

Semantic Coverage: Measuring Test Suite Effectiveness

Samia Al Blwi¹, Amani Ayad², Bisma Khairredine³, Imen Marsit⁴ and Ali Mili¹ ^a

¹*NJIT, Newark NJ, U.S.A.*

²*Kean University, Union NJ, U.S.A.*

³*University of Tunis El Manar, Tunis, Tunisia*

⁴*University of Sousse, Sousse, Tunisia*

Keywords: Software Testing, Test Suite Effectiveness, Syntactic Coverage, Mutation Coverage, Semantic Coverage.

Abstract: Several syntactic measures have been defined in the past to assess the effectiveness of a test suite: statement coverage, condition coverage, branch coverage, path coverage, etc. There is ample analytical and empirical evidence to the effect that these are imperfect measures: exercising all of a program's syntactic features is neither necessary nor sufficient to ensure test suite adequacy; not to mention that it may be impossible to exercise all the syntactic features of a program (re: unreachable code). Mutation scores are often used as reliable measures of test suite effectiveness, but they have issues of their own: some mutants may survive because they are equivalent to the base program not because the test suite is inadequate; the same mutation score may mean vastly different things depending on whether the killed mutants are distinct from each other or equivalent; the same test suite and the same program may yield different mutation scores depending on the mutation operators that we use. Fundamentally, whether a test suite T is adequate for a program P depends on the semantics of the program, the specification that the program is tested against, and the property of correctness that the program is tested for (total correctness, partial correctness). In this paper we present a formula for the effectiveness of a test suite T which depends exactly on the semantics of P , the correctness property that we are testing P for, and the specification against which this correctness property is tested; it does not depend on the syntax of P , nor on any mutation experiment we may run. We refer to this formula as the *semantic coverage* of the test suite, and we investigate its properties.

1 ON THE EFFECTIVENESS OF A TEST SUITE


1.1 Motivation

In this paper we envision to define a measure to quantify the effectiveness of a test suite. The effectiveness of an artifact can only be defined with respect to the purpose of the artifact, and must reflect its fitness for the declared purpose. If the purpose of a test suite is to reveal the presence of faults in incorrect programs, then it is sensible to quantify the effectiveness of a test suite by its ability to reveal faults. A necessary condition to reveal a fault is to exercise the code that contains the fault; hence many metrics of test suite effectiveness focus on the ability of a test suite to exercise syntactic attributes of the program (Mathur, 2014); but while achieving syntactic coverage is necessary, it

is far from sufficient, and not always possible. Indeed not all faults cause errors and not all errors lead to observable failures (Avizienis et al., 2004); also, it is not always possible to exercise all syntactic features of a program (re: infeasible paths, dead code), so that it is possible to thoroughly test a program without covering all its statements (if the code that has not been exercised contains no faults).

A better measure of test suite effectiveness is mutation coverage, which is defined as the ratio of mutants that it kills out of a set of generated mutants. But while mutation coverage is often used as the gold standard of test suite effectiveness (Inozemtseva and Holmes, 2014; Andrews et al., 2006), it has issues of its own:

- The same mutation score may mean vastly different things depending on whether the killed mutants are all distinct from each other, all equivalent, or partitioned into some equivalence classes.
- The same test suite T may yield different mutation

^a  <https://orcid.org/0000-0002-6578-5510>

scores for different sets of mutants, hence the mutation score cannot be considered as an intrinsic attribute of the test suite.

- Even assuming that mutants are a faithful proxy for actual faults (Just et al., 2014), we argue that assessing the effectiveness of test suites by their mutation score may be imperfect, because of the disconnect between fault density and failure rate (Farooq et al., 2012).

In this paper we present a measure of test suite effectiveness which depends only on the program under test, the correctness property we are testing it for, and the specification against which correctness is defined; also, this measure is intended to reflect a test suite's effectiveness to expose failures, rather than to detect faults. In the next section we present and discuss some criteria that a measure of test suite effectiveness ought to satisfy, and in section 1.3 we present and justify some design principles that we resolve to adopt to this effect.

In section 2 we introduce detector sets, and discuss their significance for the purposes of program testing and program correctness, and in section 3 we use detector sets to introduce our definition of test suite effectiveness; we validate our proposed definition in section 4 by showing, analytically, that it meets all the requirements set forth in section 1.2. In section 5 we illustrate the derivation of semantic coverage on a sample benchmark example, and show its empirical relationship to mutation scores. We conclude in section 6 by summarizing our findings, critiquing them, comparing them to related work, and sketching directions of further research.

1.2 Requirements of Semantic Coverage

We consider a program P that we want to test for correctness against a specification R and we wish to assess the fitness of a test suite T for this purpose. We argue that the effectiveness of test suite T to achieve the purpose of the test ought to be defined as a function of three parameters:

- Program P .
- Specification R .
- The standard of correctness that we are testing P for: partial correctness or total correctness (Hehner, 1992).

The requirements we present below dictate how semantic coverage ought to vary as a function of these parameters.

- Rq1. *Monotonicity with respect to test suite size.* Notwithstanding that we favor smaller test

suites for the sake of efficiency, we argue that from the standpoint of effectiveness, larger test suites are better: if T' is a superset of T then T' ought to have higher semantic coverage than T .

- Rq2. *Monotonicity with respect to relative correctness.* Relative correctness is the property of a program to be more-correct than another with respect to a specification (Diallo et al., 2015). A test suite T ought to have increasingly greater semantic coverage as the program grows more-correct, since more-correct programs have fewer failures to reveal.

- Rq3. *Monotonicity with respect to refinement.* Specifications are naturally ordered by refinement, whereby more-refined specifications represent stronger/ harder to satisfy requirements (Morgan, 1998; Hehner, 1992); a given program P fails more often against a more-refined specification than a less-refined specification. Hence the same test suite ought to have greater semantic coverage for less-refined specifications.

- Rq4. *Monotonicity with respect to the standard of correctness.* Total correctness is a stronger property than partial correctness, hence the same program will fail the test of total correctness more often than it fails the test of partial correctness. The same test suite T ought to have greater semantic coverage if it is applied to partial correctness than if applied to total correctness.

In section 4 we prove that the formula of semantic coverage presented in section 3 satisfies all the requirements (Rq1-Rq4) discussed in this section.

1.3 Design Principles

We resolve to adopt the following design principles as we define semantic coverage:

- *Focus on Failure.* We adopt the definitions of fault, error and failure proposed by Avizienis et al (Avizienis et al., 2004). A failure is an observable, verifiable, certifiable effect. By contrast, a fault (referred to in (Avizienis et al., 2004) as *the adjudged or hypothesized cause of an error*) is a hypothetical cause of the observed failure; the same failure may be caused by more than one fault or combination of faults. Hence whereas a failure is an objectively verifiable effect, a fault is a speculative hypothesized cause; our definition is based on failures rather than faults.
- *Partial Ordering.* It is easy to imagine two test suites whose effectiveness cannot be ranked: for

example, they reveal disjoint or distinct sets of failures. Hence test suite effectiveness is essentially a partial ordering. Yet if we quantify test suite effectiveness by numbers, we introduce an artificial total ordering, on what is actually a partially ordered set. Hence we resolve to define semantic coverage, not as a number, but as an element of a partially ordered set.

- *Analytical Validation.* We resolve to validate our proposed measure of effectiveness by means of analytical (vs empirical) methods because we do not know of a widely accepted ground truth of test suite effectiveness against which we can empirically validate our definition. Hence we resolve to validate our definition by arguing that it captures the right attributes and that it meets all the requirements that we mandate in section 1.2.

In section 5 we compute the semantic coverage of a set of (20) test suites of a benchmark program for two specifications and two standards of correctness, and we compare the four graphs so derived against two graphs that rank these test suites by mutation coverage, for two mutant generators.

1.4 Relational Mathematics

We assume the reader is familiar with elementary discrete mathematics; in this section, we present some definitions and notations that we use throughout the paper. We define sets by means of C-like variable declarations; if we declare a set S as:

$xType\ x; yType\ y;$

then we mean S to be the cartesian product of the sets of values represented by types $xType$ and $yType$. Elements of S are denoted by s , and have the form $s = \langle x, y \rangle$. The cartesian components of an element of S are usually decorated the same way as the element, so we write for example $s' = \langle x', y' \rangle$.

A relation on set S is a subset of the Cartesian product $S \times S$. Special relations on S include the *identity* relation ($I = \{(s, s) | s \in S\}$), the *universal* relation ($L = S \times S$) and the *empty* relation ($\phi = \{\}$). Operations on relations include the set theoretic operations of union, intersection and complement; they also include the *domain* of a relation, denoted by $dom(R)$ and defined by: $dom(R) = \{s | \exists s' : (s, s') \in R\}$. The *product* of two relations R and R' is denoted by $R \circ R'$ (or RR' for short) and defined by $RR' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. Note that given a relation R , the product of R by the universal relation L yields the relation $RL = dom(R) \times S$; we use RL as a representation of the domain of R in relational form. The *inverse* of relation R is the relation denoted by \hat{R} and defined by $\hat{R} = \{(s, s') | (s', s) \in R\}$. The *restriction* of

relation R to subset T is the relation denoted by $T \setminus R$ and defined by $T \setminus R = \{(s, s') | s \in T \wedge (s, s') \in R\}$.

2 CORRECTNESS AND DETECTOR SETS

Absolute correctness is the property of a program to be (partially or totally) correct with respect to a specification. Relative correctness is the property of a program to be more (partially or totally) correct than another with respect to a specification. The detector set of a program with respect to a specification is the set of inputs (tests) that expose the (partial or total) incorrectness of a program with respect to a specification. In this section, we will show how (partial, total) detector sets enable us to define absolute and relative correctness in simple, uniform terms.

2.1 Specification Refinement

In this paper, we represent specifications by relations and programs by deterministic relations (functions). An important concept in any programming calculus is the property of *refinement*, which ranks specifications according to the stringency of the requirements that they represent.

Definition 1. *Given two relations R and R' on space S , we say that R' refines R (abbreviation: $R' \sqsupseteq R$, or $R \sqsubseteq R'$) if and only if: $RL \cap R'L \cap (R \cup R') = R$.*

Intuitive interpretation: this definition means that R' has a larger domain than R , and assigns fewer images than R to the elements of the domain of R . This is formulated in the following Proposition.

Proposition 1. *Given two relations R and R' on space S . If R' refines R then $RL \subseteq R'L$ and $R' \cap RL \subseteq R$.*

2.2 Program Semantics

We consider a program P on space S and we let s be an element of S ; execution of P on initial state s may terminate after a finite number of steps in some final state s' when the exit statement of the program is reached; we then say that execution of P on s *converges*. Alternatively, execution of P on s may fail to converge, for any number of reasons: it enters an infinite loop; it addresses an array out of its bounds; it references a nil pointer; etc; we then say that execution of P on s *diverges*.

Given a program P on space S , the function of program P (which we also denote by P) is the set of pairs

of states (s, s') such that execution of P on state s converges and returns the final state s' . The domain of P is the set of states on which execution of P converges.

2.3 Absolute Correctness

A specification on space S is a binary relation on S ; it contains all the pairs of states (s, s') that the specifier considers correct. The correctness of a program P on space S can be determined with respect to a specification R on S according to the following definition.

Definition 2. Given a program P on state S and a specification R on S , we say that P is (totally) correct with respect to R if and only if P refines R . We say that P is partially correct with respect to R if and only if P refines $R \cap P$.

The following proposition gives set theoretic characterizations of total correctness and partial correctness.

Proposition 2. Given a program P on space S and a specification R on S , program P is totally correct with respect to R if and only if $dom(R) = dom(R \cap P)$; and program P is partially correct with respect to R if and only if $dom(R) \cap dom(P) = dom(R \cap P)$.

2.4 Detector Sets

Now that we know how to characterize correctness, we resolve to define sets of initial states that expose the incorrectness of a program with respect to a specification.

Definition 3. 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:

$$\Theta_T(R, P) = dom(R) \setminus dom(R \cap P).$$

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

$$\Theta_P(R, P) = (dom(R) \cap dom(P)) \setminus dom(R \cap P).$$

When we want to refer to a detector set without specifying a particular standard of correctness (partial, total), we simply say *detector set*, and we use the notation $\Theta(R, P)$.

Given that detector sets are intended to expose incorrectness, they are empty whenever there is no incorrectness to expose; this is formalized in the following proposition.

Proposition 3. Given a specification R on space S and a program P on S .

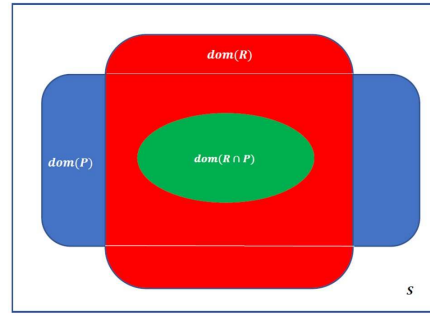


Figure 1: Detector Set of Pprogram P with Respect to Specification R for Total Correctness.

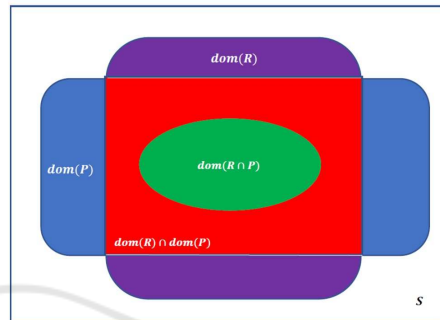


Figure 2: Detector Set of Pprogram P with Respect to Specification R for Partial Correctness.

- Program P is totally correct with respect to specification R if and only if $\Theta_T(R, P) = \emptyset$.
- Program P is partially correct with respect to specification R if and only if $\Theta_P(R, P) = \emptyset$.

2.5 Relative Correctness

Whereas absolute correctness is the property of a program to be (totally or partially) correct with respect to a specification, relative correctness is the property of a program to be more-correct than another with respect to a specification. It is natural to define relative correctness by means of detector sets: a program grows more and more (totally or partially) correct as its (total or partial) detector set grows smaller (in the sense of inclusion), culminating in absolute correctness when its detector set is empty. But relative total correctness has already been defined, in (Diallo et al., 2015); before we redefine it using a new formula, we ensure that the original formula is equivalent to the detector set-based formula we envision in this paper.

Proposition 4. Given a program P and a specification R , the following two conditions are equivalent:

$$f1 : dom(R \cap P) \subseteq dom(R \cap P').$$

$$f2 : \Theta_T(R, P') \subseteq \Theta_T(R, P).$$

Definition 4. We consider a specification R on space S and two programs P and P' on S .

Table 1: Definitions of Correctness by Means of DetectorSets.

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

- We say that P' is more-totally-correct than P with respect to R if and only if:

$$\Theta_T(R, P') \subseteq \Theta_T(R, P).$$

- We say that P' is more-partially-correct than P with respect to R if and only if:

$$\Theta_P(R, P') \subseteq \Theta_P(R, P).$$

Table 1 summarizes and organizes the definitions of correctness to help contrast them. Note the following relation between the detector sets of a program P with respect to a specification R :

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

From this simple equation, we can readily infer two properties about absolute correctness and relative correctness:

- *Absolute Correctness.* If a program P is totally correct with respect to specification R , then it is necessarily partially correct with respect to R .
- *Relative Correctness.* A program P' can be more-partially-correct than a program P either by being more-totally-correct (hence reducing the term $\Theta_T(R, P)$) or by diverging more widely (hence reducing the term $\text{dom}(P)$), or both.

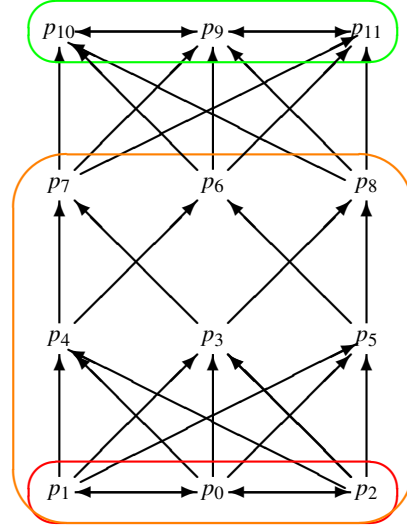
To illustrate the partial ordering properties of relative total correctness, we consider the following specification on space S of integers, defined by

$$R = \{(s, s') \mid 1 \leq s \leq 3 \wedge s' = s^3 + 3\}.$$

We consider twelve candidate programs, listed in Table 2. Figure 3 shows how these candidate programs are ordered by relative total correctness; The green oval shows those candidates that are absolutely correct, and the orange oval shows candidate programs that are incorrect; the red oval shows the candidate programs that are least correct.

3 SEMANTIC COVERAGE

We consider a program P on space S and a specification R on S , and we let T be a subset of S . We argue


 Figure 3: Ordering Candidate Programs by Relative Total (and Partial) Correctness with Respect to R .

that the purpose of test suite T is to prove or disprove the correctness of P with respect to R : T ought to be sufficiently thorough that, if P is incorrect with respect to R , then testing it on T ought to expose the incorrectness of P . Since the detector set of a program includes all the initial states on which execution of P fails, the effectiveness of a test suite T can be measured by the extent to which T encompasses all the elements of $\Theta(R, P)$. What precludes a test suite T from being a superset of $\Theta(R, P)$ are the elements of $\Theta(R, P)$ that are outside T , i.e. the set

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

The smaller this set, the higher the effectiveness of T ; if we want a measure of effectiveness that increases with the effectiveness of T , we take the complement of this set.

Definition 5. We consider a program P on space S and a specification R on S , and we let T be a subset of S .

- The semantic coverage of test suite T for the total correctness of program P with respect to specification R is denoted by $\Gamma_{[R,P]}^{TOT}(T)$ and defined by:

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

- The semantic coverage of test suite T for the partial correctness of program P with respect to specification R is denoted by $\Gamma_{[R,P]}^{PAR}(T)$ and defined by:

$$\Gamma_{[R,P]}^{PAR}(T) = T \cup \overline{\Theta_P(R, P)}.$$

See Figure 4. If we want to talk about semantic coverage without specifying the standard of correctness, we use the notation $\Gamma_{[R,P]}(T)$ defined by:

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

Table 2: Candidate Programs for Specification R .

p0: $s=\text{pow}(s,3)+4$;	p4: $s=\text{pow}(s,3)+s+1$;	p8: $s=\text{pow}(s,3)+s*s-4*s+8$;
p1: $s=\text{pow}(s,3)+5$;	p5: $s=\text{pow}(s,3)+s$;	p9: $s=2*\text{pow}(s,3)-6*s*s+11*s-3$;
p2: $s=\text{pow}(s,3)+6$;	p6: $s=\text{pow}(s,3)+s*s-5*s+9$;	p10: $s=3*\text{pow}(s,3)-12*s*s+22*s-9$;
p3: $s=\text{pow}(s,3)+s+2$;	p7: $s=\text{pow}(s,3)+s*s-3*s+5$;	p11: $s=4*\text{pow}(s,3)-18*s*s+33*s-15$;

4 ANALYTICAL VALIDATION

In this section we revisit the requirements put forth in section 1.2 and prove that the formula of semantic coverage proposed above does satisfy all these requirements.

4.1 Rq1: Monotonicity with Respect to the Test Suite

Definition 5 clearly provides that the semantic coverage of a test suite T is monotonic with respect to T .

4.2 Rq2: Monotonicity with Respect to Relative Correctness

The effectiveness of a test suite increases as the program under test grows more (totally or partially) correct.

Proposition 5. *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:*

$$\Gamma_{[R,P']}^{TOT}(T) \supseteq \Gamma_{[R,P]}^{TOT}(T).$$

Proposition 6. *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:*

$$\Gamma_{[R,P']}^{PAR}(T) \supseteq \Gamma_{[R,P]}^{PAR}(T).$$

4.3 Rq3: Monotonicity with Respect to Refinement

A test suite T grows more effective as the specification against which we are testing the program grows less-refined.

Proposition 7. *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:*

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

Proposition 8. *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:*

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

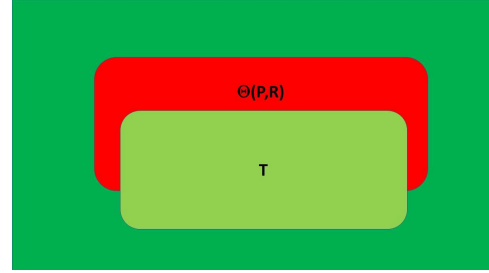


Figure 4: Semantic Coverage of Test T for Program P with respect to R (shades of green).

4.4 Rq4: Monotonicity with Respect to the Standard of Correctness

A test suite T is more effective for testing partial correctness than for testing total correctness.

Proposition 9. *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 .*

5 ILLUSTRATION

In this section we report on an experiment in which we evaluate the semantic coverage of a set of test suites; the sole purpose of this section is to illustrate the derivation of semantic coverage on a concrete example. We do compare semantic coverage against mutation coverage, but the intent of this comparison is not to validate semantic coverage any more than it is to validate mutation coverage. The sole purpose of this comparison is to satisfy our curiosity about how these two criteria rank sample test suites.

We consider the Java benchmark program of *jTerminal*, an open-source software product routinely used in mutation testing experiments (Parsai and Demeyer, 2017). We apply the mutant generation tool *LittleDarwin* in conjunction with a test generation and deployment class that includes 35 test cases (Parsai and Demeyer, 2017); we augment the benchmark test suite with two additional tests, intended to *trip* the base program *jTerminal*, by causing it to di-

verge. Application of LittleDarwin to jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants. Some of these mutants are equivalent to each other, i.e. they produce the same output for all 37 elements of T ; when we partition these 48 mutants by equivalence, we find 31 equivalence classes, and we select a mutant from each class; we let μ be this set. Orthogonally, we consider set T and we select twenty subsets thereof, derived as follows:

- $T1, T2, T3, T4, T5$: Five distinct test suites obtained from T by removing 5 elements at random.
- $T6, T7, T8, T9, T10$: Five distinct test suites obtained from T by removing 10 elements.
- $T11, T12, T13, T14, T15$: Five distinct test suites obtained from T by removing 15 elements.
- $T16, T17, T18, T19, T20$: Five distinct test suites obtained from T by removing one element.

Whereas mutation coverage is usually quantified by the mutation score (the fraction of killed mutants), in this paper we represent it by *mutation tally*, i.e. the set of killed mutants; we compare test suites by means of inclusion relations between their mutation tallies; like semantic coverage, this defines a partial ordering. We use two mutant generators, hence we get two ordering relations between test suites. To compute the semantic coverage of these test suites, we consider two standards of correctness (partial, total) and two specifications: We choose (the functions of) two mutants, M25 and M50, as specifications.

Hence we get six graphs on nodes $T1...T20$, representing six ordering relations of test suite effectiveness. Due to space limitations, we do not show these graphs, but we show in Table 3 the similarity matrix between these six graphs; the similarity index between two graphs is the ratio of the number of common arcs over the total number of arcs.

6 CONCLUSION

6.1 Summary

In this paper, we define detector sets for partial correctness and total correctness of a program with respect to a specification, and we use them to define absolute (partial and total) correctness as well as relative (partial and total) correctness. Also, we use detector sets to define the semantic coverage of a test suite, a measure of effectiveness which reflects the extent to which a test suite is able to expose the failure of an

incorrect program or, equivalently, the level of confidence it gives us in the correctness of a correct program. We illustrate the derivation of semantic coverage of sample test suites on a benchmark example.

6.2 Assessment

We do not validate our measure of effectiveness empirically, as we do not know what ground truth to validate it against; but we prove that it has a number of important properties, such as: monotonicity with respect to the standard of correctness; monotonicity with respect to the refinement of the specification against which the program is tested; and monotonicity with respect to the relative correctness of the program.

Other attributes of semantic coverage include that it is based on failures rather than faults, hence is defined formally using objectively observable effects rather than hypothesized causes. Also, semantic coverage defines a partial ordering between test suites, to reflect the fact that test suite effectiveness is itself a partially ordered attribute.

6.3 Threats to Validity

The main difficulty of the proposed coverage metric is that it assumes the availability of a specification, and that its derivation requires a detailed semantic analysis of the program. Yet as a formal measure of test suite effectiveness, semantic coverage can be used for reasoning analytically about test suites, or for comparing test suites even when their semantic coverage cannot be computed; for example, we may be able to compare $\Gamma_{[R,P]}(T)$ and $\Gamma_{[R,P]}(T')$ for inclusion without necessarily computing them, but by analyzing T , T' , $dom(P)$, $dom(R)$, and $dom(R \cap P)$.

6.4 Related Work

Coverage metrics of test suites have been the focus of much research over the years, and it is impossible to do justice to all the relevant work in this area (Hemmati, 2015; Gligoric et al., 2015; Andrews et al., 2006); as a first approximation, it is possible to distinguish between code coverage, which focuses on measuring the extent to which a test suite exercises various features of the code, and specification coverage, which focuses on measuring the extent to which a test suite exercises various clauses or use cases of the requirements specification. This can be tied to the orthogonal approaches to test data generation, using, respectively, structural criteria and functional criteria. Mutation coverage falls somehow outside of this dichotomy, in

Table 3: Graph Similarity of Semantic Coverage and Mutation Coverage.

Graph Similarity	Mut. Tally 1	Mut. Tally 2	$\Gamma_{[M25,P]}^{PAR}(T)$	$\Gamma_{[M50,P]}^{PAR}(T)$	$\Gamma_{[M25,P]}^{TOT}(T)$	$\Gamma_{[M50,P]}^{TOT}(T)$
Mut. Tally,1	1.00	0.43	0.34	0.35	0.34	0.50
Mut. Tally,2	0.43	1.00	0.67	0.70	0.67	0.53
$\Gamma_{[M25,P]}^{PAR}(T)$	0.34	0.67	1.00	0.66	1.0	0.46
$\Gamma_{[M50,P]}^{PAR}(T)$	0.35	0.70	0.66	1.00	0.66	0.62
$\Gamma_{[M25,P]}^{TOT}(T)$	0.34	0.67	1.00	0.66	1.00	0.46
$\Gamma_{[M50,P]}^{TOT}(T)$	0.50	0.53	0.46	0.62	0.46	1.00

that it depends exclusively on the program, not its specification, and that it operates by applying mutation operators; as such, it has often been used as a baseline for assessing the effectiveness of other coverage metrics (Andrews et al., 2006; Inozemtseva and Holmes, 2014). But mutation coverage also depends on the mutant generator, and can give different values for different generators.

Our work differs from other research in many ways: first, semantic coverage is not a number but a set. Second, semantic coverage is not intrinsic to the program, but depends also on the correctness standard and the specification. Third, semantic coverage is focused on failures rather than faults, since unlike faults, failures are an objectively observable attribute.

6.5 Research Prospects

We are exploring means to use the definition of semantic coverage to derive a function that is independent of the specification, and reflects the diversity of the test suite. We are also considering to expand the empirical study of semantic coverage.

ACKNOWLEDGEMENTS

The authors are very grateful to the anonymous reviewers for their valuable feedback. This work is partially supported by NSF grant DGE1565478.

REFERENCES

Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624.

Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Diallo, N., Ghardallou, W., and Mili, A. (2015). Correctness and relative correctness. In *Proceedings, 37th*

International Conference on Software Engineering, NIER track, Firenze, Italy.

Farooq, S. U., Quadri, S., and Ahmed, N. (2012). Metrics, models and measurement in software reliability. In *Proceedings, SAMI 2012*, Herlany, Slovakia.

Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., and Marinov, D. (2015). Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):1–33.

Hehner, E. C. (1992). *A Practical Theory of Programming*. Prentice Hall.

Hemmati, H. (2015). How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE.

Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings, 36th International Conference on Software Engineering*. ACM Press.

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

Mathur, A. P. (2014). *Foundations of Software Testing*. Pearson.

Morgan, C. C. (1998). *Programming from Specifications, Second Edition*. International Series in Computer Sciences. Prentice Hall, London, UK.

Parsai, A. and Demeyer, S. (2017). Dynamic mutant subsumption analysis using littedarwin. In *Proceedings, A-TEST 2017*, Paderborn, Germany.