Detecting Faults vs. Exposing Failures: Orthogonal Measures of Test Suite Effectiveness

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—One of the most important decisions we make in software testing is the choice of test data; and one of the most important factors that drives and determines this choice is the way we measure test suite effectiveness. Most measures of test suite effectiveness in use nowadays equate test suite effectiveness with the ability to detect program faults. We argue that there is an alternative way to measure the effectiveness of a test suite, which is to equate its effectiveness by its ability to expose program failures. This raises three immediate questions: First, are these metrics identical? If not, are they statistically correlated? If not, which is a better measure of test suite effectiveness? In this paper, we discuss these questions on the basis of analytical and empirical arguments.

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

1. Measuring Effectiveness

Research in test data selection has been one of the most active areas of software testing research, for two orthogonal reasons:

- Test data selection is one of the most important phases of software testing.
- Test data selection is also one of the hardest steps of software testing: the challenge of representing an infinite input space by a finite and small subset admits only imperfect approximate solutions.

In practice, the selection of test data is driven and determined by the way in which we compare candidate test suites, which is in turn determined by how we measure and quantify test suite effectiveness. It is virtually impossible to do justice to all the research that was conducted in the past in test data selection, but as a broad approximation, we can argue that test suite effectiveness has traditionally be quantified in one of two ways [34]:

- *Syntactic Coverage Metrics*. This class includes such metrics as *statement coverage*, *branch coverage*, *condition coverage*, *line coverage*, *path coverage*, etc [31, 21, 13, 34, 2, 32, 25, 20, 47, 23].
- *Mutation Coverage Metrics*. This class quantifies the effectiveness of a test suite by its mutation score [44, 1, 7, 17, 43, 40, 19, 29, 14, 48].

It is fair to argue that all of these metrics equate the effectiveness of a test suite with its ability to detect faults: Syntactic coverage metrics are based on the assumption that the more syntactic features a test suite exercises, the more likely it is to sensitize a fault; mutation coverage is based on the assumption that mutations are proxies for actual faults [7, 40, 29], hence the ability of a test suite to kill mutants is an indication of its ability to detect faults.

In this paper we submit that there is an alternative way to quantify the effectiveness of a test suite. The alternative we present is not a random arbitrary choice; rather we argue that we arrive to it by adhering to the simple premise that the effectiveness of an artifact can only be defined with respect to an explicit purpose of the artifact, and must reflect the artifact's fitness to the declared purpose. This leads us to raise the question: What is the purpose of a test suite? The answer to this question is very important because it will determine how we quantify test suite effectiveness, and will be meaningful only to the extent that our choice of purpose is sound. We postulate that the purpose of a test suite T is to expose the failures of an incorrect program (if it fails when executed on T) or to give us confidence in the correctness of the program (if it succeeds when executed on T). These two conditions are actually equivalent: The only reason a successful test gives us confidence in the correctness of a program is the assurance that if the program were incorrect, test suite T would have revealed its incorrectness by exposing its failures. Hence we reformulate the purpose of a test suite T as follows:

> **We postulate that the purpose of a test suite T of program P is to expose the failures of P if P is incorrect.**

Our measure of test suite effectiveness (which we will present in section **??**) is based on this definition of the purpose of a test suite, and attempts to reflect the extent to which a test suite fits this purpose.

The existence of two possible criteria for assessing the quality of a test suite raises three questions, which we adopt as this paper's research questions:

- *RQ1: Are these two criteria equivalent*? Is a test suite's ability to detect faults commensurate with its ability to expose failures?
- *RQ2: Are these two criteria statistically correlated*? If T' is better than T at detecting faults, how likely is it to be also better at exposing failures (and vice-versa)?
- *RQ3: If the answer to both RQ1 and RQ2 is negative, which criterion is better*?

Question RQ1 is addressed in section 2 on the basis of analytical arguments. Question RQ2 is addressed in section 6 on the basis of two separate, yet concordant, empirical studies. Question RQ3 is addressed, with all the requisite caution and circumspection due to its sensitivity, in section 7. In section 4 we introduce a measure of test suite effectiveness, called *semantic coverage*, which is intended to reflect the ability of the test suite to expose the failures of a program (if it is incorrect). For the ability of a test suite to detect faults, we adopt the concept of *mutation tally*, which is the set of mutants killed in a given mutation experiment; this is the subject of section 5. The metrics adopted in sections 4 and 5 are used to run the empirical study of section 6. In section 8 we summarize our findings, critique them, and sketch directions of future research.

2. RQ1: Detecting Faults vs. Exposing Failures

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*. An error causes a failure if it propagates all the way to the final state of the program's execution.

The contrast between failures and faults can be characterized as a contrast between observable, certifiable effects (the failures) and hypothetical speculative causes of these effects (the faults).

Indeed, whether execution of a program under test on some test datum is successful or not can be definitely determined by the test oracle. The question of what fault may have caused the observed failure is typically a much more debatable / controversial question, as the same failure may be attributed to several possible faults or combinations of faults, and may be repaired, accordingly, in several different ways. Hence, a priori, the relationship between failures and faults appears to be very tenuous.

Another reason for the disconnect between failures and faults is that, from the standpoint of failures, faults are not created equal: it is easy to imagine why some faults may cause failures far more often than others; 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 (the frequency of failures) as a fault that is part of routine code that gets invoked at each call. In [37] 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%.

The variability in the impact of faults on failures can be observed easily in mutation testing, where some mutants are easily killed whereas others can resist extensive testing [28, 49]: a mutation that occurs in dead code cannot be detected at all, and a mutation that arises in an execution path whose path condition is very stringent can only be detected for very few inputs. We conjecture that to the extent that mutations are proxies for faults [7, 40, 29], stubborn mutations are proxies for low-impact faults, i.e. faults that seldom cause failures.

3. Mathematical Background

As a prerequisite for defining and analyzing *semantic coverage*, the metric we use to quantify a test suite's ability to expose program failures, we must introduce a number of concepts pertaining to program correctness; these, in turn, require some mathematical background, which is focus of this section.

3.1 Relational Notations

In this paper, we use relations and functions [12] to capture program specifications and program semantics. We can model the semantics 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 (or a function P) by $dom(R)$ ($dom(P)$). Because we model programs and specifications by homogeneous relations / functions, we talk about initial states and final states, though we may sometimes, talk about inputs and outputs, when this terminology is better at conveying our ideas.

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

3.2 Specifications and Programs

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 [24, 39, 11, 18, 42, 10].

3.3 Partial and Total Correctness

The distinction between partial correctness and total correctness has long been a feature of the study of correctness verification [16, 22, 33], 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 [38]. 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 [36]. Program P on space S is said to be *partially correct* with respect to specification R on S if and only if:

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

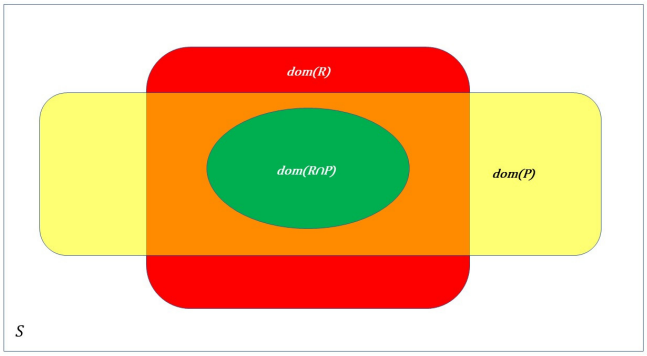


Figure 1. Detector Sets for Partial Correctness (orange) and Total Correctness (red+orange)

Khaireddine et al. show in [30] that these definitions are equivalent, modulo differences of notation, to traditional definitions of total and partial correctness [33, 16, 26, 22].

3.4 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 detector sets.

Definition 4: Due to [35]. 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. See Figure 1. 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 $\Theta(R, P)$.

	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, DUE TO [5]

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

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.

Of course a program is (partially or totally) correct if and only if no initial state disproves its (partial or total) correctness. 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.

4. SEMANTIC COVERAGE: FAILURE-BASED EFFECTIVENESS

In order to investigate the relationship between the ability to detect faults and the ability to reveal failures, we mus 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.

4.1 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; 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 [3]. 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 correctness 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).

- $\Gamma_{P,R}(T) = \emptyset$. 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. This is clearly the attribute of an ineffective test suite.
- $\Gamma_{P,R}(T) = S$. 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

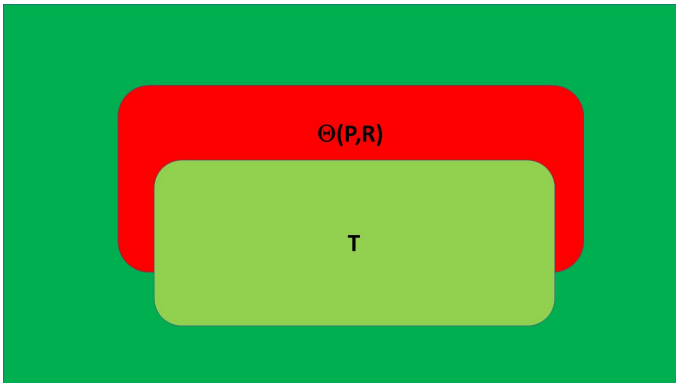


Figure 2. Semantic Coverage of Test T for Program P with respect to R (shades of green)

expose 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).

See 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. The semantic coverage of T is the union of the area that need not be tested (dark green) with the area that is actually tested (light green). Test suite T is as good as the red area is small.

4.2 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 actually meet the declared criteria. We start by 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 (or equal) 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, i.e. a stronger oracle). 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 [3], 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 total 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 than P with respect to R then:

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

Proposition 7: Monotonicity with respect to relative partial 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-partially-correct than P with respect to R then:

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

Proposition 8: 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:

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

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

4.3 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, i.e. fewer elements for T to cover.
- $\mathit{dom}(R \cap P)$: A larger competence domain means fewer failures to reveal.

Likewise, a decomposition 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 easier it is to test.

5. Mutation Tally: Fault-Based Effectiveness

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 [41, 40, 7, 8, 29, 6, 27], 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 which version of mutation coverage we envision to use:

- *Concern for Equivalence*. In its raw form, the mutation score of a test suite T is the ratio of killed mutants over the total number of generated mutants. The same raw mutation score means vastly different things depending on whether the surviving mutants are equivalent to the base program (hence no test suite can kill them) or not. Hence we do not use the raw mutation score.
- *Concern for Redundancy*. To address the concern for equivalence, we may decide to quantify the mutation coverage of a test suite by the ratio of killed mutants over the number of killable (i.e. non-equivalent) mutants, assuming we have a way to identify those; we call this the *prorated mutation score (PMS)*. The same prorated mutation score may mean vastly different things depending on whether the killed mutants are semantically distinct from each other, all semantically equivalent to each other, or partitioned into large equivalence classes; hence we do not use the prorated mutation score.
- *Concern for Precision*. To address the concern for redundancy, we may decide to quantify the mutation coverage of a test suite by considering equivalence classes of mutants (modulo semantic equivalence) rather than individual mutants; hence we define the *equivalence-based mutation score (EMS)* of a test suite T as the ratio of equivalence classes that are killed by T over the total number of equivalence classes of the killable mutants, assuming that we have a way to tell whether two mutants are semantically equivalent. The property of whether a test suite T is better than another test suite T' is not a total ordering relation, but rather a partial ordering relations: it is not always possible to compare two random test suites for effectiveness; they may be killing different sets of mutants, or disjoint sets of mutants. Yet, if we assign them numeric scores, we can always compare them, since any two numbers can be compared. Whenever we represent a partial ordering (re: the property of being a more effective test suite) by a total ordering (re: the property of having a higher EMS), we create a built-in loss of precision; hence we do not use EMS as our measure of test suite effectiveness.

Ultimately, the only way to assert with confidence that some test suite T is more effective than some test suite T' in the context of a mutation experiment is if all the mutants killed by T' are killed by T. Whence we resolve to quantify the effectiveness of a test suite to detect faults by the set of mutants that the test suite T kills, which we call the *mutation tally* of T, and we denote by $\mu(T)$. Then, we consider that T is more effective than T' if and only if:

$$\mu(T') \subseteq \mu(T).$$

The fact that we are now comparing two partial orderings (the partial ordering defined by semantic coverage, and that which is defined by mutation tally) gives further validity to our empirical study.

6. RQ2: Empirical Study

To answer question RQ2, we conduct two experiments, involving two benchmark programs and their associated test data.

6.1 A Wide Range of Variability

In this first experiment we choose twenty test suites (T_1, T_2, ... T_{20}) whose sizes vary over a wide range; hence the size of each test suite may play an important role in determining its effectiveness.

6.1.1 The Benchmark Program and Tests

We consider the following experimental data:

- *Sample Program, P*. The sample program that we use for this experiment is a method called `createNumber()` of

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

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 0.2×107 and 0.77×107. Table II shows the sizes of the test suites derived from this algorithm.

6.1.2 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.

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

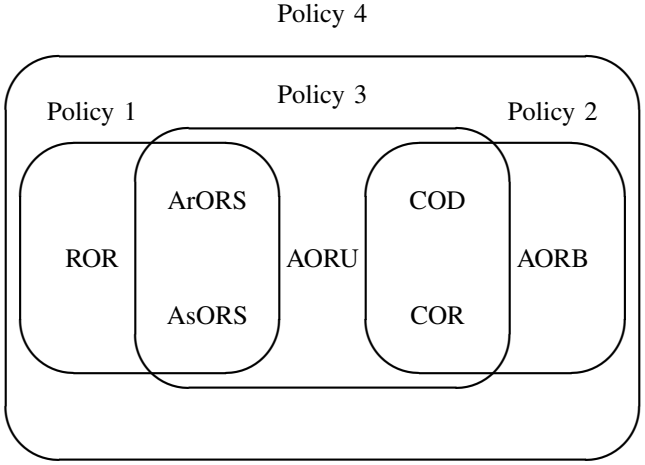


Figure 3. Mutant Generation Policies

6.1.3 The Mutation Tally

We use the mutant generator *LittleDarwin* [45, 46], 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.

6.1.4 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 analyze inclusion relations between them ($\Gamma_{P,R_n}(T_i) \subset \Gamma_{P,R_n}(T_j)$). For each specification and standard of correctness, this yields a graph on the nodes that 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

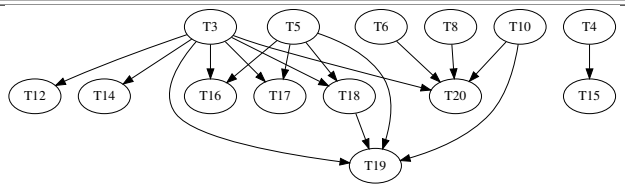
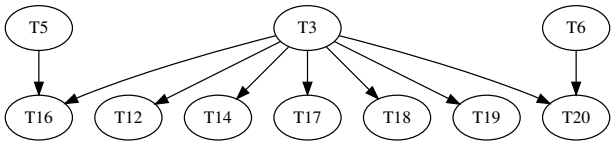
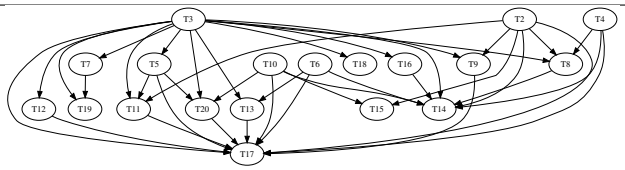
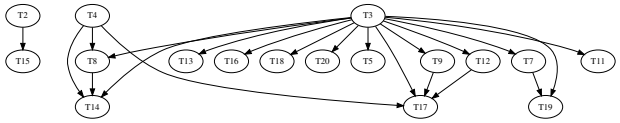
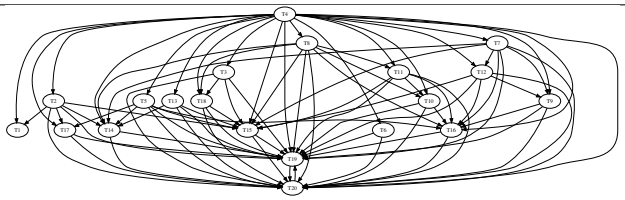
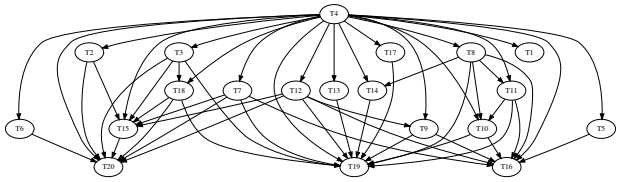
Spec	Partial Correctness	Total Correctness
R_1		
R_2		
R_3		

TABLE III
ORDERING TEST SUITES BY SEMANTIC COVERAGE

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 0.2×107 to 0.77×107 (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 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.

6.1.5 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 suites 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 [4] 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).

6.1.6 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. 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 the relationships between mutation tally and semantic coverage.

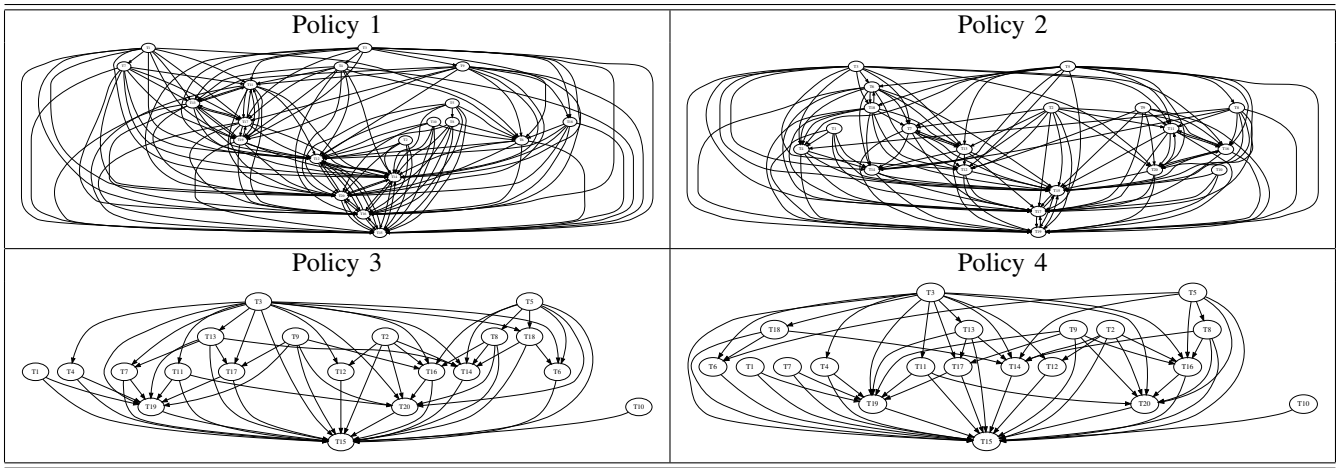


TABLE IV
ORDERING TEST SUITES BY MUTATION TALLY

	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 SEMANTIC COVERAGE AND MUTATION TALLY

6.1.7 Mutation Tally vs Mutation Tally

If we focus on the top left triangle of Table V, we find that the Jaccard index between 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 [4].

6.1.8 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. Tese 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.

6.1.9 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

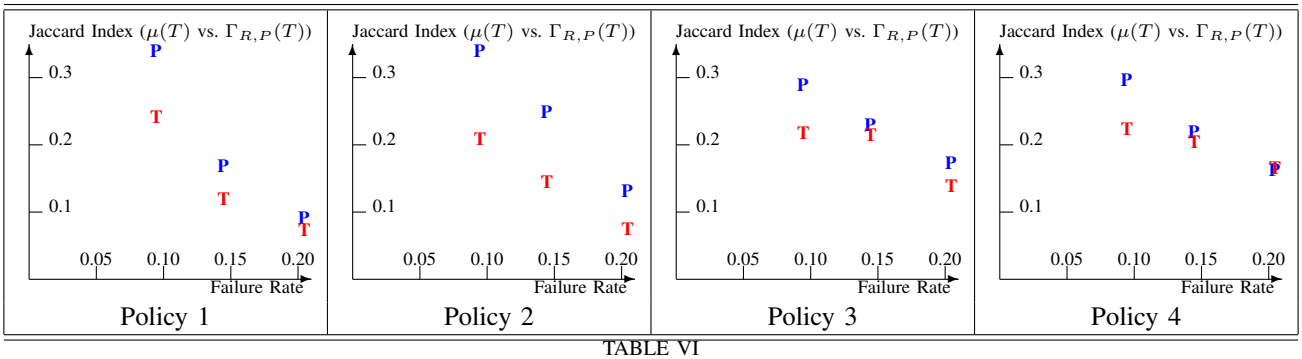


TABLE VI
JACCARD INDEX VS FAILURE RATE

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.

6.2 Second Experiment: A Narrow Range of Variability

For our second experiment, we chose to vary the size of test suites from 30% of the size of T_0 to 70% of the its size. Our goal in this experiment is to consider a situation where the effectiveness of each test suite is determined less by its size and more by its composition (the elements that it has).

- *Sample Program*. The sample program that we use for this experiment is a method called `expm1(double x, double[] hiPrecOut)` of the Java class `FastMath.java`, from the commons benchmark `commons-math3-3.6.1-src` . The size of the selected method is 141 LOC.
- *Base Test Suite*. We consider the test class that comes with the selected program: `class FastMathTest.java`. This class includes 1000 tests. Because this class is too large, we select a subset of it of size 200.
- *Sample Test Suites*. We select twenty subsets of T_0, drawn at random from size 60 (approximately, since they are drawn at random with varying threshold, starting at 30% of the size of T_0) to size 140 (approximately, at 70% of the size of T_0). We name these T_1, T_2, ... T_{20}; their sizes are given in Table VII.
- *Specifications*. To generate sample specifications, we take the base program and run it on T_0, which gives us 200 input/ output pairs. In order to distinguish between partial correctness and total correctness, we have to cause the base program to diverge for some inputs; to this effect, we change the first nine elements (t_0, t_1, ... t_8) of T_0 in such a way that the base program diverges. Then we select three specifications, R_1, R_2, R_3, as follows:
  - R_1: Specification R_1 covers inputs t_2 through t_6, to which it assigns arbitrary values, and inputs t_9 through t_{199}, which differ from the base program once every 11 elements.
  - R_2: Specification R_2 covers inputs t_0 through t_3, to which it assigns arbitrary values, and inputs t_9 through t_{199}, which differ from the base program once every 13 elements.
  - R_3: Specification R_3 covers inputs t_5 through t_8, to which it assigns arbitrary values, and inputs t_9 through t_{199}, which differ from the base program once every 17 elements.
- *Mutation Policies*. To generate mutants for the base program, we have adopted three policies, defined by the following LittleDarwin operators:
  - *Policy 1*: Operators AORU, ROR, and COR. This generates 31 mutants.
  - *Policy 2*: Operator AORB. This generates 87 mutants.
  - *Policy 3*: Operators AORU, ROR, COR, AORB. This generates 131 mutants.

The graphs of Table VIII show the partial ordering of the test suites by semantic coverage for partial and total correctness with respect to specifications R_1, R_2 and R_3; when an arrow points from test suite T_i to test suite T_j, it means T_j has higher semantic coverage than T_i; not surprisingly, nodes with higher indices tend to have higher semantic coverage —but of course this is not systematic. For all three specifications, the graph of total correctness appears to be a subgraph of the graph for partial correctness. Across specifications, the graphs appear to be very different, although they appear to be relatively concordant in terms of the maximal nodes (which appear at the bottom of the graph).

The graphs of mutation tally are shown in Table IX; the graphs for policy 2 and policy 3 look similar because these policies have a large number of mutants in common. As in the graphs of Table VIII, test suites with higher indices tend to have higher mutation tally, because they tend to be larger —yet here again this is not systematic.

Table X shows the Jaccard index of all the pairs of graphs for semantic coverage and mutation tally; the top right rectangle of this Table, which reflects the similarity between the ordering relations defined by mutation tally and semantic coverage, shows very small values throughout. Most values of the Jaccard index fall below 0.1, meaning that less than 10% of the arcs are common between the two graphs being considered. The only exception is the graph of semantic coverage for partial correctness with respect to R_3, which

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}
200	62	58	79	80	80	88	100	102	95	109	109	111	119	119	109	126	130	131	145	143

TABLE VII
SIZES OF THE RANDOMLY GENERATED TEST SUITES, SECOND EXPERIMENT

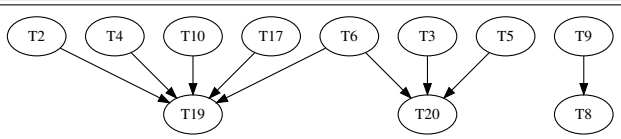
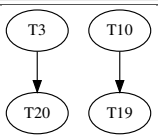
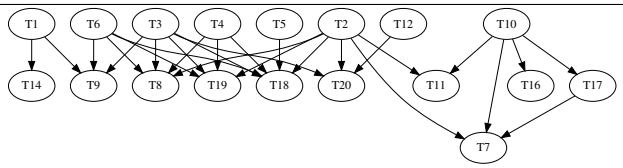
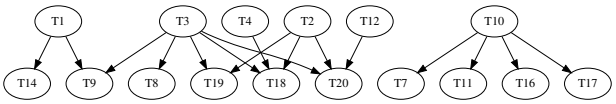
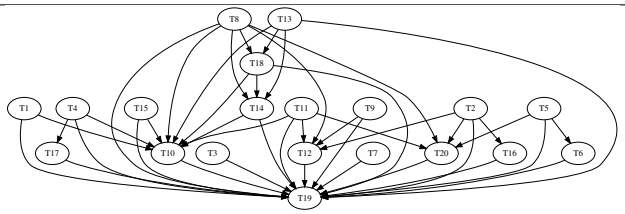
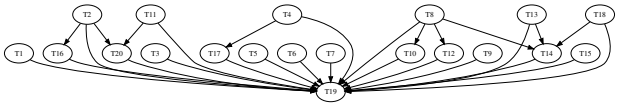
Spec	Partial Correctness	Total Correctness
R_1		
R_2		
R_3		

TABLE VIII
ORDERING TEST SUITES BY SEMANTIC COVERAGE: SECOND EXPERIMENT

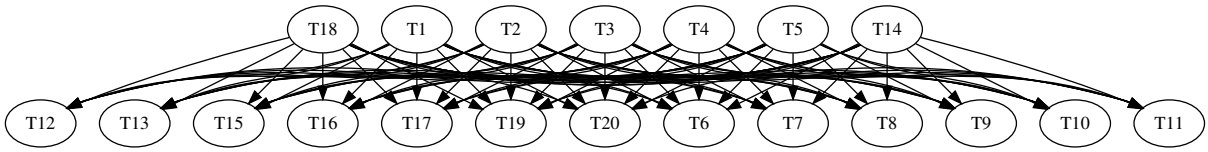
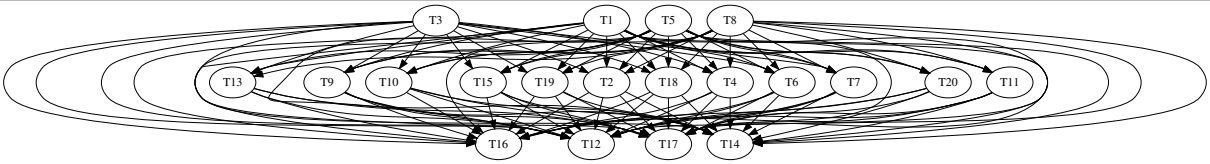
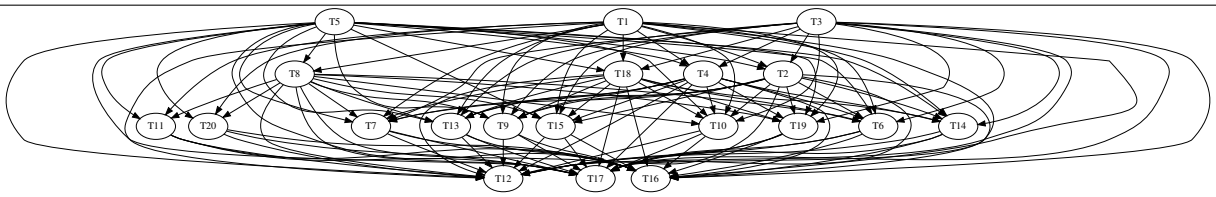
Policy 1	
Policy 2	
Policy 3	

TABLE IX
ORDERING TEST SUITES BY MUTATION TALLY, SECOND EXPERIMENT

	Policy 1	Policy 2	Policy 3	PR1	TR1	PR2	TR2	PR3	TR3
Policy 1	1.0000	0.2721	0.4759	0.0217	0.0109	0.0404	0.0312	0.1453	0.0917
Policy 2		1.0000	0.7132	0.0180	0.0090	0.0517	0.0351	0.1418	0.0866
Policy 3			1.0000	0.0161	0.0081	0.0465	0.0314	0.1448	0.0863
PR1				1.0000	0.3333	0.0714	0.0000	0.0222	0.0333
TR1					1.0000	0.0000	0.0000	0.0000	0.0000
PR2						1.0000	0.6666	0.0577	0.0526
TR2							1.0000	0.0200	0.0000
PR3								1.0000	0.6511
TR3									1.0000

TABLE X
JACCARD INDEX OF SEMANTIC COVERAGE VS. MUTATION TALLY, SECOND EXPERIMENT

shows values between 0.14 and 0.15; interestingly, and perhaps not fortuitously, R_3 is the specification which the base program violates the least (for the fewest inputs). This seems to confirm the observation made in Table VI about the first experiment.

7. RQ3: COMPARATIVE EFFECTIVENESS

Two simple premises support the conjecture that exposing failures is a better attribute of test suite effectiveness than detecting faults:

- *From the standpoint of a user, failure frequency is more consequential than fault density*. An end user cares little how many faults a program has, provided it fails seldom. Yet failure frequency and fault density can be vastly unrelated: a program may have a single fault that causes it to fail at each execution; or it may have ten faults that seldom cause it to fail, if they are located in obscure code that seldom gets executed at all.
  In fairness, we acknowledge that not all testing is based on the user's viewpoint: in unit testing, for example, the focus is on finding and removing faults, hence it is sensible to select test suite that maximize the ability to detect faults. But even then, focusing on failures leads us naturally to higher impact faults, since these are more likely to cause failure; even in unit testing, it is important to target high impact faults before low impact faults.
- *Whereas failures are observable, certifiable effects, faults are speculative, hypothetical causes of the observed effects*. In [9], Avizienis et al. define a *fault* as the *adjudged or hypothesized cause of an error*. More broadly, a fault is usually someone's assumption as to the cause of an observed failure; the same failure may be attributed to more than one fault or combination of faults, and may be remedied in more than one way.
  Of course, it is better to anchor our definition of test suite effectiveness in observable effects rather than in speculative causes.

Given that virtually all metrics that are in use nowadays equate test suite effectiveness with the ability to detect faults, these premises raise the uncomfortable question of whether we have been selecting test suites on the basis of imperfect criteria. Also, they raise the question of how to quantify the semantic coverage of a test suite in practice.

For the purposes of this study, we computed the effectiveness of test suites using elaborate calculations involving the function of the program, a formal specification, and a standard of correctness; but this is clearly impractical in general, as it requires data that is usually unavailable and calculations that are usually prohibitively complex. Hence in order to make semantic coverage useful in practice, we have to find practical means to approximate it; this is part of our plan for future research. The first resolution we make in this plan is the recognition that semantic coverage does not define a total ordering between test suites, but rather a partial ordering. Hence we resolve to represent and approximate not by a single numeric metric, but rather by a vector of metrics, which reflects the test suite, the program under test, the specification against which correctness is tested, and possibly the standard of correctness that is considered. This matter is currently under investigation.

8. CONCLUSION

The most important step in software testing is the selection of test suites; and the most important factor in this selection is the way in which we assess the effectiveness of a test suite. In this paper, we argue that there are two, possibly orthogonal, means to define the effectiveness of a test suite:

- Either by equation test suite effectiveness with its ability to detect program faults.
- Or by equating test suite effectiveness with its ability to expose program failures.

This dichotomy naturally raise a number of questions, which we use as the research questions of our paper:

- *RQ1: Are these criteria equivalent?*
- *RQ2: Are these criteria statistically correlated?*
- *RQ3: Which of these criteria is better to select test data?*

In light of our analytical and empirical arguments discussed in this paper, we are fairly confident that the answers to questions RQ1 and RQ2 is negative. Indeed, our two experiments, involving different benchmark programs (`createNumber()` vs `expm1()`), of different sizes (170 LOC vs. 141 LOC), using different size test suites (107 tests vs 200 tests), ranging

over different gaps in size (20% to 80% vs 30% to 70%), using different mutant generation policies, and specifications with different failure rates (failure frequencies of 5, 7, 11 vs failure frequencies of 11, 13, 17), still return largely concordant results: very little similarity between the ordering of the sample test suites by mutation tally (a proxy for effectiveness to detect faults) and semantic coverage (a proxy for effectiveness to expose failures).

In reference to RQ3, we argue that it is better to quantify test suite effectiveness on the basis of failures rather than faults, because it is better to anchor our definition of effectiveness in observable effects rather than speculative causes; but we are mindful that this is a precarious position, as most metrics of test suite effectiveness in use nowadays are based on fault detection.

Another issue with our metric is, of course, that it is not a number, and it is not easy to compute in practice, unlike traditional coverage metrics, which can be computed automatically using widely available software tools. We are currently conducting empirical studies to see whether semantic coverage can be approximated by a vector of numeric metrics; our goal is to imitate the partial ordering of semantic coverage by the partial ordering defined by a vector of, say, two or more numeric metrics. The challenge is to find a vector that best approximates semantic coverage; this matter is under investigation.

References

[1] 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*. Reno, NV, Oct. 2010, pp. 153–160. DOI: 10.1145/1869542.1869567.

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

[3] Samia AlBlwi, Amani Ayad, and Ali Mili. "A Measure of Semantic Coverage". In: *Proceedings, ICSOFT 2023*. Rome, Italy, July 2023.

[4] 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, Apr. 2024.

[5] Samia AlBlwi et al. "Subsumption, Correctness and Relative Correctness: implications for Software Testing". In: *Science of Computer Programming* 239 (2025).

[6] James Andrews, Lionel Briand, and Yvan Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In: Jan. 2005, pp. 402–411. DOI: 10.1145/1062455.1062530.

[7] James Andrews et al. "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria". In: *Software Engineering, IEEE Transactions on* 32 (Sept. 2006), pp. 608–624. DOI: 10.1109/TSE.2006.83.

[8] James H Andrews et al. "Using mutation analysis for assessing and comparing testing coverage criteria". In: *IEEE Transactions on Software Engineering* 32.8 (2006), pp. 608–624.

[9] Algirdas Avizienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[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. Springer, Dec. 2000, pp. 304–323. DOI: 10.1007/3-540-44525-0_18.

[12] Chris Brink, Wolfram Kahl, and Gunther Schmidt. *Relational Methods in Computer Science*. Advances in Computer Science. Berlin, Germany: Springer Verlag, 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*. Vol. 8. 2008, pp. 209–224.

[14] Xavier Devroey et al. "A Variability Perspective of Mutation Analysis". In: *Proceedings, International Symposium on Foundations of Software Engineering*. HongKong, China, Nov. 2014, pp. 841–844.

[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 2015. DOI: 10.1109/ICSE.2015.200.

[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". In: *IEEE Transactions on Software Engineering* 32.9 (2006).

[18] Geoffrey Dromey. *Program Development by Inductive Stepwise Refinement*. Tech. rep. Working Paper 83-11. University of Wollongong, Australia, 1983. DOI: 10.1002/spe.4380150102.

[19] Bouchaib Falah and Soukaina Hamimoune. "Mutation Testing Techniques: A Comparative Study". In: Nov. 2016. DOI: 10.1109/ICEMIS.2016.7745368.

[20] Milos Gligoric et al. "Guidelines for coverage-based comparisons of non-adequate test suites". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.4 (2015), pp. 1–33.

[21] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code coverage for suite evaluation by developers". In:

May 2014. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568278.

[22] David Gries. *The Science of Programming*. Springer Verlag, 1981. DOI: 10.1007/978-1-4612-5983-1.

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

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

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

[26] C.A.R. Hoare. "An axiomatic basis for Computer programming". In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–583. DOI: 10.1145/363235.363259.

[27] Laura Inozemtseva and Reid Holmes. "Coverage is not strongly correlated with test suite effectiveness". In: (May 2014). DOI: 10.1145/2568225.2568271.

[28] Yue Jia and Mark Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *Proceedings, Eighth IEEE International Working Conference on Software Code Analysis and Manipulation*. Beijing, China, Sept. 2008, pp. 249–258.

[29] Rene Just et al. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: *Proceedings, FSE*. 2014.

[30] Besma Khaireddine et al. "Toward a Theory of Program Repair". In: *Acta Informatica* 60 (Mar. 2023), pp. 209–255.

[31] Oded Lachish et al. "Hole analysis for functional coverage data". In: *Proceedings of the 39th annual Design Automation Conference*. 2002, pp. 807–812.

[32] 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)*. IEEE. 2007, 261b–261b.

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

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

[35] Ali Mili. "Differentiators and Detectors". In: *Information Processing Letters* 169 (2021).

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

[37] H.D. Mills, M. Dyer, and R.C. Linger. "Cleanroom Software Engineering". In: *IEEE Software* 4.5 (1987), pp. 19–25.

[38] Harlan D. Mills et al. *Structured Programming: A Mathematical Approach*. Boston, Ma: Allyn and Bacon, 1986.

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

[40] A. Namin, J. Andrews, and D. Murdoch. "Sufficient Mutation Operators for Measuring Test Effectiveness". In: *Proceedings, ICSE 2008*. 2008, pp. 351–360.

[41] Roberto Natella et al. "On Fault Representativeness of Software Fault Injection". In: *IEEE Transactions on Software Enginering* 39.1 (Jan. 2011), pp. 80–96. DOI: 10.1109/TSE.2011.124.

[42] J.N. Oliveira and C.J. Rodrigues. "Pointfree Factorization of Operation Refinement". In: *Lecture Notes in Computer Science*. 4085. Springer Verlag, 2006, pp. 236–251.

[43] Mike Papadakis et al. "Mutation Testing Advances: An Analysis and Survey". In: *Advances in Computers*. 2019.

[44] Ali Parsai and Serge Demeyer. "Comparing mutation coverage against branch coverage in an industrial setting". In: *International Journal on Software Tools for Technology Transfer* 22 (Aug. 2020), pp. 1–24. DOI: 10.1007/s10009-020-00567-y.

[45] Ali Parsai and Serge Demeyer. "Dynamic Mutant Subsumption Analysis Using LittleDarwin". In: *Proceedings, A-TEST 2017*. Paderborn, Germany, Sept. 2017.

[46] 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.

[47] Khashayar Etemadi Someoliayi et al. "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)*. IEEE. 2019, pp. 584–588.

[48] David Tengeri et al. "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*. Apr. 2016, pp. 174–179. DOI: 10.1109/ICSTW.2016.25.

[49] Xiangjuan Yao, Mark Harman, and Yue Jia. "A Study of Equivalent and Stubborn Mutation Operators using Human Analysis of Equivalence". In: *Proceedings, ICSE*. 2014.