# Detecting Faults vs. Revealing Failures: Exploring the Missing Link

Amani Ayad[1,*], Samia AlBlwi[2], and Ali Mili[2]
[1]Kean University, Union NJ, USA
[2]NJIT, Newark NJ, USA
amanayad@kean.edu, sma225@njit.edu, mili@njit.edu
*corresponding author

*Abstract*—When we quantify the effectiveness of a test suite by its mutation coverage, we are in fact equating test suite effectiveness with fault detection: to the extent that mutations are faithful proxies of actual faults, it is sensible to consider that the effectiveness of a test suite to kill mutants reflects its ability to detect faults. But there is another way to measure the effectiveness of a test suite: by its ability to expose the failures of an incorrect program (or, equivalently, its ability to give us confidence in the correctness of a correct program). The relationship between failures and faults is tenuous at best: a fault is the adjudged or hypothesized cause of a failure. Whereas a failure is an observable, verifiable, certifiable effect, a fault is someone's hypothesis about the possible cause of the observed effect. The same failure may be attributed to more than one fault or combination of faults. In this paper we raise two questions: is the ability to detect faults the same as the ability to reveal failures? If not, which is the better measure of test suite effectiveness? We do not give definite answers to these questions, but we use empirical data to challenge some assumptions and show why these questions are worth answering.

*Keywords–software testing; test suite effectiveness; mutation coverage; semantic coverage; detecting faults; exposing failures.*

## 1. Detecting Faults vs. Revealing Failures

> Users do not see errors in software,
> they see failures in execution.
> Mills et al. [44]

### 1.1 Defining Failures and Faults

In this paper we adopt the terminology of Avizienis et al [9], and apply it specifically to software: a *fault* is a syntactic feature of a program that precludes it from being correct with respect to a given specification; an *error* at a particular step in the execution of the program is a state of the program's execution that differs from the intended (correct) state at that step; a *failure* of the program for a particular input with respect to some specification is the observation that the program's output for the given input violates the specification. Hence, a fault is an attribute of the program's source code, an error is an attribute of the program's state during execution, and a failure is an attribute of the program's behavior. When a fault causes an error, we say that the fault is *sensitized*; when an error in some execution state causes errors in subsequent states, we say that the error has been *propagated*; else we say that the error has been *masked*. When the error is propagated all the way to the final state, it causes a failure.

Given a program and a specification, a failure of the program is the event whereby the behavior of the program for a particular input violates the specification. To the extent that the specification is precise, the occurrence of a failure is an observable, verifiable, repeatable (if the program is deterministic) event. By contrast, a fault in a program is not easy to define: In [34], [9], [33], [35] Laprie et al. define a fault as the *"adjudged or hypothesized cause of an error"*; this definition is vague, first because adjudging and hypothesizing are highly subjective human endeavors, which cannot be used as a basis for a definition; and second because the concept of error is itself insufficiently defined, as it depends on a detailed characterization of correct system states at each stage of a computation, which is usually unavailable. According to the IEEE Standard *IEEE Std 7-4.3.2-2003* [1], a software fault is "*An incorrect step, process or data definition in a computer program*". This definition is also inadequate, since it merely replaces an undefined concept (fault) by another (correctness/ incorrectness), but most of all because it assumes that we have an independent specification for what is a *correct* step, process or data definition at every location in the program. In fairness, we acknowledge that defining software faults is fraught with difficulties:

- *Discretionary determination*. Usually we determine that a program part is faulty because we think we know what the designer intended to achieve in that particular part, and we find that the program does not fulfill the designer's intent; clearly, this determination is only as good as our assumption about the designer's intent.
- *Contingent determination*. The same faulty behavior of a software product may be repaired in more than one way, possibly involving more than one part; hence the determination that one part is a fault is typically contingent upon the assumption that other parts are not in question.
- *Tentative determination*. If a program has only one fault, then we can characterize the fault by the fact that once it is repaired, the program becomes correct; but since programs typically have several faults, we have to assume that once we repair a fault, the program does not become (absolutely) correct, it just becomes (relatively) more-correct than it was initially. Hence, the definition of a fault ought to be based on some concept of relative correctness.

- *Fault Bookkeeping*. In [30], [31], Khaireddine et al. distinguish between three measures of faultiness of a program: *fault density* (the number of faults in a program), *fault multiplicity* (the number of program mutations that represent a unitary fault) and *fault depth* (the minimal number of fault repairs that separate a program from correctness). Khaireddine et al. argue that the most important measure of faultiness if not fault density but fault depth. When a program has $N$ faults and we repair one of them, we do not necessarily end up with $(N-1)$ faults; the number of faults remaining in the program depends on which fault was repaired and how it was repaired.

### 1.2 Measuring Test Suite Effectiveness

Whereas we can all agree that faults cause failures, we find that while failures are easy to define formally as observable, verifiable, certifiable effects, faults are the adjudged or hypothesized causes of the observed effects. The relationship between the observed effects and the hypothesized causes is a tenuous relation, since it is based on human judgement and a wide range of implicit assumptions (about the programmer's intent, about which program parts are assumed correct, about what makes a program more-correct, etc). Given the mission to quantify the effectiveness of test suites, we consider two possible approaches:

- *Test Suite Effectiveness as the Ability to Detect Faults*.
- *Test Suite Effectiveness as the Ability to Reveal Failures*.

This duality raises two questions:

- *First: Are these two attributes correlated?* If test suite $T$ is better than test suite $T'$ in terms of fault detection, how likely is it to be also better in terms of ability to reveal failures.
- *Second: If they are not, which attribute is a better measure of test suite effectiveness?*

In this paper we do not give definite answers to either question but we provide some empirical evidence to the effect that the answer to the first question is negative, and some analytical arguments to the effect that the answer to the second question favors the ability to reveal failures over the ability to detect faults. Most of all, we argue that these questions are worth investigating, and are worth exploring to draw practical lessons.

### 1.3 Related Work

To the best of our knowledge, most existing metrics of test suite effectiveness focus on faults, and equate effectiveness with ability to detect faults. Consequently, test suite effectiveness has traditionally been measured by a test suite's ability to achieve coverage, including syntactic coverage (statements, branches, conditions, paths, etc) [32], [22], [13], [40], [3], [37], [26], [21], [55], [24] and mutation coverage [52], [2], [8], [17], [50], [46], [19], [29], [14], [56]. Because mutants may act as proxies for actual faults [29], mutation coverage has often been used as a reference to assess alternative measures of coverage [20], [36], [57], [47], [54], [28]. We argue in this paper that while mutation coverage is a better measure of effectiveness than syntactic coverage metrics, it has ample issues of its own.

### 1.4 Agenda

In the next section we present some elementary mathematical background that we use in section 3 to discuss the concept of *semantic coverage*, which we adopt as a measure of a test suite's effectiveness to reveal failures; in section 4 we introduce the operator of *mutation tally*, and discuss why we use it as a measure of a test suite's ability to detect faults. In section 5 we present an experiment in which we take a benchmark program, generate twenty test suites thereof, then:

- We evaluate their semantic coverage under six distinct circumstances,
- We evaluate their mutation tally under four distinct mutant generation policies,

and we consider to what extent mutation tally and semantic coverage rank the test suites in the same way. The design of the experiment is discussed in 5, its results are given in raw form in section 6, and analyzed in section 7. In the conclusion we summarize our results, critique them, and sketch directions of further research.

## 2. Relational Mathematics for Correctness

The goal of this section is to introduce some mathematical notations that we use throughout the paper, and to use these mathematics to introduce two properties that pertain to software testing, namely absolute correctness and relative correctness. Absolute correctness is important because it is the property we test programs against; and relative correctness is important because it gives meaning to the task of fault repair (aka debugging) since repairing a fault makes the program more-correct than it was (while it may still be incorrect, due to the presence of other faults).

### 2.1 Definitions and Notations

In this paper, we use relations and functions [12] to capture program specifications and program semantics. We can model the semanticss of a program by means of a function from inputs to outputs, or as a function from initial states to final states. For the sake of simplicity, and without loss of generality, we adopt the latter model, because it involves homogeneous functions and relations, and yields a simpler algebra. We represent sets by C-like variable declarations, and we generally denote sets (referred to as *program spaces*) by $S$, elements of $S$ (referred to as *program states*) by lower case $s$, specifications (binary relations on $S$) by $R$ and programs (functions on $S$) by $P$. We denote the domain of a relation $R$ by $dom(R)$. Because we model programs and specifications by homogeneous relations / functions, we talk about initial states and final states, rather than inputs and outputs.

Among the operations on relations, we consider the set theoretic operations of union, intersection, and complement; we also consider the *prerestriction* of a relation $R$ on set $S$ to a subset $A$ of $S$ as the relation denoted by

$$_{A\backslash}R = \{(s, s') | s \in A \wedge (s, s') \in R\}.$$

A specification $R$ on space $S$ includes all the initial state / final state pairs that the specifier considers correct; hence the domain of a specification $R$ ($dom(R)$) includes all the initial states for which candidate programs must make provisions (i.e. generate a final state). When a program $P$ on space $S$ is executed on initial state $s$, it may terminate normally after a finite number of steps in a final state $s'$; we then say that $P$ *converges* for initial state $s$. Alternatively, it may fail to terminate (due to an infinite loop), or it may attempt an illegal operation, such as a division by zero, an array reference out of bounds, a reference to a nil pointer, an arithmetic overflow, etc; we then say that it *diverges* for initial state $s$. Given a program $P$ on space $S$, the function of $P$, which we denote by the same symbol, is the set of pairs of states $(s, s')$ such that if execution of $P$ starts at state $s$, it converges and yields state $s'$. From this definition, it stems that the domain of program $P$ ($dom(P)$) is the set of initial states $s$ such that execution of $P$ on $s$ converges.

We define an ordering relation between specifications whose interpretation is that one relation captures more stringent requirements, hence is harder to satisfy.

*Definition 1:* Given two relations $R$ and $R'$ on space $S$, we say that $R'$ *refines* $R$ if and only if:

$$dom(R) \subseteq dom(R') \wedge_{dom(R)\setminus} R' \subseteq R.$$

This definition is equivalent, modulo differences of notation, to traditional definitions of refinement, which equate refinement with weaker preconditions and stronger postconditions [25], [45], [11], [18], [49], [10].

### 2.2 Partial and Total Correctness

The distinction between partial correctness and total correctness has long been a feature of the study of correctness verification [39], [23], [45], but not considered in testing. Yet, testing a program for partial correctness is different from testing it for total correctness. We can cite at least two arguments to this effect:

- If a program $P$ is executed on some initial state $s$ and it fails to converge, then the conclusion we draw depends on the correctness property we are testing it against: if we are testing $P$ for total correctness we conclude that $P$ fails the test, hence is incorrect; if we are testing it for partial correctness we conclude that the choice of $s$ is incorrect, and choose another initial state.
- The main purpose of test data selection is to generate a finite and small test suite $T$ to represent a potentially infinite set of initial states: if we are testing $P$ for total correctness with respect to $R$, the set we are trying to represent is $dom(R)$; for partial correctness, that set is $(dom(R) \cap dom(P))$.

Hence the distinction between partial correctness and total correctness is relevant for software testing, even though it has not traditionally been given much consideration. We adopt the following definitions for total and partial correctness.

*Definition 2:* Due to [43]. Program $P$ on space $S$ is said to *totally correct* with respect to specification $R$ on $S$ if and only if:

$$dom(R) = dom(R \cap P).$$

*Definition 3:* Due to [42]. Program $P$ on space $S$ is said to be *partially correct* with respect to specification $R$ on $S$ if and only if:

$$dom(R) \cap dom(P) = dom(R \cap P).$$

Khaireddine et al. show in [31] that these definitions are equivalent, modulo differences of notation, to traditional definitions of total and partial correctness [39], [16], [27], [23].

### 2.3 Detector Sets and Relative Correctness

The concepts of detector sets and relative (partial or total) correctness are useful for our purposes, as we use them to check the validity of our definition of semantic coverage. Given a program $P$ on space $S$ and a specification $R$ on $S$, the detector set of $P$ with respect to $R$ is the set of initial states that disprove the correctness of $P$ with respect to $R$. Given that there are two standards of correctness, there are two versions of differentiator sets.

*Definition 4:* Due to [41]. Given a program $P$ on space $S$ and a specification $R$ on $S$:

- The *detector set for total correctness* of program $P$ with respect to $R$ is denoted by $\Theta_T(R, P)$ and defined by:

$$\Theta_T(R, P) = dom(R) \cap \overline{dom(R \cap P)}.$$

- The *detector set for partial correctness* of program $P$ with respect to $R$ is denoted by $\Theta_P(R, P)$ and defined by:

$$\Theta_P(R, P) = dom(R) \cap dom(P) \cap \overline{dom(R \cap P)}.$$

Intuitive interpretation: the term $dom(R \cap P)$ represents the set of initial states for which program $P$ delivers a final state that satisfies specification $R$; we call it the *competence domain* of program $P$ with respect to specification $R$. For total correctness, $dom(R)$ represents all the initial states for which $P$ must behave according to $R$; in other words, $dom(R)$ represents all the initial states that must be in $dom(R \cap P)$; hence the initial states that disprove the total correctness of $P$ with respect to $R$ are all the elements of $dom(R)$ that are outside $dom(R \cap P)$. A similar justification may be presented for the detector set of partial correctness. When we want to talk about a detector set but we do not wish to specify whether we refer to total correctness or partial correctness, we use the notation $\Theta(R, P)$.

Refer to Figure 1; the green area represents the competence domain of $P$ with respect to $R$, i.e. the set of inputs for which $P$ satisfies specification $R$; the detector set for total correctness in colored in red, and the detector set of partial correctness is colored in orange. Since total correctness is a stronger property than partial correctness, it is easier to disprove total correctness than partial correctness; hence the detector set for total correctness (red) is a superset of the detector set for partial correctness (orange).

The following Propositions stem readily from the definitions (hence will be given without proof), and are perfectly easy to understand intuitively.

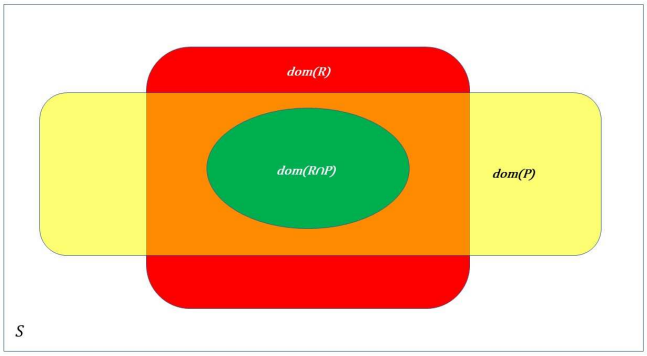



Figure 1. Detector Sets for Partial Correctness (orange) and Total Correctness (red+orange)

| | Partial Correctness | Total Correctness |
|---|---|---|
| Absolute Correctness $P$ correct wrt $R$ | $\Theta_P(R,P)=\emptyset$ | $\Theta_T(R,P)=\emptyset$ |
| Relative Correctness $P'$ more-correct than $P$ wrt $R$ | $\Theta_P(R,P') \subseteq \Theta_P(R,P)$ | $\Theta_T(R,P') \subseteq \Theta_T(R,P)$ |

TABLE I
DEFINITIONS OF CORRECTNESS

*Proposition 1:* A program $P$ on space $S$ is totally correct with respect to a specification $R$ on $S$ if and only if the detector set of $P$ for total correctness with respect to $R$ is empty.
*Proposition 2:* A program $P$ on space $S$ is partially correct with respect to a specification $R$ on $S$ if and only if the detector set of $P$ for partial correctness with respect to $R$ is empty.
Whereas absolute (partial or total) correctness is a bipartite property between a program and a specification, relative (partial or total) correctness is a tripartite property between two programs, say $P$ and $P'$, and a specification, say $R$.
*Definition 5:* Due to [15]. Given a specification $R$ on space $S$ and two programs $P$ and $P'$ on $S$, we say that $P'$ is *more-totally-correct* than $P$ with respect to $R$ if and only if:

$$dom(R \cap P) \subseteq dom(R \cap P').$$

Using detector sets, we can readily infer the following Proposition.
*Proposition 3:* Given a specification $R$ on space $S$ and two programs $P$ and $P'$ on $S$, $P'$ is more-totally-correct than $P$ with respect to $R$ if and only if the detector set of $P'$ for total correctness with respect to $R$ is a subset of the detector set of $P$ for total correctness with respect to $R$.
It is perfectly intuitive to consider that a program that has a smaller (by set inclusion) detector set is more-totally-correct (closer to being totally correct) than a program that has a larger detector set; ultimately, when the detector set of a program becomes so small that it is empty, the program is (absolutely) correct. We extend this definition to partial correctness, whence we obtain the simple characterization of absolute correctness and relative correctness given in Table I.

## 3. Semantic Coverage: Revealing Failures

In order to investigate the relationship between the ability to detect faults and the ability to reveal failures, we must devise means to quantify each of these attributes. In this section, we consider the latter, and we refer to it as the *semantic coverage* of a test suite.

### 3.1 Definitions

Whereas traditional coverage metrics treat the effectiveness of a test suite as an attribute of the test suite and the program under test, we must recognize that whether a test suite is able to reveal the failure of a program does also depend on the standard of correctness that we are testing the program against (total or partial), and on the specification with respect to which correctness is tested. In order for a test suite to reveal all the failures of a program, it must be a superset of its detector set. What precludes a test suite $T$ from revealing all the failures of a program is the set of initial states that are in the detector set but are not in $T$:

$$\Theta(R,P) \cap \overline{T}.$$

The smaller this set, the better the test suite $T$ is; if we want a metric that increases with the quality of $T$ rather than to decrease, we take the complement of this expression. Whence the following definition.
*Definition 6:* Due to [4]. Given a specification $R$ on space $S$ and a program $P$ on $S$, the *semantic coverage* of a test suite $T$ is denoted by $\Gamma_{P,R}(T)$ and defined by:

$$\Gamma_{P,R}(T) = T \cup \overline{\Theta(P,R)}.$$

This definition represents, in effect, two distinct definitions, depending on whether we are interested to test $P$ for partial correctnes or total correctness:

- *Partial Correctness*. The semantic coverage of test suite $T$ for program $P$ relative to partial correctness with respect to specification $R$ is denoted by $\Gamma^{PAR}_{P,R}(T)$ and defined by:

$$\Gamma^{PAR}_{P,R}(T) = T \cup \overline{\Theta_P(P,R)},$$

- *Total Correctness*. The semantic coverage of test suite $T$ for program $P$ relative to total correctness with respect to specification $R$ is denoted by $\Gamma^{TOT}_{P,R}(T)$ and defined by:

$$\Gamma^{TOT}_{P,R}(T) = T \cup \overline{\Theta_T(P,R)},$$

To gain an intuitive feel for this formula, consider under what condition it is minimal (the empty set) and under what condition it is maximal (set $S$ in its entirety).

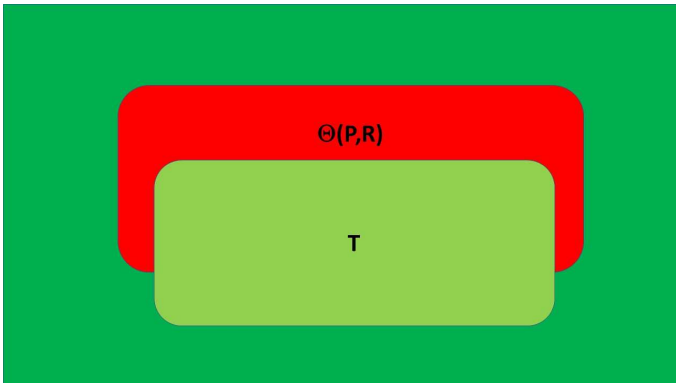



Figure 2. Semantic Coverage of Test $T$ for Program $P$ with respect to $R$ (shades of green)

- $\Gamma_{P,R}(T) = \emptyset$. The semantic coverage of a test $T$ for program $P$ with respect to specification $R$ is empty if and only if $T$ is empty and the complement of the detector set of $P$ with respect to $R$ is empty; in such a case the detector set of $P$ with respect to $R$ is all of $S$. In other words, even though any element of $S$ exposes a failure of $P$ with respect to $R$, $T$ does not reveal that $P$ is incorrect since it is empty.
- $\Gamma_{P,R}(T) = S$. If the union of two sets equals $S$, the complement of each set is a subset of the other set. Whence: $\Theta(P,R) \subseteq T$. In other words, $T$ contains all the tests that reveal the failure of $P$ with respect to $R$. This is clearly the attribute of an ideal test suite.
  As a special case, if $P$ is correct with respect to $R$, then the semantic coverage of any test suite $T$ with respect to $P$ and $R$ is $S$ (if there are no failures to reveal, then any test suite reveals all the failures).

Refer to Figure 2; the semantic coverage of test suite $T$ for program $P$ with respect to specification $R$ is the area colored (both shades of) green. The (partially hidden) red rectangle represents the detector set of $P$ with respect to $R$; the dark green area represents the complement of this set, i.e. in fact all the test data that need not be exercised; the light green area represents test suite $T$.

### 3.2 Criteria for Semantic Coverage

The semantic coverage of a test suite $T$ for a standard of correctness (partial or total) of a program $P$ with respect to a specification $R$ depends on four factors: $T$, $P$, $R$ and the standard of correctness. In this section we discuss how one would want a measure of test suite effectiveness to vary as a function of each of these parameters, then we show that semantic coverage does meet the selected criteria. We start with citing and justifying the criteria.

- *Monotonicity with respect to $T$*. Of course we want the effectiveness of a test suite to be monotonic with respect to $T$: if we replace $T$ by a superset, we get a higher semantic coverage.
- *Monotonicity with respect to $R$*. Specifications are ordered by refinement (re: Definition 1), whereby a more-refined specification represents a more stringent requirement. We argue that it is easier to test a program for correctness against a specification $R$ than against a specification $R'$ that refines $R$; indeed, a more-refined specification involves a larger input domain (hence a larger set to cover) and stronger output conditions (hence more conditions to verify). Whence we expect that the same test suite $T$ have lower semantic coverage for more-refined specifications: i.e. semantic coverage ought to decrease when $R$ grows more-refined.
- *Monotonicity with respect to $P$*. If and only if program $P'$ is more-correct than program $P$, the detector set of $P'$ is a subset of the detector set of $P$, which means that we have fewer failures of $P'$ to reveal than failures of $P$. Hence the semantic coverage of a test suite $T$ ought to be higher for a more-correct program.
- *Monotonicity with respect to the standard of correctness*. Total correctness is a stronger property than partial correctness, hence it is more difficult to test a program for total correctness than for partial correctness. Consequently, the same test suite $T$ ought to have a lower semantic coverage for total correctness than for partial correctness (the same tool would be less effective against a more difficult task than an easier task).

We present below Propositions to the effect that semantic coverage satisfies all these monotonicity properties; these are due to [4], and are given without proof.

*Proposition 4: Monotonicity with respect to $T$*. Given a program $P$ on space $S$ and a specification $R$ on $S$, and given two subsets $T$ and $T'$ of $S$, if $T \subseteq T'$ then:

$$\Gamma^{TOT}_{R,P}(T) \subseteq \Gamma^{TOT}_{R,P}(T'),$$
$$\Gamma^{PAR}_{R,P}(T) \subseteq \Gamma^{PAR}_{R,P}(T').$$

*Proposition 5: Monotonicity with respect to the standard of Correctness*. Given a program $P$ on space $S$, a specification $R$ on $S$, and test suite $T$ (subset of $S$), the semantic coverage of $T$ for partial correctness of $P$ with respect to $R$ is greater than or equal to the semantic coverage for total correctness of $P$ with respect to $R$:

$$\Gamma^{TOT}_{R,P}(T) \subseteq \Gamma^{PAR}_{R,P}(T).$$

*Proposition 6: Monotonicity with respect to relative correctness of $P$*. Given a specification $R$ on space $S$ and two programs $P$ and $P'$ on $S$, and a subset $T$ of $S$. If $P'$ is more-totally-correct (resp. more-partially-correct) than $P$ with respect to $R$ then:

$$\Gamma^{TOT}_{[R,P]}(T) \subseteq \Gamma^{TOT}_{[R,P']}(T).$$
$$\Gamma^{PAR}_{[R,P]}(T) \subseteq \Gamma^{PAR}_{[R,P']}(T).$$

*Proposition 7: Monotonicity with respect to Refinement of $R$*. Given a program $P$ on space $S$ and two specifications $R$ and $R'$ on $S$, and a subset $T$ of $S$. If $R'$ refines $R$ then:

$$\Gamma^{TOT}_{[R',P]}(T) \subseteq \Gamma^{TOT}_{[R,P]}(T).$$
$$\Gamma^{PAR}_{[R',P]}(T) \subseteq \Gamma^{PAR}_{[R,P]}(T).$$

### 3.3 Intuitive Interpretation

To give the reader some intuition about what the semantic coverage of a test suite represents, and why it is a sensible measure of test suite effectiveness, we expand its formula and justify its various components. We start with the semantic coverage for total correctness:

$$\Gamma^{TOT}_{[R,P]}(T) = T \cup \overline{\Theta_T(R,P)},$$

where

$$\Theta_T(R,P) = \mathit{dom}(R) \cap \overline{\mathit{dom}(R \cap P)}.$$

Replacing $\Theta_T(R,P)$ by its formula and applying DeMorgan's laws, we find:

$$\Gamma^{TOT}_{[R,P]}(T) = T \cup \overline{\mathit{dom}(R)} \cup \mathit{dom}(R \cap P).$$

The semantic coverage of $T$ for total correctness of $P$ with respect to $R$ is the union of three terms:

- $T$: clearly a bigger $T$ is more effective than a smaller $T$.
- $\overline{\mathit{dom}(R)}$: A smaller domain of $R$ means fewer inputs to test / to worry about.
- $\mathit{dom}(R \cap P)$: A larger competence domain means fewer failures to reveal.

Likewise, a decompsotion of the formula of semantic coverage for partial correctness yields:

$$\Gamma^{PAR}_{[R,P]}(T) = T \cup \overline{\mathit{dom}(R)} \cup \overline{\mathit{dom}(P)} \cup \mathit{dom}(R \cap P).$$

Hence for partial correctness we have one extra term in the formula of semantic coverage:

- $\overline{\mathit{dom}(P)}$: From the standpoint of partial correctness, a program is tested (held accountable) only wherever it terminates / converges; the smaller the set of inputs where the program converges, the higher the semantic coverage.

## 4. Mutation Tally: Detecting Faults

In the previous section we introduce semantic coverage as the attribute we use to quantify a test suite's ability to reveal program failures; in this section we briefly discuss how we quantify a test suite's ability to detect faults. To the extent that mutations are faithful proxies of actual faults [48], [46], [7], [8], [29], [6], [28], it is sensible to use mutation coverage as a measure of a test suite's ability to detect faults. In this section we briefly discuss alternative measures of mutation coverage, and justify why we select *mutation tally*, which is the set of mutants killed by a test suite.

- RMS: Raw Mutation Score. The *raw mutation score* of a test suite in a mutation experiment is the ratio of killed mutants over generated mutants. The same value of *RMS* means vastly different things depending on whether the surviving mutants are equivalent to the base program or not; hence we dismiss *RMS* for this experiment.
- *PMS: Prorated Mutation Score*. The *prorated mutation score* of a test suite is the raio of killed mutants over the set of killable (i.e. non-equivalent) mutants. The same value of *PMS* means vastly different things depending on whether the killed mutants are all semantically distinct, all semantically equivalent, or partitioned into a large number of equivalence classes; hence we dismiss *PMS* for this experiment.
- *EMS: Equivalence-Based Mutation Score*. The *equivalence-based mutation score* of a test suite is the ratio of the number of equivalence classes killed by the test suite over the total number of equivalence classes of the killable mutants. Because EMS takes numeric values, it defines a total ordering between test suites, but the property of being a more effective test suite is a partial ordering (not all pairs of test suites can be compared for effectiveness). Representing a partial ordering by a total ordering creates a built-in potential for loss of precision; hence we dismiss *EMS* for this experiment.
- *MT: Mutation Tally*. The *mutation tally* of a test suite for a mutation experiment is merely the set of mutants that are killed by the test suite; we denote the mutation tally of test suite $T$ by $\mu(T)$. We order mutation tallies by inclusion: in the context of a mutation experiment, a test suite $T$ is considered more-effective than a test suite $T'$ if and only if all the mutants killed by $T'$ are killed by $T$. We adopt *MT* as the measure of mutation coverage we use in this experiment; we use it to represent the effectiveness of a test suite to detect faults, on the grounds that mutants can be considered as proxies for actual faults.

## 5. Empirical Study

### 5.1 The Benchmark Program and Tests

Now that we have decided how to quantify a test suite's ability to reveal failures (through semantic coverage) and a test suite's ability to detect faults (through mutation tally), we resolve to explore to what extent these two attributes are related. To this effect, we run the following experiment:

- *Sample Program, $P$*. The sample program that we use for this experiment is a method called `createNumber()` of the Java class `NumberUtils.java`, from the commons benchmark (`commons-lang3-3.13.0-src`)[1]. The size of the selected method is 170 LOC.
- *Base Test Suite, $T_0$*. We consider the test class that comes with the selected program: `class NumberUtilsTest.java`. This class includes 107 tests.
- *Test Suites $T_1$, $T_2$, ... $T_{20}$*. To generate these test suites, we run the following script, where `rand()` returns random numbers between 0.0 (inclusive) and 1.0 (exclusive):

```
threshold = 0.2;
for (int i=1; i<=20; i++)
   {print ("test suite t",i);
    int size=0;
    for (int j=1; j<=107; j++)
        {if (rand()<=threshold)
              {print (j); size++;}
    print ("size of test suite t",
            i,": ", size);}
    threshold = threshold + 0.03;}
```

Our expectation is to obtain 20 test suites $T_1 ... T_{20}$ whose sizes range between approximately $0.2 \times 107$ and $0.77 \times 107$.

[1] `https://commons.apache.org/proper/commons-lang/`

| $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ | $T_9$ | $T_{10}$ | $T_{11}$ | $T_{12}$ | $T_{13}$ | $T_{14}$ | $T_{15}$ | $T_{16}$ | $T_{17}$ | $T_{18}$ | $T_{19}$ | $T_{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 107 | 25 | 31 | 22 | 22 | 36 | 46 | 44 | 40 | 53 | 49 | 65 | 56 | 67 | 69 | 76 | 77 | 82 | 78 | 86 | 86 |

TABLE II
SIZES OF THE RANDOMLY GENERATED TEST SUITES

Table II shows the sizes of the test suites derived from this algorithm.

### 5.2 The Semantic Coverage

The semantic coverage of test suites $T_1...T_{20}$ depends not only on the program and the test suites, but also on two additional factors: what correctness standard we are considering (partial, total) and what specification we are testing the program against. We have generated three specifications, $R_1$, $R_2$, $R_3$ derived as follows: We consider the 107 elements of $T_0$ and run the base program on this data. Then we scan the list of (input,output) pairs generated by the program, and generate the specifications according to the following rules:

- $R_1$: We alter each fifth output, so that $P$ fails for each fifth input with respect to $R_1$.
- $R_2$: We alter each seventh output, so that $P$ fails for each seventh input with respect to $R_2$.
- $R_3$: We alter each eleventh output, so that $P$ fails for each eleventh input with respect to $R_3$.

In addition, we extend the domains of $R_1$, $R_2$ and $R_3$ beyond the domain of $P$, so as to make $P$ fail to converge for some elements in the domains of the specifications; this is important to distinguish between partial correctness and total correctness. For each specification, we compute the semantic coverage of test suites $T_1...T_{20}$ for partial correctness and total correctness; this yields a total of six measures of semantic coverage.

### 5.3 The Mutation Tally

We use the mutant generator *LittleDarwin* [51], [53], and we consider its mutation operators:

- *ROR*: Relational Operator Replacement.
- *AORB*: Arithmetic Operator Replacement Binary.
- *AORU*: Arithmetic Operator Replacement Unary.
- *ArORS*: Arithmetic Operator Replacement Shortcut.
- *AsORS*: Assignment Operator Replacement Shortcut.
- *COD*: Conditional Operator Delection.
- *COR*: Conditional Operator Replacement.

From these seven operators, we form four mutant generation policies as shown in Figure 3:

- *Policy 1*: ROR, AsORS, ArORS. This policy produces 61 mutants.
- *Policy 2*: COD, COR, AORB. This policy produces 58 mutants.
- *Policy 3*: ArORS, AsORS, AORU, COD, COR. This policy produces 61 mutants.
- *Policy 4*: ArORS, AsORS, AORB, AORU, COD, COR, ROR. This policy produces 133 mutants.

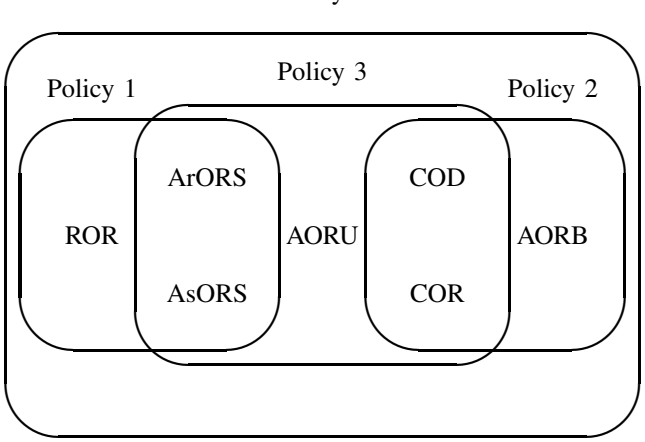



Figure 3. Mutant Generation Policies

## 6. EXPERIMENTAL DATA

### 6.1 Semantic Coverage: Exposing Failures

For each specification ($R_1$, $R_2$, $R_3$) and each standard of correctness (partial, total), we compute the semantic coverage of each test suite ($T_1...T_{20}$) and we check inclusion relations between them ($\Gamma_{P,R_n}(T_i) \subseteq \Gamma_{P,R_n}(T_j)$). For each specification and standard of correctness, this yields a graph whose nodes represent the test suites. The six graphs are shown in Table III.

One can make several observations from these graphs. As a preamble, we must specify that when an arrow goes from node $T_i$ to node $T_j$, it means that $T_j$ has higher semantic coverage than $T_i$ (i.e. is better at revealing program failures).

- The first observation we can make about these graphs is that test suites with higher indices tend to have higher semantic coverage; this is due, of course, to how these were generated, to range in size from about $0.2\times107$ to $0.77\times107$ (of course, bigger test suites tend to be better).
- The second observation can be made by comparing graphs for the same specification: the graphs for partial correctness and total correctness with respect to the same specification are different, which means that whether a test suite is better than another depends on whether we are testing the program for partial correctness or for total correctness. It appears that for each specification, the graph for total correctness is a subgraph of the graph for partial correctness, which means if a test suite is better than another for total correctness, it is necessarily better for partial correctness.
- The third observation can be made by comparing graphs across specifications: for the same standard of correctness (say, e.g. total) whether a test suite is better than another

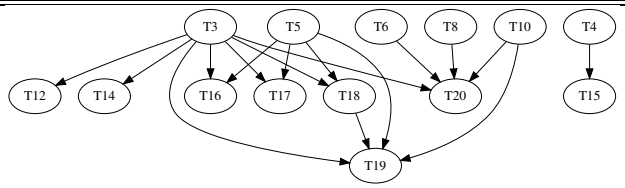
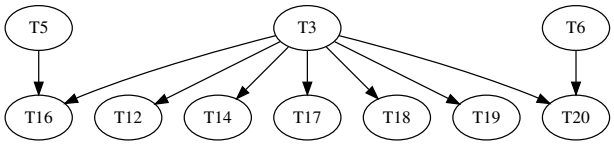
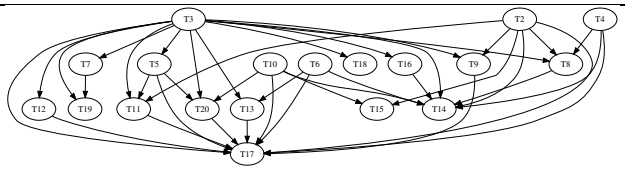
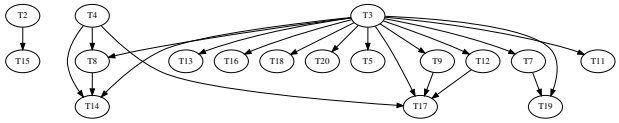
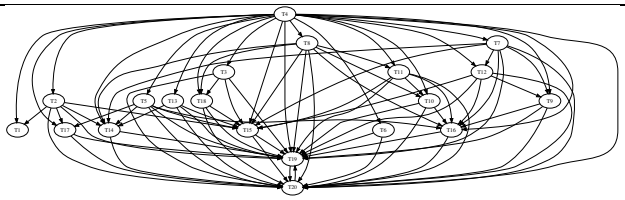
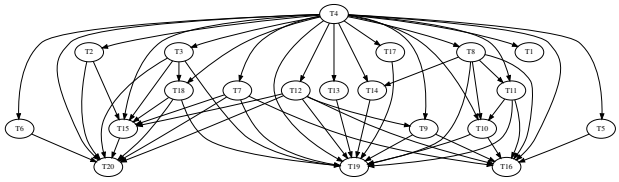
| Spec | Partial Correctness | Total Correctness |
|---|---|---|
| $R_1$ |  |  |
| $R_2$ |  |  |
| $R_3$ |  |  |

TABLE III
ORDERING TEST SUITES BY SEMANTIC COVERAGE

depends heavily on the specification against which the program is being tested for correctness. While this may seem obvious, it may mean that assessing test suite effectiveness by considering only the program and the test suite may be missing an important / influential parameter: the specification.

- The fourth observation is that the graphs become increasingly denser as we have fewer and fewer failures to report (each fifth input, vs each seventh input, vs each eleventh input). Conversely, the more failures we have to report, the fewer comparative relations exist between test suites.

### 6.2 Mutation Tally: Detecting Faults

For each mutant generation policy (Policy 1 ... Policy 4), we compute the mutation tally of each test suite ($\mu(T_1)...\mu(T_{20})$) and we rank test suite according to the inclusion relationships between their mutation tallies ($\mu(T_i) \subseteq \mu(T_j)$). Table IV shows the graphs that represent these ordering relations between the test suites. Some observation can be made on these graphs, from a cursory analysis:

- For the same reason as above, test suites with higher index tend to have higher mutation tallies, because they are generally larger sets.
- This experiment bears out an observation made in [5] to the effect that mutation coverage varies significantly according to the mutant generation policy: the same test suite can have widely varying mutation scores depending on the mutant generation policy.
- As an exception, the graphs for Policies 3 and 4 are very similar even though they use use different sets of mutation operators and are based on vastly different mutant sets (61 mutants vs 133 mutants).

### 6.3 Detecting Faults vs Revealing Failures

Table V captures the pairwise relationships between the six measures of semantic coverage and the four measures of mutation tally. Each entry of this table represents the *Jaccard* index of the corresponding graphs, which is the ratio of the number of arcs that the two graphs have in common over the total number of arcs in the two graphs.

## 7. OBSERVATIONS AND ANALYSIS

In this section we analyze the results shown in Table V, by considering in turn, the interrelations between measures of mutation tally, then the interrelations between measures of semantic coverage, then relationships between mutation tally and semantic coverage.

### 7.1 Mutation Tally vs Mutation Tally

If we focus on the top left triangle of Table V, we find that the Jaccard index betweeen the graphs derived from different mutant generation policies vary between 0.40 and 0.50, with the exception of Policy 3 vs Policy 4, which is 0.95. This is consistent with findings of other experiments, where we find that the Jaccard index that stems from different policies of the same tool fall in the range of 0.40 to 0.50, whereas the Jaccard index of different mutant generation tools can be as low as zero or even negative [5].

### 7.2 Semantic Coverage vs Semantic Coverage

In this section we focus on the lower right triangle of Table V. The Jaccard index of the graphs for partial correctness and total correctness with respect to the same specification are fairly high: 0.529 for $R_1$, 0.525 for $R_2$ and 0.647 for $R_3$. In all other cases, when they deal with different specifications, the indices are fairly low, ranging between 0.05 and 0.30. These

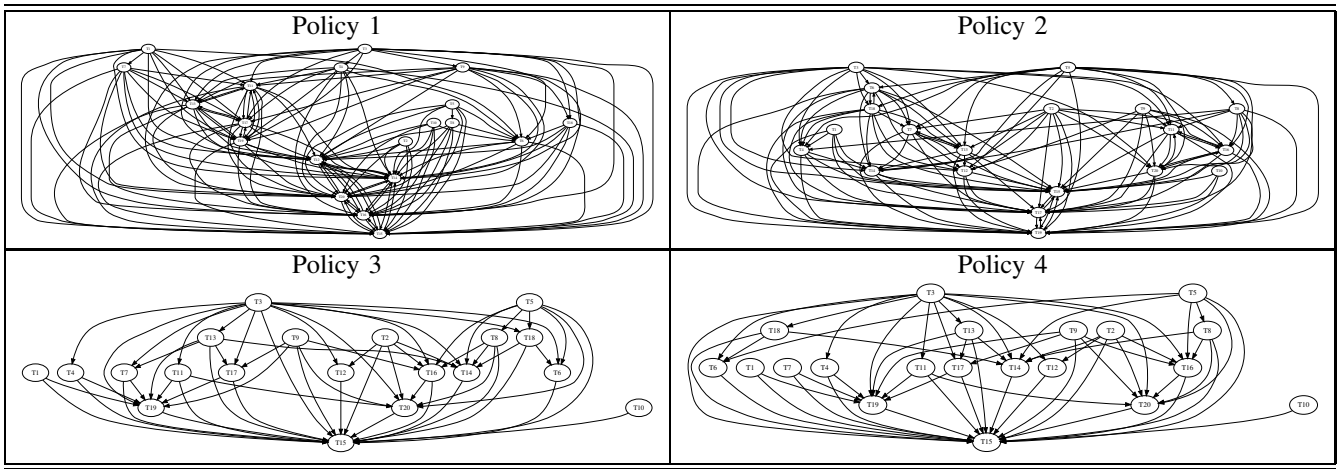



TABLE IV
ORDERING TEST SUITES BY MUTATION TALLY

| | Policy 1 | Policy 2 | Policy 3 | Policy 4 | PR1 | TR1 | PR2 | TR2 | PR3 | TR3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Policy 1 | 1.0000 | 0.4162 | 0.4028 | 0.4113 | 0.0822 | 0.0638 | 0.1602 | 0.1096 | 0.3294 | 0.2327 |
| Policy 2 | | 1.0000 | 0.5401 | 0.4793 | 0.1219 | 0.0655 | 0.2385 | 0.1360 | 0.3290 | 0.1973 |
| Policy 3 | | | 1.0000 | 0.9508 | 0.1642 | 0.1290 | 0.2169 | 0.2059 | 0.2807 | 0.2083 |
| Policy 4 | | | | 1.0000 | 0.1538 | 0.1356 | 0.2099 | 0.1970 | 0.2882 | 0.2150 |
| PR1 | | | | | 1.0000 | 0.5294 | 0.1875 | 0.2258 | 0.1333 | 0.1250 |
| TR1 | | | | | | 1.0000 | 0.1667 | 0.3043 | 0.0562 | 0.0847 |
| PR2 | | | | | | | 1.0000 | 0.5250 | 0.1261 | 0.1046 |
| TR2 | | | | | | | | 1.0000 | 0.0928 | 0.1343 |
| PR3 | | | | | | | | | 1.0000 | 0.6471 |
| TR3 | | | | | | | | | | 1.0000 |

TABLE V
JACCARD INDEX OF MUTATION TALLY (POLICY 1... POLICY 4) VS. SEMANTIC COVERAGE (PR1...TR3)

low values reflect the important role that specifications play in determining whether a test suite is effective in revealing program failures; this in turn, means that the effectiveness of a test suite cannot be considered as an attribute of the program alone, but must also involve an analysis of the specification with respect to which correctness is tested. The low values of these Jaccard indices reflect to what extent we stand to lose precision when we overlook specifications.

### 7.3 Mutation Tally vs Semantic Coverage

In this section we focus on the top right rectangle of Table V, defined by rows (*Policy 1* to *Policy 4*) and columns (*PR1* to *TR3*). The Jaccard indices in this rectangle range between 0.06 and 0.32; the average value in this rectangle is 0.186; in other words, if we consider how test suites are ranked by their ability to detect faults and how they are ranked by their ability to reveal failures, the two criteria concur on only only 18.6% of their findings.

But we observe another interesting phenomenon: For each Policy and for each standard of correctness (partial, vs total), the Jaccard index increases monotonically as we transition from $R_1$ to $R_2$ to $R_3$; as we recall, $R_1$, $R_2$ and $R_3$ differ by how often the program $P$ fails to comply to them: for each fifth input, for each seventh input, and for each eleventh input, respectively. This seems to indicate that mutation tally (ability to detect faults) and semantic coverage (ability to reveal failures) are more tightly coupled for lower failure rates than for higher failure rates. These results are illustrated in Table VI, which plots the Jaccard index of the mutation tally graph and the semantic coverage graph for partial and total correctness with respect to $R_1$, $R_2$ and $R_3$ for each mutant generation policy, as a function of the failure rate.

### 7.4 Discussion

The contrast between failures and faults is essentially a contrast between observable, verifiable effects and hypothesized, speculative causes. The same observed effect (failure) may be attributed to a wide range of possible causes (faults) and may be remedied by a wide range of possible fixes (fault repairs). This contrast may be interpreted to mean that if we want to quantify the effectiveness of test suites in a meaningful, measurable way, we may be better off defining it in terms of failures rather than faults.

There is another reason why, theoretically, it may be more sensible to focus on failures rather than faults: ultimately, the reliability and operational quality of a software product

pertains not to how many faults it has (fault density) but rather how often it fails (failure rate, or its inverse, inter-failure span). Even though failures are the result of faults, the statistical correlation between failure rate and fault density is very weak, as the impact of faults on failures varies very widely in practice: a fault in an obscure part of the code that is visited only under very exceptional circumstances does not have the same impact on reliability as a fault that is part of routine code that gets invoked at each call. In [44] Mills et al. report on empirical studies of IBM software projects where they find that the impact of faults on failure rates varies from 18 months between failures to 5000 years between failures; more than half the faults have failure rates of 1500 years between failures; specifically, they find that one may remove 60 percent of faults in a software product and enhance its reliability by a mere 3%. In other words, from the standpoint of reliability, faults are not created equal: some have a much greater impact on reliability than others, hence from the standpoint of a tester, are much more worthy of attention.

To illustrate the contrast between focusing on faults and focusing on failures, consider the hypothetical case where we have three faults in a program, say $f1$, $f2$, $f3$. We consider two test suites, say $T$ and $T'$, and we assume that $T$ detects $f1$ and $f2$ while $T'$ detects $f3$. In terms of fault detection, $T$ is better than $T'$ because it detects twice as many faults. But imagine, for the sake of argument, that $f1$ and $f2$ cause the program to fail, on average, once every 1000 executions, whereas $f3$ causes the program to fail once every 10 executions. From the standpoint of failure rate (reliability), it is better to repair $f3$ than to repair $f1$ and $f2$; hence test suite $T'$ is better than $T$, since it alerts us to a more consequential fault. When we focus on exposing failures rather than detecting faults, the failures that occur more often will naturally arise more frequently, hence leading us to repair high impact faults before (or instead of, if the test budget is limited) low impact faults.

Hence it seems that the ability to expose failures is a better measure of test suite effectiveness than the ability to detect faults, but we qualify this conclusion with an important caveat: There are really two broad families of test processes, with different goals:

- *Tests Which Focus on Fault Detection*. These include unit tests and integration tests, which are carried out under the purview of the system designers and are geared towards finding and repairing faults.
- *Tests Which Focus on Black Box Behavior*. These include acceptance tests and reliability tests [44], [38] which are carried out by third parties and are geared towards ensuring the absence (or infrequency) of failures.

It is possible that these different families of tests mandate different criteria for judging test suite effectiveness.

### 7.5 Threats to the Validity of Our Results

The results presented in this paper are based on six experiments to compute the semantic coverage of test suites, and four experiments to compute the mutation tally the same test suites; hence these results are valid only to the extent that this limited experiment is statistically significant. But the main purpose of our paper is not to deliver definite results as much as it is to shake some assumptions and raise some questions about current practice in test suite assessment.

## 8. Conclusion

In this paper we consider two alternative means to quantify the effectiveness of a test suite: By the test suite's ability to detect faults, or by its ability to expose failures. This choice raises two questions: Are these two attributes correlated? If not, which better reflects the effectiveness of a test suite?

To answer the first question, we have resolved to adopt means to represent these two attributes: we represent a test suite's ability to detect faults by its mutation tally, and we represent a test suite's ability to expose failures by its semantic coverage. Then we run an experiment in which we consider a sample benchmark program, we generate twenty test suites thereof, we compute the mutant tally and the semantic coverage of each of the twenty test suites, then we check to what extent the two measures are compatible with each other. We find that there is little concordance between the rankings of test suites by mutation tally and by semantic coverage. We also find that mutation tally has greater correlation with semantic coverage for partial correctness than for total correctness, and that it has greater correlation with semantic coverage when the program has lower failure rates, i.e. higher reliability.

To answer the second question, we argue that failures are more definite observations, hence are more adequate as a basis for formal definitions. Also, we argue that in most contexts, failure rate is more meaningful than fault density, and faults have widely varying impacts on the failure rate. We also observe that different test processes have different goals, hence may be subject to different criteria for test suite effectiveness.

Our prospects for future research involve broadening the scope of our experiments, further exploring the relationship between fault delection and failure exposure, and investgating means to estimate the semantic coverage of a test suite by approximate means, since its precise calculation is complex and impractical.

### Acknowledgements

The authors are very grateful to the anonymous reviewers for their thoughtful, insightful comments, which have greatly improved the content and presentation of this paper. This work is partially supported by NSF grant DGE 2043104.

## References

[1] IEEE Std 7-4.3.2-2003. Ieee standard criteria for digital computers in safety systems of nuclear power generating stations. Technical report, The Institute of Electrical and Electronics Engineers, 2003.

[2] Kalle Aaltonen, Petri Ihantola, and Otto Seppala. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Companion to the 25th Annual ACM SIGPLAN Conference on OOPSLA*, pages 153–160, Reno, NV, 10 2010.

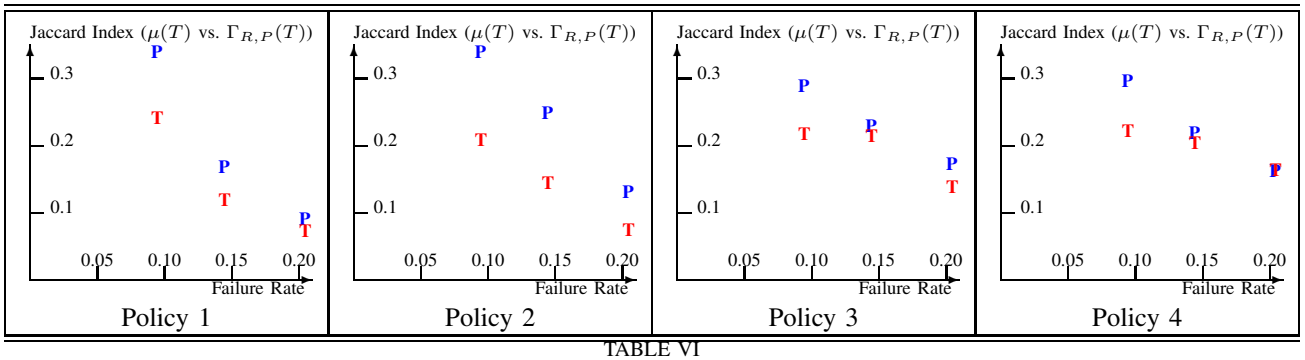



TABLE VI
JACCARD INDEX VS FAILURE RATE


[3] Alireza Aghamohammadi, Seyed-Hassan Mirian-Hosseinabadi, and Sajad Jalali. Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness. *Information and Software Technology*, 129:106426, 2021.

[4] Samia AlBlwi, Amani Ayad, and Ali Mili. A measure of semantic coverage. In *Proceedings, ICSOFT 2023*, Rome, Italy, July 2023.

[5] Samia AlBlwi, Amani Ayad, and Ali Mili. Mutation coverage is not strongly correlated with mutation coverage. In *Proceedings, IEEE Conference on Automated Software Testing*, Lisbon, Portugal, April 2024.

[6] James Andrews, Lionel Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? pages 402–411, 01 2005.

[7] James Andrews, Lionel Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32:608–624, 09 2006.

[8] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[9] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl E Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[10] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[11] R. Banach and M. Poppleton. Retrenchment, refinement and simulation. In *ZB: Formal Specifications and Development in Z and B*, Lecture Notes in Computer Science, pages 304–323. Springer, December 2000.

[12] Chris Brink, Wolfram Kahl, and Gunther Schmidt. *Relational Methods in Computer Science*. Advances in Computer Science. Springer Verlag, Berlin, Germany, 1997.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[14] Xavier Devroey, Gille Perrouin, Maxime Cordy, Mike Papadakis, Axel LeGay, and Pierre Yves Schoebbens. A variability perspective of mutation analysis. In *Proceedings, International Symposium on Foundations of Software Engineering*, pages 841–844, HongKong, China, November 2014.

[15] Nafi Diallo, Wided Ghardallou, and Ali Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering, NIER track*, Firenze, Italy, May 20–22 2015.

[16] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[17] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessment of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), 2006.

[18] Geoffrey Dromey. Program development by inductive stepwise refinement. Technical Report Working Paper 83-11, University of Wollongong, Australia, 1983.

[19] Bouchaib Falah and Soukaina Hamimoune. Mutation testing techniques: A comparative study. 11 2016.

[20] Phyllis Frankl, Stewart Weiss, and Cang Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38, 08 2000.

[21] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–33, 2015.

[22] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. 05 2014.

[23] David Gries. *The Science of Programming*. Springer Verlag, 1981.

[24] Kelly J Hayhurst. *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

[25] E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[26] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, 2015.

[27] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[28] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. 05 2014.

[29] Rene Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings, FSE*, 2014.

[30] Besma Khaireddine and Ali Mili. Quantifying faultiness: What does it mean to have $n$ faults? In *Proceedings, FormaliSE 2021, ICSE 2021 colocated conference*, May 2021.

[31] Besma Khaireddine, Aleksandr Zakharchenko, Matias Martinez, and Ali Mili. Toward a theory of program repair. *Acta Informatica*, 60:209–255, March 2023.

[32] Oded Lachish, Eitan Marcus, Shmuel Ur, and Avi Ziv. Hole analysis for functional coverage data. In *Proceedings of the 39th annual Design Automation Conference*, pages 807–812, 2002.

[33] Jean Claude Laprie. *Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese*. Springer Verlag, Heidelberg, 1991.

[34] Jean Claude Laprie. Dependability —its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–19. Springer Verlag, 1995.

[35] Jean Claude Laprie. Dependable computing: Concepts, challenges, directions. In *Proceedings, COMPSAC*, 2004.

[36] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, pages 220 – 229, 05 2009.

[37] Raghu Lingampally, Atul Gupta, and Pankaj Jalote. A multipurpose code coverage tool for java. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 261b–261b. IEEE, 2007.

[38] R.C. Linger. Cleanroom software engineering for zero-defect software. In *Proceedings, 15th Hawaii International Conference on Software Engineering*, Baltimore, MD, May 1993.

[39] Zohar Manna. *A Mathematical Theory of Computation*. McGraw-Hill, 1974.

[40] Aditya P. Mathur. *Foundations of Software Testing*. Pearson, 2014.

[41] Ali Mili. Differentiators and detectors. *Information Processing Letters*, 169, 2021.

[42] Ali Mili and Fairouz Tchier. *Software Testing: Operations and Concepts*. John Wiley and Sons, 2015.

[43] Harlan D. Mills, Victor R. Basili, John D. Gannon, and Dick R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.

[44] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, 1987.

[45] Carroll C. Morgan. *Programming from Specifications, Second Edition*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[46] A. Namin, J. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings, ICSE 2008*, pages 351–360, 2008.

[47] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings, ISSTA*, 2011.

[48] Roberto Natella, Domenico Cotroneo, Joao Duraes, and Henrique Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Enginering*, 39(1):80–96, January 2011.

[49] J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In *Lecture Notes in Computer Science*, number 4085, pages 236–251. Springer Verlag, 2006.

[50] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*. 2019.

[51] Ali Parsai and Serge Demeyer. Dynamic mutant subsumption analysis using littledarwin. In *Proceedings, A-TEST 2017*, Paderborn, Germany, September 4-5 2017.

[52] Ali Parsai and Serge Demeyer. Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer*, 22:1–24, 08 2020.

[53] Ali Parsai, Alessandro Murgia, and Serge Demeyer. Littledarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. In *FSEN 2017, Foundations of Software Engineering*, 2016.

[54] Donghuan Shin and Doo Hwan Bae. A theoretical framework for understanding mutation-based testing methods. In *Proceedings, ICST 2016*, Chicago, IL, April 2016.

[55] Khashayar Etemadi Someoliayi, Sajad Jalali, Mostafa Mahdieh, and Seyed-Hassan Mirian-Hosseinabadi. Program state coverage: a test coverage metric based on executed program states. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 584–588. IEEE, 2019.

[56] David Tengeri, Laszlo Vidacs, Arpad Beszedes, Judit Jasz, Gergo Balogh, Bela Vancsics, and Tibor Gyimothy. Relating code coverage, mutation score and test suite reducibility to defect density. In *Proceedings, 2016 IEEE 9th International Conference on Software Testing, Verification and Validation Workshops*, pages 174–179, 04 2016.

[57] Q Zhu, A. Panichella, and A. Zaidman. A systematic literature review of how mutation testing supports test activities. *PeeJ Preprint*, (e2483v1):1–57, 2016.