# Programs, Tests, and Oracles: The Foundations of Testing Revisited*

Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl
Department of Computer Science and Eng.
University of Minnesota
[staats,whalen,heimdahl]@cs.umn.edu

## ABSTRACT

In previous decades, researchers have explored the formal foundations of program testing. By exploring the foundations of testing largely separate from any specific method of testing, these researchers provided a general discussion of the testing process, including the goals, the underlying problems, and the limitations of testing. Unfortunately, a common, rigorous foundation has not been widely adopted in empirical software testing research, making it difficult to generalize and compare empirical research.

We continue this foundational work, providing a framework intended to serve as a guide for future discussions and empirical studies concerning software testing. Specifically, we extend Gourlay's functional description of testing with the notion of a test oracle, an aspect of testing largely overlooked in previous foundational work and only lightly explored in general. We argue additional work exploring the interrelationship between programs, tests, and oracles should be performed, and use our extension to clarify concepts presented in previous work, present new concepts related to test oracles, and demonstrate that oracle selection must be considered when discussing the efficacy of a testing process.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Theory

## Keywords

Theory of testing, testing formalism

*This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and NSF grants CCF-0916583 and CNS-0931931.


*ICSE*'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA


## 1. INTRODUCTION

For several decades, testing has been an important research topic and a standard part of software development practice. In the 1980s and early 1990s, several authors explored the theory of testing in an attempt to put testing research on a solid formal foundation and provide coherent frameworks for discussion. Notable contributions include explorations of: the problem of test set selection [16, 17]; the problem of constructing test data adequacy criteria [26, 38, 44]; the need to compare test data adequacy criteria [12, 17, 41]; the use of input partitioning when constructing test data adequacy criteria [12, 40]; and the use of test hypotheses when selecting test sets [4, 5, 14]. This body of work largely discusses the foundation of testing—the goals, underlying problems, and limitations of testing—separate from any specific method of testing.

While these early contributions are valuable and helped shape the direction of testing research as well as 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 generally poorly evaluated. This lack of a formal foundation and rigorous evaluation of proposed new approaches has been a persistent problem in the research community [6].

Consider as examples the generally well conducted and highly influential studies of Rothermel et al. [31], and Wong et al. [42]. 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. This in turn makes it difficult to interpret 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 synthesizing the empirical results from several independent investigations.

In this work, we attempt to remedy this situation and provide a common framework for empirical testing research by revisiting work on the formal foundations of testing, highlighting and clarifying issues with the existing work. 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 such assumptions lead to research results that may be misleading or may not be generally applicable in real testing projects. In addition, as mentioned above, such implicit assumptions makes comparisons of research efforts difficult.

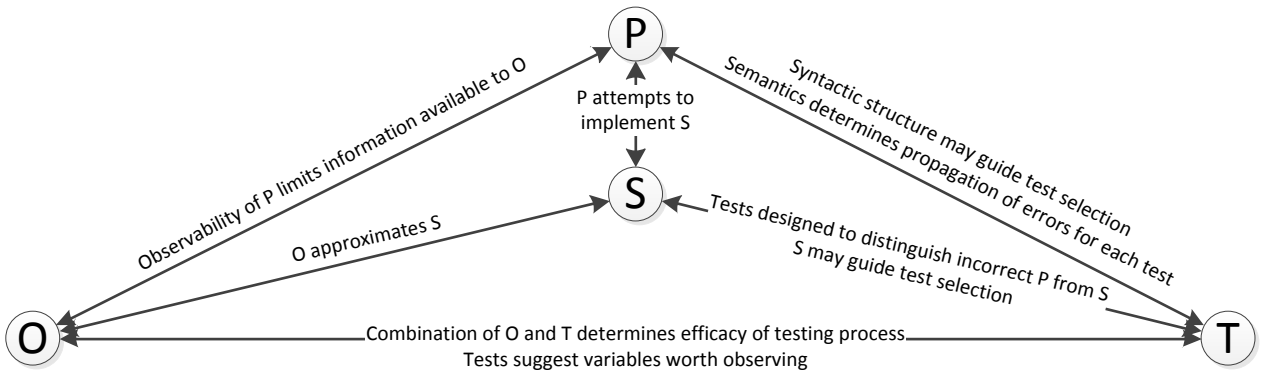


**Figure 1: Example relationships between testing factors.**

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. 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. In particular, the *interrelationship* between the programs under test, the test set selected, and the oracle used is not well understood. For example, the effects of program structure on the efficacy of structural coverage criteria and the effect of oracle selection on fault finding have not been adequately explored in current literature.

The implications of this extend to all testing processes. We have in our empirical work observed how failing to consider all factors in testing can impact the efficacy of the testing process, specifically: how the the structure of the program under test affects the efficacy of the MC/DC structural coverage criterion [28], and how the percentage of program state considered by an oracle and the size of a randomly generated test set jointly influence fault finding ability [32].

In this paper, we begin to address these issues by first providing a formal foundation of testing acknowledging the role of *test oracles* in the testing process. We then use this framework to explore the interrelationship between the artifacts involved in software testing. In particular, we consider how one common implicit assumption—the existence of a test oracle—affects the results of previous work on the theory of testing. We find this implicit assumption is key to existing results, demonstrating, for example, that comparisons between test coverage criteria must be made with respect to a constant test oracle; we clarify existing work, demonstrating, for example, that the spectrum of mutation testing techniques in fact represents one technique defined in terms of the adequacy of both a test set and an oracle; and finally we discuss new concepts, defining, for example, desirable properties of test oracles such as *sound* and *complete*. Based on our formalization and observations, we identify a collection of research challenges we see as important to further the frontiers in software testing.

## 2. A HOLISTIC VIEW OF TESTING

Early work on the testing process focused on test selection, leading to the definition of test data adequacy criteria by Goodenough and Gerhart [16] as well as others [17, 40]. Subsequent work in testing has reinforced this test-selection-centric view of testing, with one author even observing that the problem of testing is to

> *"find a graph and cover it"* [3].

Unsurprisingly, a large quantity of work explores issues such as what test coverage criteria to use and how to generate tests satisfying those criteria.

Nevertheless, some authors have acknowledged and explored the influence of other artifacts on testing research. In Figure 1, we illustrate the interrelationships between 4 testing artifacts commonly discussed in the literature: specifications, programs, tests, and oracles. In this context, specifications $S$ represent the abstract, perfect notion of correctness; the only arbiter of correctness representing the true requirements of the software—not the possibly flawed requirements as captured in a document or formal specification.

Some of these relationships are straightforward and intuitive. For example, the relationship between the specification and the other artifacts is obvious—the program is derived from, and intended to implement, the specification (for instance, through requirements capture, architecture, design, and coding, or through the creation of a formal model and code generation), tests are in a similar way derived from the specification and are intended to demonstrate executions where the program violates the the specification, and, similarly, the oracle is based on the specification and is intended to determine, for each test, if the program has violated the specification. The interrelationships between programs, tests, and oracles are less obvious, however, and warrant further discussion.

It is easy to see that all three artifacts are intertwined in the testing process. For example, consider a stateful embedded program with a sequence of simple decisions[1]. In testing such a program, it is likely that longer test cases will be beneficial, allowing corrupted state to propagate to outputs. It is also likely that a more powerful oracle considering internal state information will be beneficial since it may be able to detect corrupted internal states early. In using longer test cases, however, we reduce the benefits of using an oracle observing internal state, as corrupted state information is more likely propagate to the output. Thus, the characteristics of the program suggest two methods of improving fault finding, but the use of one method diminishes the value of the other. Such issues involve the largely unexplored interrelationship between programs, tests, and oracles.

While programs, tests and oracles are intertwined, to manage the complexity of testing research we tend to think of the artifacts in terms of pairs. Most testing research has been concerned with the relationship between programs $P$ and tests $T$. Examples include the development of structural coverage criteria, where the adequacy of a test suite is based on how well it covers syntactic aspects of the program structure [46] and the creation of automated techniques to generate tests providing the desired coverage [29, 13] . Work

[1]A decision in this context is any boolean expression containing a boolean operator.

relating other aspects of programs and tests is less common, and includes work by Rajan et al. on how the structure of the program affects the efficacy of a test suite providing structural code coverage [28], and work by Voas et al. and others demonstrating how the likelihood of errors propagating to the output can be used to guide test selection [1, 10, 33].

In comparison, the interrelationships between oracles $O$ and both the programs $P$ or the tests $T$ have received relatively little attention. Concerning the interrelationship between the programs $P$ and oracles $O$, Voas et al. explored how testability information can be used to identify program locations where faults can hide, thus indicating locations where assertions may be placed to improve the quality of the oracle [35]. Concerning the interrelationship between tests $T$ and oracles $O$, there have been a handful of studies investigating how the oracle and test set used jointly affect fault finding, including work by Memon et al. in the domain of GUI testing [24], work by Briand et al. comparing state-based invariants and input/expected value oracles [8], and our own work exploring the joint affect of oracle data and test suite size [32].

The practical implications of this holistic view of testing are threefold. First, we must develop new theories, techniques, and tools for finding effective combinations of program characteristics, tests, and oracles. In the next section, we propose an extension to Gourlay's testing framework aimed at addressing the lack of testing theory related to oracles. We provide a formal treatment of oracles, oracle selection, and oracle properties similar to that provided for tests and test selection. Note that this formal framework is intended to provide a conceptual framework that can be used as a foundation for future work in empirical software testing research, not as a framework intended to prove obscure (and often unrealistic) properties of the testing process.

Second, empirical research should more accurately reflect the holistic nature of testing. At a minimum, testing research should make explicit what is changed (e.g., the coverage criterion) and what is kept constant (e.g., the structure of the program and the oracle). Authors should then clearly state why their experimental parameters were chosen and argue why they are reasonable. Unfortunately, this has not generally been the case in the literature, leaving the reader to guess what types of programs and oracles the research generalizes to. Other researchers have also expressed concerns regarding these issues, for example, Briand touches upon some of these issues in his discussion of validity issues in empirical testing research [7].

Finally, we must study the interrelationships between artifacts in greater detail; in particular, we believe greater study on the effect of test oracles is warranted. Example questions concerning the relationship between oracles and programs include: *"how do we create suitable oracles for stateful programs where errors may take significant time to propagate to outputs?"*. Example questions concerning the relationship between oracles and tests include: *"given test coverage criteria sensitive to program structure, which internal variables should the oracle observe to improve fault finding ability [28]?"*, and more generally, *"given finite testing resources, for this a program $P$ what combination of tests $T$ and oracle $O$ should I use to achieve high levels of fault finding?"*—maybe improving the test oracle by inserting more assertions in the code may be a more cost effective solution than producing more tests.

# 3. FUNCTIONAL MODEL OF TESTING

Beginning with Goodenough and Gerhart's seminal work [16], 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 [17], 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 $(P, S, T, corr, ok)$ where:

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

As in our discussion in the previous section, each specification $s \in S$ represents an abstract, perfect notion of correctness.

The predicate $corr$ is defined such that for $p \in P$, $s \in S$, $corr(p, s)$ 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 $p \in P, s \in S, t \in T$ $ok(t, p, s)$ implies that $p$ is judged as correct with respect to specification $s$ for test $t$. Furthermore, $ok$ is defined such that $\forall p \in P, \forall s \in S, \forall t \in T\ corr(p, s) \Rightarrow ok(t, p, s)$, 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 [30]. Selecting an oracle is the problem of *oracle selection*, 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 $p \in P$ and $s \in S$, if $corr(p, s)$ then we know that $\forall t \in T,\ ok(t, p, s)$. 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 often (always?) be the case that the program $p$ does not satisfy the specification $s$, 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 $O$ of test oracles. We state that an oracle $o \in O$ is a predicate:

$$o \subseteq T \times P$$

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 $t \in T$. This predicate is[2]:

$$corr_t \subseteq T \times P \times S$$

The predicate $corr_t(t, p, s)$ holds if and only if the specification $s$ holds for program $p$ when running test $t$. Obviously,

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

[2]Note that the $t$ subscript in $corr_t$ is used to differentiate the predicate from Gourlay's $corr$ predicate. It does not relate to any specific test $t$

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

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

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 $s \in S$ to be the true (idealized) specification of the desired functionality of program $P$, possibly including internal state behavior. As mentioned earlier, 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.

## 3.1 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, for example, 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 testing, 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 detect 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 [23]. 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 [37], noting several practical challenges concerning test oracle construction; we are unaware of any formulation of these oracle properties however.

In this paper, 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 [2], 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*.

### *3.1.1 Oracle Data*

As noted above, the problem of constructing an oracle, while in principle simply involved partitioning each $(t, p)$ tuple into *True* and *False* outcomes, is quite complex in practice. Many factors in oracle selection are dependent on the method underlying the construction of the oracle; these factors are outside the scope of our framework.

One factor present in all oracles, however, is the portion of the program state considered by the oracle, which we term the *oracle data*. For example, a commonly used oracle in the testing literature is one which determines correctness for test $t$ by comparing the outputs produced by the program to outputs specified in the oracle. The oracle data for this oracle is the set of outputs. If the oracle were instead to consider the value of every internal variable as well as the output, the oracle data would be the set of program variables (internal as well as outputs). For simplicity, in this paper we limit our discussion to oracles where the oracle data is a set of variables.

In our discussions, we will use as examples oracles that operate by comparing values produced by the program for some test against expected values for said tests. We will refer to such oracles as *input/expected value oracles*. When presenting several oracles for the same system, these oracles will typically differ in their oracle data.

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 [30]. 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* and *oracle data*).

## 3.2 Test and Oracle Adequacy

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

$$M : P \times S \rightarrow T$$

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

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

that is, test methods producing sets of tests. Nevertheless, the use of *test coverage criteria* (also called *test data adequacy criteria* [38] or *test selection criteria* [16]) where one or more (or none) 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 [36]:

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

We are exploring 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:

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

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:

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

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 5, we will explore an existing example of a complete adequacy criterion.

# 4. ORACLE COMPARISONS

By extending Gourlay's framework with a set $O$ 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 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 [17, 41] and discussed in this paper in Section 5.1.

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, i.e., it 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.

## 4.1 Power Comparison

Our first relation is based on Gourlay's definition of *power* [17]. 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:

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

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 2: 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 2. 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:

$$\begin{array}{lll} o_x & >_{T_t} & o_y \\ o_y & >_{T_f} & o_x \end{array}$$

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

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

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, $o_1 \geq o_2$.

## 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.

Weyuker et al. recognized this problem with respect to test coverage criteria and have defined a more useful probabilistic comparison between test criteria called *PROBBETTER* [41]. (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 [38]. 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 $o_1\ PB_T\ o_2$, where $T$ 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:

$$o_1(t,p) \quad \wedge \quad o_2(t,p) \tag{1}$$
$$\neg o_1(t,p) \quad \wedge \quad \neg o_2(t,p) \tag{2}$$
$$\neg o_1(t,p) \quad \wedge \quad o_2(t,p) \tag{3}$$

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 $t \in TS$, $o_1$ is detects a fault that $o_2$ does not. Therefore $o_1\ PB_{TS}\ o_2$.

## 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 $P$ and the set of tests $T$, including software testability [33], various test coverage criteria, and the test coverage criteria comparison relations of power, *PROBBETTER*, subsumes, etc. [41].

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, 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. In the remainder of the paper, we will explore how our extended framework explicitly considering test oracles influences the testing process.

# 5. APPLICATIONS 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.

## 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.

### *5.1.1 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 [36] 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:

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

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 [41], 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 consider-

ing 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.

### 5.1.2 Probabilistic Comparison

The *power* relation between test coverage criteria is known to be vacuous for most criteria and is thus of limited value [41]. We extend the Weyuker et al. *PB* relation [41] 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: PROBBETTER Example Program**

To demonstrate the effect of oracles on *PB*, consider the following example. Let $p$ be the program in Figure 3, 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 $s$:

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

### 5.1.3 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 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.

## 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 [11]. 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 [11], 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 [21], 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 [43], 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. Recall that in Section 3.2 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.

## 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* [34]. 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 [33, 34, 35].

Voas et al. define several testability metrics [33]. 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.

### 5.3.1 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 [35].

### 5.3.2 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 [33], thus using testability information to improve the testing process. As noted above (and by Voas [35]) 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.

# 6. ADDITIONAL RELATED WORK

We have discussed several related works throughout this paper. In this section, we discuss additional related works, with a focus on work in the theory of testing. To the best of our knowledge, our work is the only foundational work focused on how test oracles influence the testing process, though Weyuker has an interesting discussion on practical difficulties encountered in testing practice [37].

Goodenough and Gerhart's seminal work outlines a theory of testing based on test data selection [16]. Weyuker et al. [40], Budd and Angluin [9], and Gourlay [17] subsequently highlight problems in this theory; nevertheless, Goodenough and Gerhart's ideal of a testing criterion capable of finding all faults in a program captures the general goal of test selection and subsequent theories of program testing generally focus on methods of test selection. Gourlay presents a mathematical framework for testing [17], and uses it to re-interpret previous work by Goodenough and Gerhart [16], Howden [20], Gellar [15], and Weyuker [40].

There exists a large body of formal and semi-formal work on the specific problem of test selection criteria. Weyuker et al. proposes a set of axioms for test data adequacy criteria that charac-

terize "good" test data adequacy criteria [38]. This idea is further discussed by Zhu and Hall [44, 45], by Parrish and Zweben [26, 27], and by Weyuker again [39]. Formal analysis of methods of comparing test selection criteria has been performed by many authors, including Weyuker, Weiss and Hamlet [41], and by Frankl and Weyuker (specifically for partition testing methods) [12]. Hierons illustrates how the presence of test hypotheses and fault domains influence comparison of test selection criteria [19].

Several theories of program testing based on test selection exist. Morell introduces a theory of fault-based testing in which the goal of testing is to show the absence of a certain set of faults [25]. Hamlet presents an outline of a theory of software dependability arguing that the foundations of software testing should be statistical in nature [18]. Zhu and He propose a theory of behavior observation for testing concurrent software systems [47].

Bernot, Gaudel and other authors have developed a theory of testing based on formal specifications [14, 4, 5]. 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 work uses algebraic specifications as the basis of testing; tests are created based on the axioms of the algebra that define the program interface. Gaudel notes that it is not sufficient to provide a statement of correctness (specification) and program, as it is often the case that certain parts of the program necessary to ascertain correctness are not observable, e.g., "opaque" type equality and internal component state [14]. The *oracle* defines the often imperfect mapping between the specification of correctness and what it is possible to observe about the outcome of a test. The authors also discuss the relationship between testing hypotheses and test selection criteria, and the need for test oracles. Note that this body of work is in terms of algebraic sorts and is not applicable to most real-world testing situations; furthermore, the formalization is quite terse.

# 7. CONCLUDING REMARKS

In this paper, 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 include the notion of test oracles in the framework and point out the crucial interrelationship between tests, programs, and oracles. To accomplish our goals, we extended Gourlay's well known framework to account for oracles, allowing us to discuss the problem of oracle selection and explore oracle properties. We then continued to reexamine previous work in testing, demonstrating the effect of explicitly considering oracles. It is worth reiterating that our goal is not yet another formalization of software testing; our goal is to provide solid foundation for the future empirical exploration of software testing. Given our results, we make two recommendations related to future testing research directions.

First, in both theoretical and empirical testing research we must acknowledge all factors influencing the efficacy of a testing approach; *we must state and defend our all relevant assumptions*. We are often interested in exploring only one or two aspects of the testing process. Nevertheless, we must be aware that our results are influenced by other factors, for example, the choice of oracle. In some cases, simply making our assumptions explicit is sufficient. This was the case in the definition of the *PROBBETTER* relation for test coverage criteria for example—the results were sound, but only with respect to a specific oracle.

In other cases, we must argue that our assumptions are reasonable. This is particularly important in empirical testing research since it is labor intensive and time consuming and, thus, is generally only capable of thoroughly exploring one factor at the time, holding other factors constant. Authors must explicitly chose constant factors, argue why they were chosen and discuss the level of generalizablity resulting from these choices. For example, in [10] Chen et al. explore the effect of fault exposure estimates on the efficacy of the testing process. This is a exceptionally well conducted study; nevertheless, the results are dependent on the tests used (category partitioning, with manually generated tests), the oracle used (presumably output-only), and the characteristics of the programs studied (the popular Siemens programs [22]). By placing a greater emphasis on these factors, the ability of other researchers to interpret and apply the results could be greatly improved—ignoring or leaving assumptions unstated or implicit makes interpreting and comparing empirical studies impossible and hinders progress in the field.

This leads to our next recommendation; *greater emphasis should be placed how combinations of factors influence the testing process*. Some work in this area, discussed previously, has been done. Nevertheless, we strongly believe that more work exploring how combinations of factors related to the program under test, the tests themselves, and the oracle used influence the testing process is necessary to better understand and make improvements in the efficacy of the testing process. For example, questions such as how to mate a coverage criterion with a suitable oracle and what coverage criterion suits a particular program structure have not yet been systematically addressed.

# 8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous ICSE referees and Elaine Weyuker for their comments on this work.

# 9. REFERENCES

[1] Z. Al-Khanjari, M. Woodward, and H. Ramadhan. Critical analysis of the pie testability technique. *Software Quality Journal*, 10(4):331–354, 2002.

[2] 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.

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

[4] 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.

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

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

[7] L. 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.

[8] L. 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.

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

[10] W. Chen, R. 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.

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

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

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

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

[15] M. Geller. Test data as an aid in proving program correctness. In *Proc. of the 3rd ACM SIGACT-SIGPLAN Symp. on Principles on Programming Languages*, pages 209–218. ACM New York, NY, USA, 1976.

[16] 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.

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

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

[19] R. 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] A. Rajan, M. Whalen, and M. 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.

[29] S. Rayadurgam and M. P. 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.

[30] 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.

[31] 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.

[32] M. Staats, M. Whalen, and M. Heimdahl. Better testing through oracle selection (NIER track). In *Proc. of the Int'l Conf. on Software Engineering 2011*, 2011.

[33] J. Voas. Dynamic testing complexity metric. *Software Quality Journal*, 1(2):101–114, 1992.

[34] J. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. on Software Engineering*, 18(8):717–727, 1992.

[35] J. Voas and K. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994., 5th Int'l Symposium on*, pages 152–157, 1994.

[36] S. Weiss. Comparing test data adequacy criteria. *ACM SIGSOFT Software Engineering Notes*, 14(6):42–49, 1989.

[37] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465, 1982.

[38] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. on Software Engineering*, 12(12):1128–1138, 1986.

[39] E. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[40] E. Weyuker and T. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. on Software Engineering*, pages 236–246, 1980.

[41] E. Weyuker, S. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proc. of the Symposium on Testing, Analysis, and Verification*, page 10. ACM, 1991.

[42] W. Wong, J. Horgan, S. London, and A. 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.

[43] M. 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.

[44] H. Zhu. Axiomatic assessment of control flow-based software test adequacy criteria. *Software Engineering Journal*, 10(5):194–204, 1995.

[45] H. Zhu and P. Hall. Test data adequacy measurement. *Software Engineering Journal*, 8(1):21–29, 1993.

[46] H. Zhu, P. Hall, and J. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[47] H. Zhu and X. He. A theory of behaviour observation in software testing. Technical report, 1999.