less impacted by the presence of masking and are more likely to propagate faults to the outputs, resulting in less improvements when using the maximum oracle over the output-only oracle. Testers using requirements coverage metrics may therefore spend less resources on oracle selection, as the potential improvements are less than, for example, when using branch coverage.

6.6.4 Joint Influence of Random Test Suite Size and Oracle Data

As random test suite size increases, the potential influence of oracle size rapidly decreases for each case example. This is clearly illustrated in the drop in the relative improvement of the maximum oracle over the output-only oracle (Figure 6.7 shows a drop from over 20-150% to less than 10% for all case examples) and the drop in correlation of oracle size and fault finding (Figure 6.8 shows a drop of over 0.9 to 0.8-0.0 depending on case example and oracle subtype). Thus, as the power of the

test suite increases, the influence of the test oracle decreases (and vice-versa).

This highlights the same essential tradeoff in testing we previously highlighted when discussing test suites satisfying coverage criteria: we can achieve similar levels of fault finding in several ways. We further explore this tradeoff in Section 6.4.4, where we show that depending on the cost function for the testing process (e.g., the cost of running a test input, the cost of large oracles versus small oracles), we may select different combinations of test suites and oracles. In particular, our results indicate given the same cost function across different case examples, we may select very different combinations. This is illustrated in Figure 6.11, in which—for the cost function given—the *Latctl_Batch* and *DWM_1* systems favor relatively larger test suites with small oracles, while the opposite is true for the *Vertmax_Batch* system. (Interestingly, the *DWM_2* system is effectively indifferent to the selection of test suite size and test oracle—all combinations along our cost function line perform nearly identically.)

These results, along with those related to coverage criteria, demonstrate the potential effectiveness of *complete adequacy criteria*, defined in Chapter 3 as:

$$TO_C \subseteq P \times S \times 2^T \times O$$

Clearly, the interaction between test suites and oracles implies that we must consider one artifact when selecting the other. Indeed, we have already identified how the following factors influence the effectiveness of the test oracle: rigor of coverage criteria; requirements coverage versus structural coverage; large test suites versus small test suites; case example. Accordingly, it seems likely that we may be able to improve the effectiveness of the testing process by developing methods of selecting both the test inputs and the test oracle *simultaneously*. To the best of our knowledge, no such methods exist; we therefore believe this is a suitable task for future work.

Modeling Effectiveness of Testing Process

In addition to illustrating the joint influence of random test suite size and oracle data size, we also attempted to create a regression model for this relationship. Our analysis indicated that models of the form $FF = \beta_1 \cdot log(TS) + \beta_2 \cdot log(OS)$ provide the best model relative to other permutations of TS and OS, thus providing further evidence of the joint relationship between test suite size and oracle size and indicating the general logarithmic behavior informally noted previously is indeed present. However, the adjusted R^2 goodness of fit varies considerably across case examples, with a low of 0.28 (*Latctl_Batch*) and a high of 0.91 (*DWM_2*) for the unrestricted oracle subtype and a low of 0.29 (*Latctl_Batch*) and a high of 0.78 (*Vertmax_Batch*) for the output-base subtype. In general, the adjusted R^2 was high for the unconstrained subtype.

Given this consistency across case examples, we believe that while this model clearly requires refinement, our results are encouraging. This, along with results from Namin and Andrews [49], indicates that modeling the effectiveness of testing as factors warrants future study.

6.7 Chapter Conclusion and Future Work

In this chapter, we have explored how the selection of oracle data influence the effectiveness of the testing process. Our results indicate that within the domain of avionics system, oracle selection has a strong and practically significant effect on testing effectiveness. Selected results of interest include:

- Oracle size corresponds to fault finding ability with high, positive correlation.
- As the power of the test suite increases (either due to test suite size or the use of more rigorous coverage criteria), potential gains from improved test oracles diminish.

- In general, the effectiveness of the testing process is more strongly influenced by test oracle selection when using tests suites satisfying structural coverage criteria rather than requirements coverage criteria.

- Depending on the case example and cost function, it may be advisable to direct some finite testing resources towards improving the test oracle, rather than the test suite.

- The effectiveness of individual variables varies, with many variables finding very few faults and some variables—which we term critical-variables—finding 3-10 times more faults than the average variable.

These results indicate that while oracle selection is a key component in determining testing effectiveness, and that test oracles other than the industry standard output-only oracle may yield significant improvements in fault finding effectiveness. However, the variability between case examples, test coverage criteria, and test suite size indicates that general, cost-effective guidelines for oracle selection are elusive. Accordingly, we believe this work highlights the need for future work concerning methods of test oracle selection. We therefore propose the following challenges for future work:

- For a given test suite, how can we efficiently determine the most effective oracle for a given system? In particular, what methods can efficiently identify critical-variables, and can knowledge of the critical variables be used to generate small, effective oracles?

- For a given system, how can we efficiently determine the most effective combination of test suite and oracle for a given system?

- What are the limitations of oracle selection? Can we gain most, or all, of the benefits of the maximum oracle without incurring the overhead from monitoring such a large number of variables?
- What existing conclusions concerning the relative effectiveness of testing approaches are strongly dependent on the test oracle selected? In other words, how much of our current understanding of testing effectiveness is flawed due to implicit assumptions concerning the test oracle?

Chapter 7

Impact of Limited Observability on the Influence of Internal State on Testing Effectiveness

In this chapter, we explore how the structure of the program impacts the influence of the test oracle, specifically with respect to the influence of internal state information. This chapter is structured similarly to previous empirical chapters.

7.1 Motivation

Recall from Chapter 4 that the *observability* of a program is linked to our ability to monitor the program's behavior, i.e., our ability to determine the program's state. The observability of the program is a limiting factor in the construction of oracles—if we cannot observe some aspect of the program, this information cannot be incorporated into our oracle. Thus, for a set of programs $p, p', p'' \ldots$ which are semantically—but not observationally—equivalent, it is possible the set of potential oracles O we can use in testing varies.

Of concern in this chapter is how program structure impacts observability, and how this variation in observability impacts the influence of the test oracle. If we assume that the observability of the program is linked to the number of internal variables—as is often the case in practice, and as we assumed in Chapter 4—the structure of the program may impact its observability. Recall that when inlining a program, we remove internal state variables; consequently, inlining a program has a

negative impact on observability. This represents a potential problem for testers—depending on how the program is structured, the influence of the test oracle may be lesser or greater, thus complicating the issue of oracle selection. We therefore wish to better understand how program structure impacts the influence of oracles on testing effectiveness.

Note that while the potential for program structure to impact the influence of test oracles represents a potential problem, it also represents a potential opportunity. Clearly, restrictions on our ability to construct test oracles is an undesired limitation. However, this is a problem that can be overcome through engineering—if desired, we could develop tools, virtual machines (for languages such as Java) and/or compilers (for languages such as C) to allow us to observe computed values not assigned to internal state variables. Nevertheless, building such a tool is a non-trivial task, and there currently does not exist any evidence that such a tool would provide any potential benefits. By improving our understanding of how program structure and test oracles interact, we may provide such evidence.

7.2 Research Questions

We began by asking the following question:

Question 1 (Q1): How does the variation in observability, as caused by changes in program structure, impact the influence of internal state information in the oracle data?

For this question, we are only interested in the output-base oracle subtype, as changes here best showcase how change in the number of internal state variables impacts oracle effectiveness. As we will see, the reduction in observability that results from inlining a program has a strong negative impact on the influence of oracle

for all coverage criteria. However, we found in Chapter 5 that this inlining has a strong positive impact on the effectiveness of test suites satisfying structural coverage criteria. We therefore asked the following question:

Question 2 (Q2): With respect to structural coverage criteria and the maximum test oracle, is the reduction in the power of the maximum oracle (due to decreased observability) counterbalanced by the increased effectiveness of test suites satisfying structural coverage criteria? In other words, assuming we use a maximum oracle and a set of tests generated to satisfy a structural coverage criteria with a fixed program structure, which yields higher fault finding: the inlined or noninlined program?

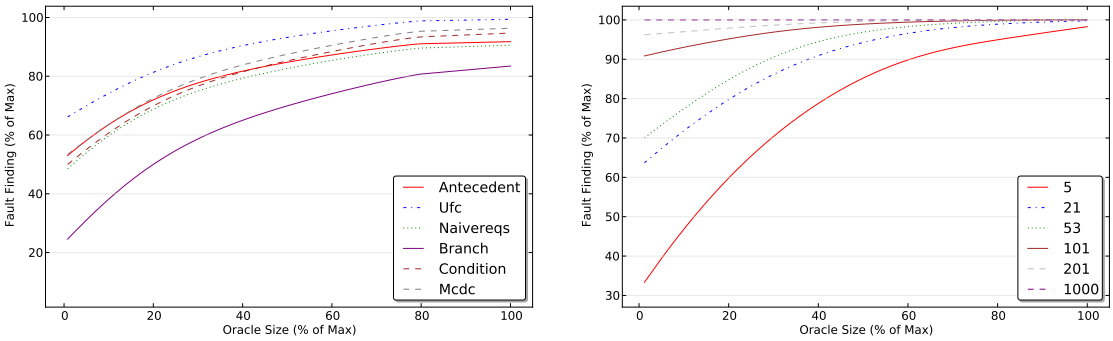
We explore the following factors in this chapter:

Program Structure: We examine programs that are *inlined* and *noninlined*.

Test Inputs: We examine test inputs derived to satisfy all coverage criteria, though per Question 2 we have specific interest in the *branch*, *condition*, and *MCDC* coverage criteria.

Recall that the test inputs generated to satisfy structural coverage criteria differ depending on program structure. Therefore, for several analyses, we will use only random test inputs, which remain the same across inlined/noninlined program structures.

Test Oracle: We explore the effectiveness of oracles from the output-base subtype, varying the size of the oracle data. Of particular interest, per Question 2, are the output-only and maximum oracles. Unrestricted oracles are not used.



(a) DWM_1 Unrestricted, All Coverage Criteria (b) DWM_2 Output-base, Random Test Suites

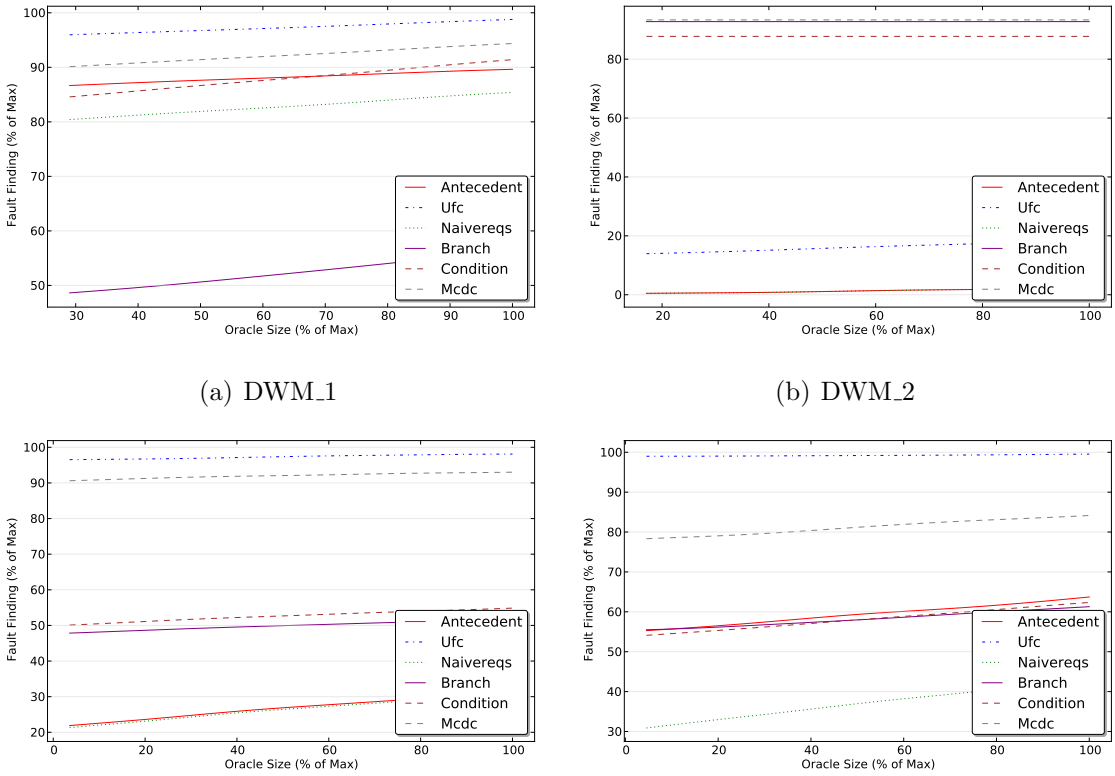
Figure 7.1: Oracle Size vs Fault Finding Examples for Noninlined Programs

7.3 Impact of Program Structure on Influence of Internal State Information

To answer Question 1, we could simply repeat our all the analyses performed in Chapter 6 over the inlined program structure. However, it becomes quickly apparent that the program structure has a strong, negative impact on the influence of oracle data; exhaustive repetition is unnecessary to demonstrate this. We therefore focus on (1) how relationship between oracle size and fault finding changes when using an inlined program, (2) the reduction in relative improvement of using a maximum oracle over an output-only, and (3) how the relationship between random test suite size, oracle size, and fault finding changes.

7.3.1 Influence of Oracle Size Given Fixed Test Data

In Chapter 6, we saw the relationship between oracle size and fault finding can be characterized as logarithmic. This relationship generally holds for both oracle subtypes and across methods of test input, as shown in Figure 7.1. We would like to determine if this relationship still holds for inlined programs when using output-base



(a) DWM_1

(b) DWM_2

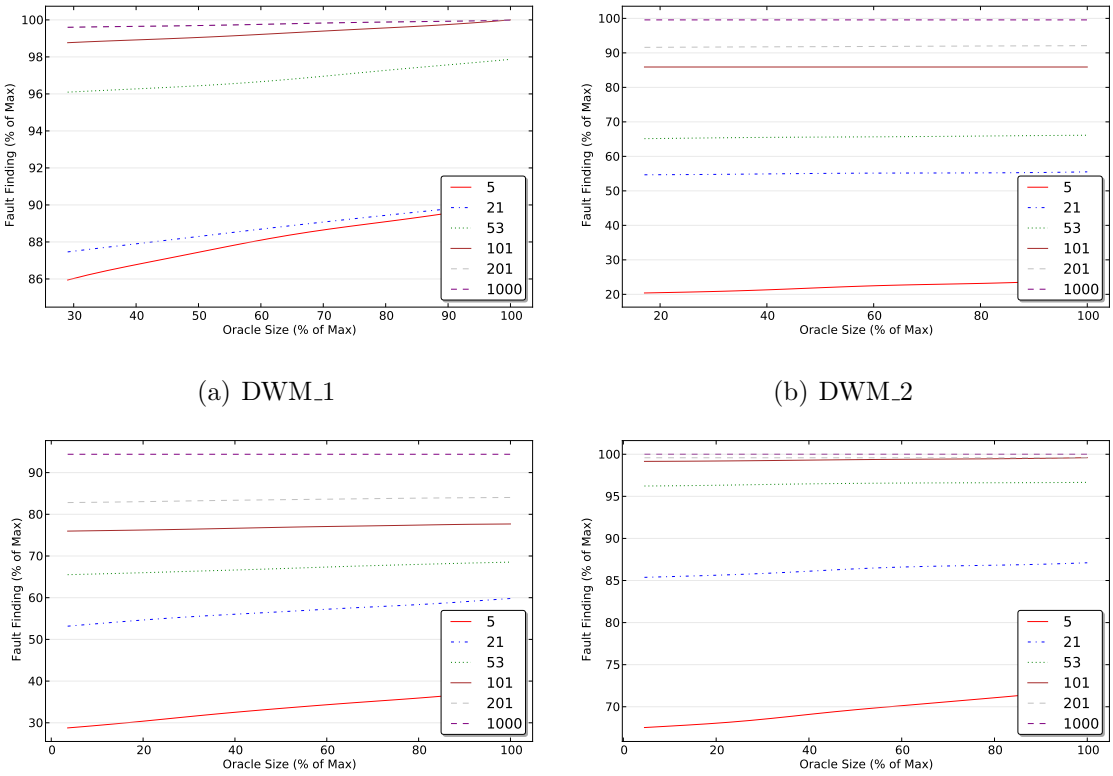
(c) Vertmax_Batch

(d) Latctl_Batch

Figure 7.2: Oracle Size vs Fault Finding, Output-base oracles, Test Suites Satisfying Coverage Criteria

oracles. We therefore visualized this relationship, as shown in Figure 7.2 and 7.3 for test suites satisfying coverage criteria and randomly generated test suites, respectively.

As we can see, the relationship between oracle size and fault finding has changed considerably; for each case example, coverage criterion, and set of randomly generated tests, the relationship now appears to be largely linear. This indicates that, unlike when testing the noninlined programs, the set of faults detected by each internal state variable does not overlap—each variable detects different faults.



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 7.3: Oracle Size vs Fault Finding, Output-base oracles, Randomly Generated Test Suites

Upon examination of the inlined systems, we saw that internal state variables rarely reference one another. Therefore, errors in the computation of an expression can often only be detected by observing either an output or the internal state variable the expression result is assigned to. This likely accounts for linear nature of the relationship between fault finding and oracle size—outputs are always observed in output-base oracles, and as each internal state variable detects different faults, and adding an internal state variable to an existing output-base oracle increases the effectiveness of fault finding, irrespective of the number of internal state variables already

DWM_1 System			
	Naive Req	**Antecedent**	**UFC**
Noninlined % FF Imp	10.22%	6.74%	2.33%
Inlined % FF Imp	5.83%	3.38%	2.93%
DWM_2 System			
Noninlined % FF Imp	∞	∞	320.66%
Inlined % FF Imp	400.0%	400.0%	33.05%
Latctl_Batch System			
Noninlined % FF Imp	135.26%	53.45%	1.28%
Inlined % FF Imp	39.61%	14.92%	0.55%
Vertmax_Batch System			
Noninlined % FF Imp	120.07%	117.41%	1.96%
Inlined % FF Imp	44.80%	42.95%	1.54%

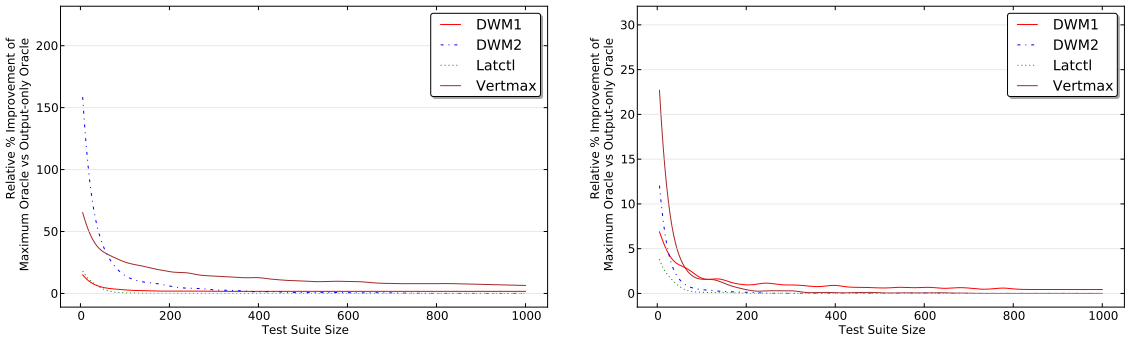
Table 7.1: Fault Finding Effectiveness, Output-Only vs Maximum Oracle, Noninlined Versus Inlined Programs

FF = Fault Finding

present.

7.3.2 Relative Improvement using Maximum Oracle

In addition to the shift towards a linear relationship, we can also see that in several instances the relative improvement between the output-only and maximum oracle is smaller than previously seen. We quantify these differences for both test suites satisfying requirements coverage criteria and randomly generated test suites. In Table 7.1, we list the relative improvement when using the maximum oracle over the output-base oracle for test suites satisfying requirements coverage. In Figure 7.4 we plot the relative improvement versus test suite size for randomly generated tests, for both the inlined and noninlined programs. (Recall that unlike randomly generated test suites or test suites satisfying requirements coverage, test suites satisfying structural coverage criteria are generated and run separately for inlined and noninlined programs. A direct comparison of these tests suites is therefore unfair.)



(a) Noninlined Programs

(b) Inlined Programs

Figure 7.4: Relative Improvement of Maximum Oracle over Output-only Oracle, Randomly Generated Test Suites

As we can see, the relative improvement when using both randomly generated test suites and test suites generated to satisfy requirements coverage criteria is often considerably less when using the inlined systems as compared to the noninlined systems. In the case of requirements coverage metrics, the relative improvement is lower across case examples, with the relative improvement for inlined systems being often less than half that of the improvements observed for noninlined systems. The exception to this occurs for UFC coverage, in which the already very low relative improvements leave little room for further reductions.

In the case of the randomly generated tests, we see roughly the same pattern is present when considering how test suite size influences the potential relative improvement. However, the relative improvements for the inlined systems are considerably smaller than those for the noninlined systems, beginning at just over 20% and dropping quickly to below 5% by 35 tests, as compared to 150% to 75% for the noninlined systems.

7.3.3 Relationship of Random Test Suite Size, Oracle Size, and Fault Finding

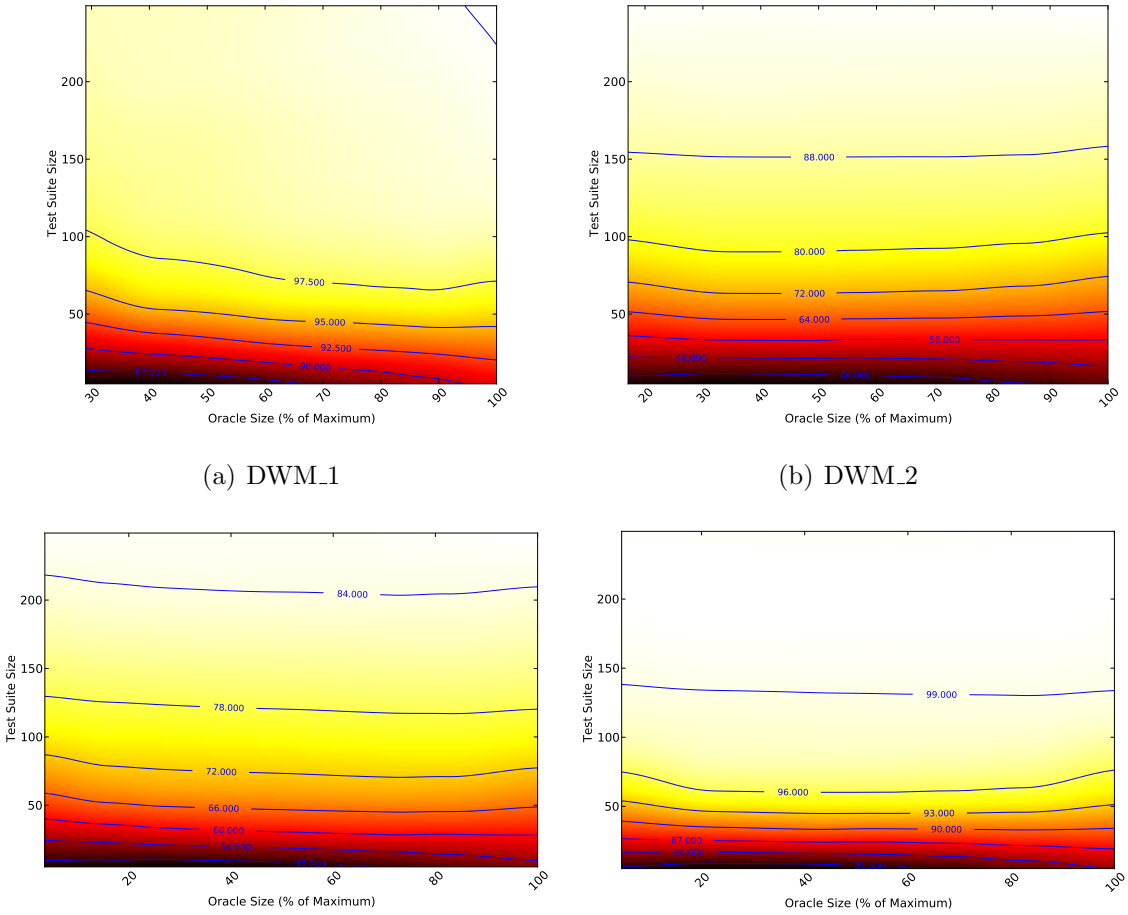
In Section 6.4.4, we explored the relationship between test suite size, oracle size, and fault finding effectiveness. These results illustrated that a potential tradeoff exists—we can achieve the same level of fault finding using different approaches, e.g., by using a large test suite with a small oracle, or a small test suite with a large oracle. Thus, depending on the case example and the cost function, we may wish to use different approaches when testing different systems.

In Figure 7.5, we plot contour maps illustrating the fault finding effectiveness of variously sized random test suites and test oracles for the inlined systems, using oracles of the output-base subtype. As we can see, the potential tradeoffs observed for noninlined systems are very slight, if present at all. For each case example, the contour maps consist of nearly horizontal lines, implying that in general, it is easier to improve fault finding by increasing the size of our test suite.

7.4 Power of Inlined Structural Coverage Criteria Versus Power of Noninlined Maximum Oracle

Based on the results to Question 1, we can see that the reduced observability due to program structure has a strong negative impact on the influence of internal state, at least with respect to randomly generated tests and tests satisfying requirements coverage metrics. These results lead to an interesting dichotomy: inlining a program has a positive impact on the effectiveness of structural coverage criteria, and a negative impact on the effectiveness of the maximum oracle. This led to the following question:

Question 2 (Q2): Assuming we use a maximum oracle and a set of tests generated



(a) DWM_1

(b) DWM_2

(c) Vertmax_Batch

(d) Latctl_Batch

Figure 7.5: Relationship of Test Suite Size, Oracle Size and Fault Finding, Output-base Subtype

to satisfy a structural coverage criterion with a fixed program structure, which yields higher fault finding: the inlined or noninlined program?

To answer this question, we compare the fault finding of test suites satisfying each structural coverage across the inlined and noninlined programs using the maximum oracle. These results are listed for each case example in Table 7.2.

DWM_1 System			
	Branch	**Condition**	**MCDC**
Noninlined MX % FF	83.17%	94.52%	96.17%
Inlined MX % FF	56.19%	91.16%	94.20%
DWM_2 System			
Noninlined MX % FF	82.65%	93.09%	88.99%
Inlined MX % FF	92.74%	87.73%	93.33%
Latctl_Batch System			
Noninlined MX % FF	89.30%	93.60%	96.70%
Inlined MX % FF	61.25%	62.17%	84.16%
Vertmax_Batch System			
Noninlined MX % FF	85.16%	100.0%	100.0%
Inlined MX % FF	51.6%	54.79%	93.08%

Table 7.2: Fault Finding Effectiveness Using Maximum Oracle, Inlined vs Noninlined Program Structure

MX = Maximum, FF = Fault Finding

As shown, the results lean heavily towards favoring the noninlined program. For three of the four case examples (*Latctl_Batch*, *DWM_1* and *Vertmax_Batch*), the artifacts generated for the noninlined program outperform the artifacts generated for the inlined program for every structural coverage criterion. For one case example, the *DWM_2* system, the results are mixed—for branch and MCDC coverage, the artifacts generated from the inlined program outperform those generated from the noninlined program, while for condition coverage the opposite is true. Recall, however, that the results in Chapter 6 indicated that tests generated to satisfy structural coverage criteria performed exceptionally poorly over the noninlined version *DWM_2* relative to tests generated using the inlined version, with improvements of 336.18% to 556.70% observed when moving from the noninlined to the inlined program structure.

The results therefore indicate that, if required to choose, the noninlined program structure generally yields better results when using the maximum oracle paired with test suites generated to satisfy a structural coverage criteria (barring exceptional im-

provements in test suite effectiveness when moving from the noninlined to inlined system). In short, the improved effectiveness resulting from better observability outweighs the improved effectiveness resulting from better test suites.

7.5 Discussion

Our results clearly indicate that variation in observability, as caused by changes in program structure, has a strong, negative impact on the potential effectiveness of internal state. This is evidenced by the consistent reduction in relative improvement when testing the inlined systems versus the noninlined systems. This reduction was observed across coverage criteria and case examples, with relative improvements for inlined systems often dropping to half that (or less) of the corresponding improvements for the noninlined systems. Exceptions to this pattern of reduction occur only for case examples and coverage criteria pairings in which the relative improvement was already very low ($< 3\%$).

Unsurprisingly, we see in Figure 7.5 that as a result, the relationship between test suite size and oracle size has changed substantially. Namely, the potential tradeoffs highlighted in Chapter 6—for example, the ability to achieve similar levels of fault finding when selecting a strong test suite with a weak oracle, or a weak test suite with a strong oracle—is only marginally present when testing the inlined systems. This implies that when testing systems with low observability, the primary method of improving testing performance, given some test suite and an output-base oracle, will be to improve the test suite (e.g., use a stronger coverage criterion or a larger test suite). Only in scenarios in which using a large test oracle is free or nearly free would considering internal state be justifiable.

The implications of these results are twofold. First, we can see the results from Chapter 6 are not completely generalizable—it is possible to construct a system such

that our observations concerning the impact of oracle selection, and the relationship between test input and oracle selection, do not generalize. In the case of reduced observability, the impact of considering internal state in oracle data matters considerably less; potentially, in the case of increased observability, the impact may be considerably more. This highlights the lack of strong general guidelines available for selecting oracle data, and reinforces the need (per Chapter 6 conclusions) of developing methods of determining, for an arbitrary case example, what combination of test inputs and oracle data is the most cost-effective.

Second, as we discussed when motivating this chapter, a lack of observability is a problem that can be solved through engineering. However, to the best of our knowledge, tools specifically designed to improve observability during testing do not exist, and constructing such tools is a non-trivial task. By demonstrating how the lack of observability can negatively impact the influence of test oracles, we have provided motivation for the construction of such tools. If we could easily and cheaply use all computed values as part of the oracle data—not just internal state and output variables—the effect of program structure on observability could be mitigated, and the effectiveness of testing could potentially be improved as a result.

The results for Question 2 do not necessarily have any direct implications for testing practice. However, we feel the results—highlighting how the factors of test suite selection via structural coverage criteria, program structure, and oracle selection all interact—make a fitting ending to this dissertation, and answer a question that that long gone unanswered in the CriSys lab at the University of Minnesota.

7.6 Chapter Conclusion and Future Work

In this chapter, we have explored how program structure, and its influence on observability, impacts the influence of test oracle selection on the effectiveness on the testing

process. Our results indicate that the influence of test oracle selection is strongly impacted by program structure; in particular, the ability of internal state to improve the effectiveness of output-base oracles is reduced when testing an inlined program (with many internal variables inlined) as compared to when testing a noninlined program. These results raise several questions for future work, including:

- Can we objectively determine if a program is sufficiently testable? In other words, can we determine if a program's structure allows for the construction of effective test oracles?

- How can we design programs to maximize the testability, while avoiding unnecessary costs? In other words, how can we structure programs such that the maximum oracle will be very effective without assigning every computed value to an internal state variable?

Chapter 8

Conclusions and Final Thoughts

In this dissertation, we have explored the influence of multiple artifacts on the effectiveness of software testing. We began by presenting two formalisms for discussing testing: a functional formalism by Gourlay, extended to better represent testing in practice, and a computational formalism based on Kripke structures. We then used these formalisms to define several properties of interest in testing, and explored the implications of our extended formalism on previous theoretical work. We also performed a large scale empirical study using four real-world avionics systems, and explored how program structure, test oracle selection, and test input selection interact to influence the effectiveness of software testing. We have concluded each chapter

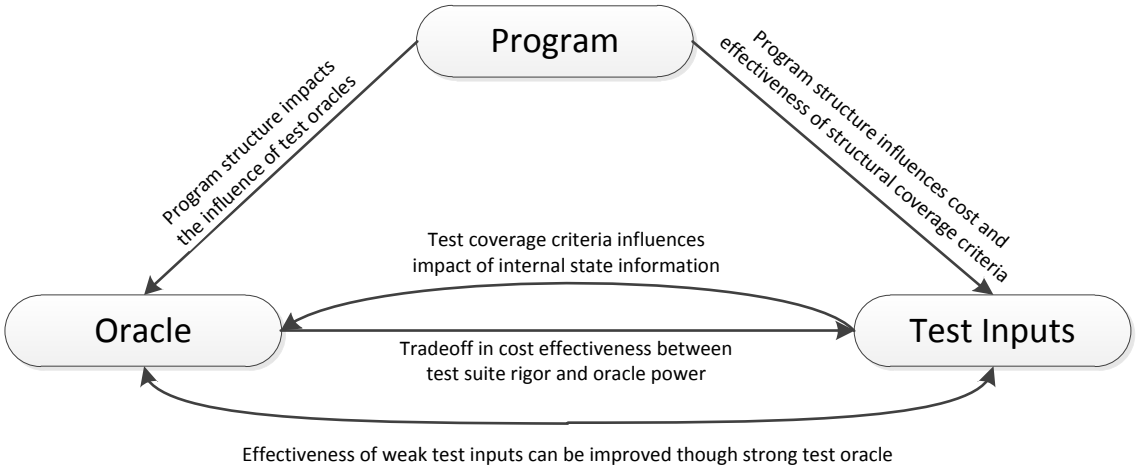


Figure 8.1: Observed Relationships Between Testing Artifacts

with a short discussion of the implications of our findings, along with questions for future work. Rather than repeat these discussions, we conclude this dissertation with a short overview of its larger implications.

In the context of our long-term objective, we believe this dissertation represents an important step in improving the effectiveness of testing through better understanding of how testing artifacts interact. We have highlighted, both in both theory and practice, how testing artifacts can interact in significant ways to influence the effectiveness of software testing. Broad example interactions observed in this dissertation are shown in Figure 8.1.

These results highlight both *opportunity* and *danger* in testing research. With respect to *opportunity*, we noted in the introduction that existing research in the testing community has largely ignored how artifacts other than testing inputs impact the effectiveness of testing. This work, by quantifying the potential benefits of considering other artifacts—notably, test oracles—provides evidence that this focus is misplaced, and that exploring other avenues of testing research (or at least exploring the same avenues while considering other artifacts) may lead to improvements in testing effectiveness. We highlight questions raised in this dissertation in Figure 8.2.

The opportunities found in this work are perhaps best exemplified by Chapter 6, in which we found there exists a tradeoffs when selecting test oracles and test inputs. Previous work in software testing has focused almost exclusively on how test inputs alone, or (less commonly) oracles alone can impact testing cost and effectiveness. By exploring both artifacts together, we have demonstrated that, in some scenarios, we may be able to do better job selecting test inputs and test oracles. Such insights point towards new directions in testing research that we have recently begun to explore.

With respect to *danger*, this work demonstrates how factors apart from test inputs can impact testing effectiveness, and thus aptly illustrates how an uncontrolled or

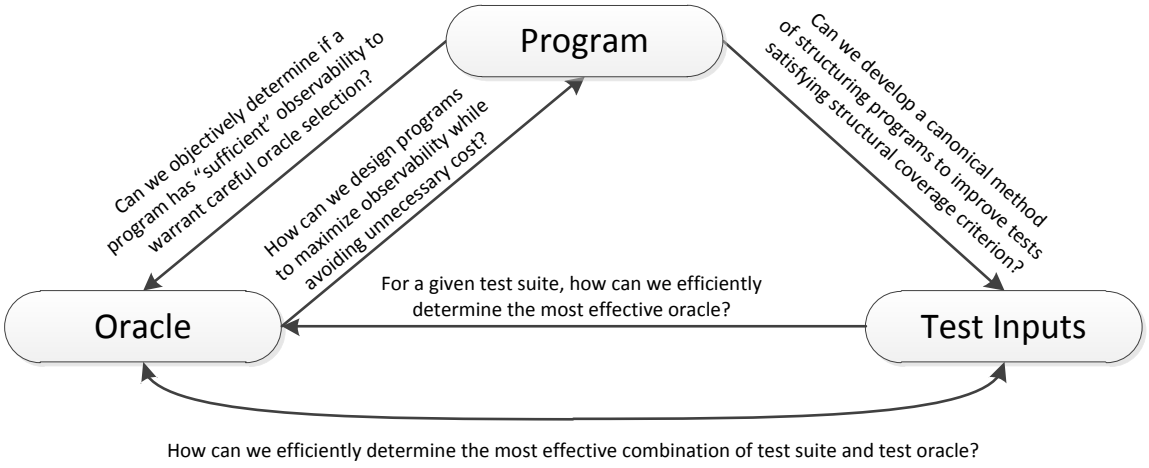


Figure 8.2: Questions Raised Concerning Relationships Between Testing Artifacts

undiscussed factor—for example, the program structure, or the test oracle used—may impact the conclusions reached in an empirical study. Accordingly, we believe these results provide strong motivation to consider how such artifacts may influence results in an empirical study, and we hope to see this acknowledged by other testing researchers in the design of their studies.

The potential for danger in testing studies is illustrated by an example developed across this dissertation. In Chapter 5, we explored how program structure influenced the effectiveness of structural coverage criteria, finding that this factor has a strong influence on the effectiveness structural coverage criteria. These results indicate that we can potentially improve testing effectiveness when using structural coverage criteria by restructuring our program. Coupled with results from Chapter 6, in which we find that a maximum test oracle outperforms the output-only oracle, we can make two recommendations for maximizing testing effectiveness: use the inlined program when using structural coverage criteria, and use the maximum oracle.

However, as we found in Chapter 7, these recommendations are, in fact, incom-

patible. When using an inlined program structure, we greatly reduced the power of the maximum oracle—the structural test inputs improve, but the maximum oracle becomes weaker. Therefore, when using the maximum oracle and a test suite satisfying a structural coverage criteria, if forced to select a single program structure, we are best served by selecting the noninlined program. This drives home the importance of controlling for factors, and the danger of failing to do so.

Moving forward, we hope that this work, by highlighting the interaction between testing artifacts, will encourage others in testing research to adopt a more holistic view of the testing process, and that this view will lead to improvements in the effectiveness of the testing process.

Bibliography

[1] N. Amla and P. Ammann. Using z specifications in category partition testing. In *Computer Assurance, 1992. COMPASS'92., Proc. of the 7th Annual Conf. on*, pages 3–10, 1992.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pages 402–411, 2005.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] Luciano Baresi and Michal Young. Test oracles. In *Technical Report CIS-TR-01-02, Dept. of Computer and Information Science, Univ. of Oregon*.

[5] Boris Beizer. *Software Testing Techniques, 2nd Edition*. Van Nostrand Reinhold, New York, 1990.

[6] G. Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT'91: Colloquium on Trees in Algebra and Programming (CAAP'91)*, page 99. Springer, 1991.

[7] G. Bernot, MC Gaudel, B. Marre, and U.R.A.C. Liens. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[8] A. Bertolino. Software testing research: Achievements, challenges, dreams. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[9] LC Briand. A critical analysis of empirical research in software testing. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First Int'l Symposium on*, pages 1–8, 2007.

[10] L.C Briand, M. DiPenta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. on Software Engineering*, 30 (11), 2004.

[11] T.A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[12] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symp. Principles Program. Lang. (POPL)*, pages 178 – 188, Munich, Germany, 1987.

[13] T.Y. Chen and Y.T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, pages 977–980, 1994.

[14] T.Y. Chen and Y.T. Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, 1996.

[15] W. Chen, R.H. Untch, G. Rothermel, S. Elbaum, and J. Von Ronne. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Software Testing Verification and Reliability*, 12(4):197–218, 2002.

[16] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.

[17] John Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.

[18] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[19] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.

[20] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 18(2):159–182, 2002.

[21] P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. In *IEEE Trans. on Software Engineering*, 1993.

[22] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Prc. of the 19th Int'l Symp. on Software Testing and Analysis*, pages 147–158. ACM, 2010.

[23] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[24] M.C. Gaudel. Testing can be formal, too. *Lecture Notes in Computer Science*, 915:82–96, 1995.

[25] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. *PLDI05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[26] J. B. Goodenough and S. L. Gerhart. Toward a theory of testing: Data selection criteria. In R. T. Yeh, editor, *Current trends in programming methodology*. Prentice Hall, 1979.

[27] J.S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. on Software Engineering*, pages 686–709, 1983.

[28] J.P. Guilford and B. Fruchter. *Fundamental statistics in psychology and education*. McGraw-Hill, New York, 1956.

[29] W.J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.

[30] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[31] D. Hamlet. Foundations of software testing: dependability theory. *ACM SIGSOFT Software Engineering Notes*, 19(5):128–139, 1994.

[32] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *Software Testing, Verification, and Analysis, 1988., Proc of the 2nd Workshop on*, pages 206–215, 1988.

[33] M.J. Harrold. Testing: a roadmap. In *Proc. of the Conf. on the Future of Software Engineering*, pages 61–72. ACM New York, NY, USA, 2000.

[34] K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[35] Mats P.E. Heimdahl and George Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[36] R.M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(4):448, 2002.

[37] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3), 1976.

[38] WE Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. on Software Engineering*, pages 371–379, 1982.

[39] W.E. Howden. Theory and practice of functional testing. *IEEE Software*, 2(5):6–17, 1985.

[40] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

[41] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Emgineering*, 29(3):195–209, March 2003.

[42] N. Juristo, A.M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1):7–44, 2004.

[43] P.H. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley-Interscience, 2007.

[44] O. Grumberg M. C. Browne, E. M. Clarke. Characterizing finite kripke structures in porpositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[45] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[46] J. Mayer and R. Guderlei. Test oracles using statistical methods. In *Proc. of the First Int'l Workshop on Software Quality*, pages 179–189. Citeseer, 2004.

[47] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? *Automated Software Engineering, 2003. Proc. 18th IEEE Int'l Conf. on*, pages 164–173, 2003.

[48] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[49] A.S. Namin and J.H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the 18th Int'l Symp. on Software Testing and Analysis*, pages 57–68. ACM, 2009.

[50] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[51] A. Parrish and S.H. Zweben. Analysis and refinement of software test data adequacy properties. *IEEE Trans. on Software Engineering*, 17(6):565–581, 1991.

[52] A.S. Parrish and S.H. Zweben. Clarifying some fundamental concepts in software testing. *IEEE Trans. on Software Engineering*, 19(7):742–746, 1993.

[53] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[54] A. Rajan, M. Whalen, M. Staats, and M.P. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.

[55] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[56] Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[57] D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proc. of the 14th Int'l Conference on Software Engineering*, pages 105–118. Springer, May 1992.

[58] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

[59] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

[60] M. Staats, M.W. Whalen, and M.P.E. Heimdahl. New ideas and emerging results track: Better testing through oracle selection. In *Proc. of NIER Workshop, Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.

[61] M. Staats, M.W. Whalen, and M.P.E. Heimdahl. Programs, testing, and oracles: The foundations of testing revisited. In *Proc. of Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.

[62] Matt Staats, Michael W. Whalen, Ajitha Rajan, and Mats P.E. Heimdahl. Coverage metrics for requirements-based testing: Evaluation of effectiveness. In *Proceedings of the Second NASA Formal Methods Symposium*. NASA, April 2010.

[63] CAJ Van Eijk. Sequential equivalence checking based on structural similarities. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 19(7):814–819, 2002.

[64] J. Voas. Dynamic testing complexity metric. *Software Quality Journal*, 1(2):101–114, 1992.

[65] J.M. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, 1992.

[66] J.M. Voas and K.W. Miller. The revealing power of a test case. *Software Testing, Verification and Reliability*, 2(1):25–42, 1992.

[67] J.M. Voas and K.W. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994., 5th Int'l Symposium on*, pages 152–157, 1994.

[68] SN Weiss. Comparing test data adequacy criteria. *ACM SIGSOFT Software Engineering Notes*, 14(6):42–49, 1989.

[69] E.J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465, 1982.

[70] E.J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. on Software Engineering*, 12(12):1128–1138, 1986.

[71] E.J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[72] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. on Software Engineering*, 17(7):703–711, 1991.

[73] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. on Software Engineering*, pages 236–246, 1980.

[74] E.J. Weyuker, S.N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proc. of the Symposium on Testing, Analysis, and Verification*, page 10. ACM, 1991.

[75] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 25–36. ACM, July 2006.

[76] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proc. of the 17th Int'l Conf. on Software Engineering*, pages 41–50. ACM, 1995.

[77] M.R. Woodward and Z.A. Al-Khanjari. Testability, fault size and the domain-to-range ratio: An eternal triangle. In *Proc. of the 2000 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 168–172. ACM, 2000.

[78] MR Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proc. of the 2nd Workshop on*, pages 152–158, 1988.

[79] Q. Xie and A.M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 16(1):4, 2007.

[80] H. Zhu. Axiomatic assessment of control flow-based software test adequacy criteria. *Software Engineering Journal*, 10(5):194–204, 1995.

[81] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Trans. on Software Engineering*, 22(4):248–255, 1996.

[82] H. Zhu and P.A.V. Hall. Test data adequacy measurement. *Software Engineering Journal*, 8(1):21–29, 1993.

[83] H. Zhu, P.A.V. Hall, and J.H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[84] H. Zhu and X. He. A theory of behaviour observation in software testing. Technical report, 1999.

[85] H. Zhu and X. He. Constructions of behaviour observation schemes in software testing. In *Proc. of Fifth IEEE Symposium on HASE*, pages 15–17, 2000.

[86] H. Zhu and X. He. A theory of testing high level petri nets. In *Proc. of the Int'l. Conf. on Software: Theory and Practice, 16th IFIP World Computer Congress*, pages 443–450. Citeseer, 2000.