Semantic Coverage: Measuring Test Suite Effectiveness

Samia Al Blwi[1], Amani Ayad[2], Besma Khaireddine[3], Imen Marsit[4], and Ali Mili[1] [a]

[1]*NJIT, Newark NJ USA*

[2]*Kean University, Union NJ, USA*

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

[4]*University of Sousse, Sousse Tunisia*

sma225@njit.edu, amanayad@kean.edu, besma.Khaireddine@gmail.com, imen.marsit@gmail.com, mili@njit.edu

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 test suites 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 a baseline for assessing the value of other coverage metrics (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; if a test suite T kills N mutants, what we can infer about T depends on whether T killed N different

[a] https://orcid.org/0000-0002-6578-5510

mutants or N times the same mutant (under distinct syntactic forms).

  – As an illustration, a test suite that kills ten equivalent mutants has a higher mutation score than a test suite that kills nine distinct mutants, even though the latter is nine times more effective than the former.

- The same test suite T may yield different mutation 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 (Andrews et al., 2005; Namin and Kakarla, 2011; 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. The impact of faults on failure rates is known to vary widely from one fault to another; an often cited empirical study reports an instance where repairing 60 percent of the faults in a software product leads to a 3 percent improvement in failure rate (Farooq et al., 2012; Sommerville, 2004).

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 as we define our measure.

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 under the name *semantic coverage*; 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 (Hoare, 1969; Manna, 1974; Gries, 1981).

The requirements we present below dictate how semantic coverage ought to vary as a function of these parameters. To follow our foregoing discussions, it is helpful to consider that the effectiveness of an artifact to fulfill a mission depends on the intrinsic attributes of the artifact as well as the difficulty of the mission. For example, the effectiveness of a locomotive to tow a train from A to B on a train track depends on the horsepower of the locomotive as well as the difficulty of the task (mass of the train, slope of the tracks, friction of the wheels, required speed, etc).

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[2].

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., 2015b). A test suite T ought to have increasingly greater semantic covarage as the program grows more-correct, since more-correct programs have fewer failures to reveal. Since relative correctness culminates in absolute correctness (Diallo et al., 2015b), we expect the semantic coverage of a test suite to reach its maximal value when applied to a (absolutely) correct program[3].

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; Wright, 1990; Aichernig et al., 2013; Banach and Poppleton, 2000); a given program P fails more often against a more-refined

[2]re: the train analogy; locomotives with higher horsepower ought to have higher scores.

[3]re: the train analogy; the effectiveness of a locomotive increases as it becomes easier to tow the train; if the track slopes downhill at a sufficient angle to reach the required speed, even a locomotive with zero horsepower will do the job.

(harder to satisfy) specification than a less-refined specification. Hence the same test suite ought to have greater semantic coverage for less-refined specifications[4].

Rq4 *Monotonicity with respect to the standard of correctness*. The distinction between partial correctness and total correctness has been central to the study of correctness verification (Hoare, 1969; Manna, 1974; Dijkstra, 1976; Gries, 1981), but has not been considered in software testing. Yet there is a difference between testing a program for total correctness and testing it for partial correctness: If we execute a program P on test t and P fails to terminate normally, then

- Under total correctness, we conclude that P is incorrect, and we repair P.
- Under partial correctness, we conclude that t is the wrong test, and we find another test.

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, since it has fewer failures to reveal.

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), which we summarize as follows: a *fault* is a feature of the program that precludes it from being correct; an *error* is the impact of a fault on the state of the program for a particular execution; a *failure* is the event where an error propagates to the output of the program and causes it to violate its specification.

  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; by focusing on failures rather than faults, we anchor our definition of semantic coverage in objectively observable effects rather than subjective speculations about causes.
- *Partial Ordering*. It is easy to imagine two test suites whose effectiveness cannot be ranked (i.e. we cannot say that one of them is better than the other): 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; this creates a potential for poor precision, whereby any two test suites will be ranked by their numeric scores, even when their effectiveness cannot be ranked. Hence we resolve to define semantic coverage, not as a number, but as an element of a partially ordered set; our goal is to ensure whenever the semantic coverage of two test suites $T1$ and $T2$ are ranked, e.g. $T1 \geq T2$, it is because $T1$ is better (in a sense to be defined) than $T2$.
- *Analytical Validation*. There are several reasons why we resort to analytical (vs. empirical) methods to validate our definition of semantic coverage:
  - First and foremost, we do not know of a widely accepted ground truth of test suite effectiveness against which we can validate our definition.
  - We acknowledge that many authors use mutation coverage as a baseline measure of test suite effectiveness, but we have no expectation that semantic coverage and mutation coverage be strongly correclated, not only because the former is a partial ordering whereas the latter is a total ordering, but also because semantic coverage and mutation coverage depend on different artifacts. Semantic coverage depends on the program, its specification and the standard of correctness; mutation coverage depends on the program and the mutation generator.
  - We cannot validate semantic coverage by means of its correlation with traditional metrics of syntactic coverage for several reasons: semantic coverage is a partial ordering whereas syntactic coverage metrics define total orderings; semantic coverage depends on the semantics of the program whereas syntactic metrics depends on its representation; semantic coverage depends on the specification and the standard of correctness in addition to the program,

[4] re: the train analogy; if the requirement for speed increases, all locomotives will have lower scores.

but syntactic coverage depends exclusively on the program.

Hence we resolve to validate our definition of semantic coverage 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; we do so without the expectation that the graphs be identical, for the reasons invoked above.

1.4 Relational Mathematics

We assume the reader is familiar with elementary discrete mathematics (Brink et al., 1997; Schmidt, 2010); in this section, we present some definitions and notation that we use throughout the paper. We define sets by means of C-like variable declarations; if we declare a set S by means of the following declarations:

```
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 lower case 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 usual set theoretic operations of union, intersection and complement; they also include the *domain* of a relation, denoted by $dom(R)$ for relation 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 $\widehat{R}$ and defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$. The *restriction* of relation R to subset T is the relation denoted by ${}_{T\backslash}R$ and defined by ${}_{T\backslash}R = \{(s,s')|s \in T \wedge (s,s') \in R\}$.

A relation R is said to be *reflexive* if and only if $I \subseteq R$; R is said to be *symmetric* if and only if $R \subseteq \widehat{R}$; R is said to be *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$; R is said to be *transitive* if and only if $RR \subseteq R$; R is said to be *deterministic* (or: to be a *function*) if and only if $\widehat{R}R \subseteq I$. A relation R is said to be a *partial ordering* if and only if it is reflexive, transitive and antisymmetric; R is said to be a *total ordering* if and only if it is a partial ordering and $R \cup \widehat{R} = L$.

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. In section 3 we see how detector sets can also be used to define semantic coverage in a way that meets all the requirements of section 1.2.

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 capture.

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 $R' \cap RL \subseteq R$.*

Proof. *Proof of $RL \subseteq R'L$.* We compute RL:

RL
= {by hypothesis}
$(RL \cap R'L \cap (R \cup R'))L$
= {the domain of a pre-restriction}
$RL \cap R'L \cap (R \cup R')L$
= {the domain of a union, associativity}
$(RL \cap R'L) \cap (RL \cup R'L)$
= {set theory}
$RL \cap R'L$.

From $RL = RL \cap R'L$ we infer, by set theory, $RL \subseteq R'L$.

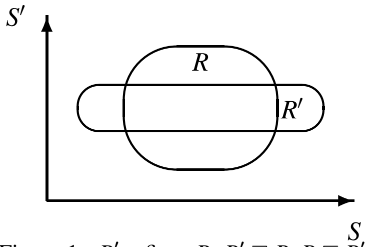


Figure 1: R' refines R: $R' \sqsupseteq R$, $R \sqsubseteq R'$.

Proof of $R' \cap RL \subseteq R$. We compute $R' \cap RL$.

$$\begin{array}{ll}
 & RL \cap R' \\
\subseteq & \{\text{set theory}\} \\
 & RL \cap (R \cup R') \\
= & \{\text{since } RL \subseteq R'L\} \\
 & RL \cap R'L \cap (R \cup R') \\
= & \{\text{by hypothesis}\} \\
 & R.
\end{array}$$

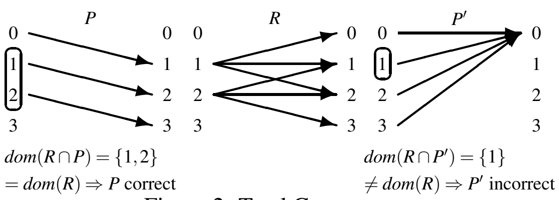
qed

This proposition shows that our definition of refinement is similar (modulo its relational formulation) to traditional definitions of refinement which equate refinement with having a weaker precondition ($RL \subseteq R'L$) and a stronger postcondition ($RL \cap R' \subseteq R$) (Hehner, 1992; Morgan, 1998; Gries, 1981; Dijkstra, 1976). See Figure 1.

2.2 Program Semantics

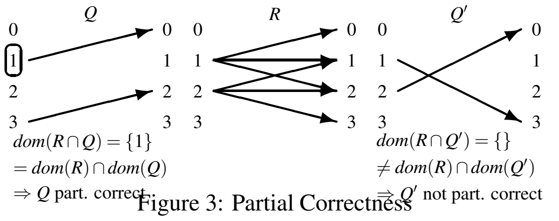
The semantics of a program can be modeled by a heterogeneous function from some input space to some output space, or by a homogeneous function from initial states to final states on the same space. For the sake of simplicity, and with little loss of generality, we choose the latter model. We consider 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 adresses an array out of its bounds; it references a nil pointer; it causes an arithmetic overflow or underflow; it attempts an illegal operation such division by zero, log() of a non-positive number, square root of a negative number, etc.; we then say that execution of P on s *diverges*.

Given a program P on space S, the function of program P (which, by abuse of notation, we also denote by P) is the set of pairs of states (s,s') such that if execution of program P starts in state s, it converges and returns the final state s'. As a consequence of this definition, the domain of P is the set of states such that execution of P on s converges.



P
R
P'
0 1 2 3
$dom(R \cap P) = \{1,2\}$
$= dom(R) \Rightarrow P$ correct
$dom(R \cap P') = \{1\}$
$\neq dom(R) \Rightarrow P'$ incorrect

Figure 2: Total Correctness



Q
R
Q'
0 1 2 3
$dom(R \cap Q) = \{1\}$
$= dom(R) \cap dom(Q)$
$\Rightarrow Q$ part. correct
$dom(R \cap Q') = \{\}$
$\neq dom(R) \cap dom(Q')$
$\Rightarrow Q'$ not part. correct

Figure 3: Partial Correctness

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 thet 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 $R \cap PL$.*

Figures 2 and 3 illustrate the properties of total and partial correctness; to be totally correct with respect to specification R, a program must obey the specification for all elements of $dom(R)$, whereas a partially correct program is required to obey the specification only where it converges.

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

Proof. The first proposition is due to Mills et al (Mills et al., 1986). To prove the second proposition, consider that the definition of partial correctness with respect to R is equivalent to total correctness with respect to $R' = R \cap PL$. Then, we find that $dom(R \cap PL) = dom(R) \cap dom(P)$, and $dom(R \cap PL \cap P) = dom(R \cap P)$. **qed**

We use this Proposition to briefly show that our formula of total and partial correctness is equivalent to traditional definitions of these properties (Hoare, 1969; Manna, 1974; Gries, 1981; Dijkstra, 1976). The formula of total correctness is: $dom(R) = dom(R \cap P)$; since $dom(R \cap P) \subseteq dom(R)$ is a tautology, the equation of total correctness is equivalent to:

$dom(R) \subseteq dom(R \cap P)$
$\Leftrightarrow$ {set theory}
$\forall s : s \in dom(R) \Rightarrow s \in dom(R \cap P)$
$\Leftrightarrow$ {interpreting the right hand clause}
$\forall s : s \in dom(R) \Rightarrow \exists s' : (s,s') \in (R \cap P)$
$\Leftrightarrow$ {definition of the domain}
$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge \exists s' : (s,s') \in (R \cap P)$
$\Leftrightarrow$ {since P is deterministic}
$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s,P(s)) \in R$.

We interpret this as: for any initial state s that satisfies the precondition ($s \in dom(R)$), program P converges ($s \in dom(P)$) and returns a final state $P(s)$ that satisfies the postcondition ($(s,P(s)) \in R$); this is exactly the definition of total correctness as it is known in (Manna, 1974; Gries, 1981; Dijkstra, 1976).

A similar agument will likewise show that our formula for partial correctness ($dom(R) \cap dom(P) = dom(R \cap P)$) is equivalent to traditional definitions of partial correctness (Hoare, 1969; Manna, 1974; Gries, 1981; Dijkstra, 1976).

The definitions in this section give us an opportunity to distinguish, once again, between testing a program for partial correctness and testing it for total correctness. The most critical step of software testing is the derivation of test data; this consists of defining a finite (and small) subset T of an infinite (or prohibitively large) set of possible tests τ. For total correctness of program P with respect to specification R, the set we are trying to approximate is $\tau = dom(R)$; for partial correctness that set of $\tau = dom(R) \cap dom(P)$.

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. The conditions of total and partial correctness are, respectively:

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

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

Since by set theory the left hand side of these equations is a superset of the right hand side, what precludes equality are elements of the left hand side that are outside the right hand side. Whence the following definition, due to (Mili, 2021).

Definition 3. *Given a program P on space S and a specification R on S:*

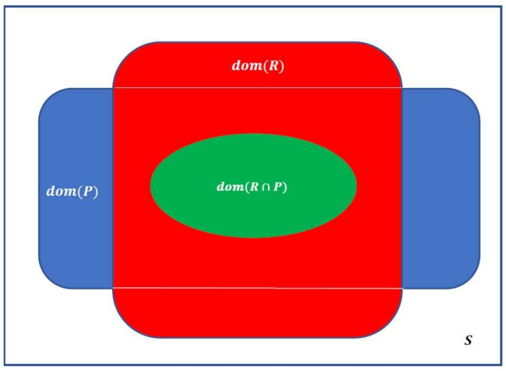


Figure 4: Detector Set of Pprogram P with Respect to Specification R for Total Correctness

- *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:*

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

This definition generalizes the concept of *detector* set introduced in (Shin et al., 2018) by taking into consideration the definition of correctness and the specification against which correctness is tested. See Figures 4 and 5. Since total correctness is a stronger property than partial correctness, it is a harder property to prove, hence an easier property to disprove; this is illustrated by Figures 4 and 5, where we can see that the detector set of total correctness is a superset of the detector set of partial correctness (offering more opportunity to disprove total correctness than partial correctness). 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 formualetd in the following proposition.

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

- *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 $\Theta_P(R,P) = \emptyset$.*

Proof. *For total correctness.* Necessity is trivial, per Proposition 2.

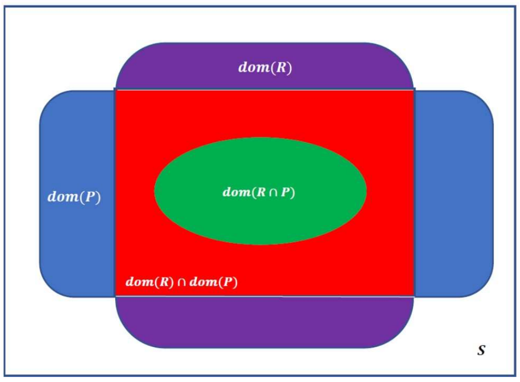


Figure 5: Detector Set of Pprogram P with Respect to Specification R for Partial Correctness

Proof of Sufficiency: From $dom(R) \setminus dom(R \cap P) = \emptyset$ we infer $dom(R) \subseteq dom(R \cap P)$; the inverse inclusion is a relational tautology, hence $dom(R) = dom(R \cap P)$, whence, by Proposition 2, P is totally correct with respect to R.

For partial correctness. Necessity is trivial, per Proposition 2.

Proof of Sufficiency: From $(dom(R) \cap dom(P)) \setminus dom(R \cap P) = \emptyset$ we infer $(dom(R) \cap dom(P)) \subseteq dom(R \cap P)$; the inverse inclusion is a relational tautology, hence $dom(R) \cap dom(P) = dom(R \cap P)$, whence, by Proposition 2, P is partially correct with respect to R. **qed**

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 (Mili et al., 2014; Diallo et al., 2015a); 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:*

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

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

Proof. To prove that $f1$ logically implies $f2$, it suffices to apply the complement on both sides of $f1$, inverting the inequality, then taking the intersection with $dom(R)$ on both sides.

To prove that $f2$ logically implies $f1$, we replace $\Theta_T()$ by its formula:

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

We rewrite both sides of the inequality using intersection and complement:

$dom(R) \cap \overline{dom(R \cap P')} \subseteq dom(R) \cap \overline{dom(R \cap P)}$.

We take the complement on both sides and invert the inequality:

$\overline{dom(R)} \cup dom(R \cap P) \subseteq \overline{dom(R)} \cup dom(R \cap P')$.

Taking the intersection with $dom(R)$ on both sides, we find, after distribution and cancellation:

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

Since $dom(R)$ is a superset of $dom(R \cap P)$ and $dom(R \cap P')$, this can be simplified as:

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

With the assurance that the new definition of relative total correctness is equivalent to the original definition (Mili et al., 2014; Diallo et al., 2015b), we write.

Definition 4. *We consider 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:*

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

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

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

Figure 6 illustrates relative total correctness by showing a specification (R) and two sets of programs: Q' is more-correct than Q with respect to R by virtue of imitating the correct behavior of Q; P' is more-correct than P with respect to R by virtue of a different correct behavior. Relative total correctness culminates in absolute total correctness in the following sense: a totally correct program is more-totally-correct than any candidate program. Figure 7 illustrates relative partial correctness by showing a specification (R) and two sets of programs: Q' is more-partially-correct than Q because it is more totally correct than Q; by contrast, P' is more-partially-correct than P by virtue of diverging more often (from the standpoint of partial correctness, a program that diverges evades accountability, and is considered partially correct).

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:

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

	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:	$\Theta_P(R,P') \subseteq \Theta_P(R,P)$	$\Theta_T(R,P') \subseteq \Theta_T(R,P)$

Table 1: Definitions of Correctness by Means of Detector Sets

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 $dom(P)$), or both.

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

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

We consider twelve candidate programs, listed in Table 2. Figure 8 shows how these candidate programs are ordered by relative total correctness; this ordering stems readily from the inclusion relations between the detector sets of the candidate programs with respect to R; the detector sets are given in Table 3. 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 (they violate specification R for every initial state in the domain of R). This example is clearly artificial, but we use it for illustration. Note that all twelve programs in this example converge for all initial states in S, hence $dom(P_i)=S$ for all P_i. Consequently, the detector sets of these programs for total correctness are identical to their detector sets for partial correctness; hence their ordering by relative partial correctness is identical to their ordering by relative total correctness.

3 Semantic Coverage

The effectiveness of an artifact is defined in reference to a specific purpose of the artifact, and ought to reflect to what extent the artifact fulfills its purpose. Hence the first question we must consider as we study the effectiveness of a test suite is: *what is the purpose of a test suite*?

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 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 runs successfyly on T, we should be able to infer that P is correct with respect to R; equivalently, if P is incorrect with respect to R, then testing it on T ought to expose the incorrectness of P (i.e. testing P on T ought to fail for at least one element t of T). 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

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

The smaller this set, the higher the effectiveness of T; if we want a measure of effectiveness that increases, rather than decreases, with the effectiveness of T, we take the complement of this set; whence the following definition.

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.*

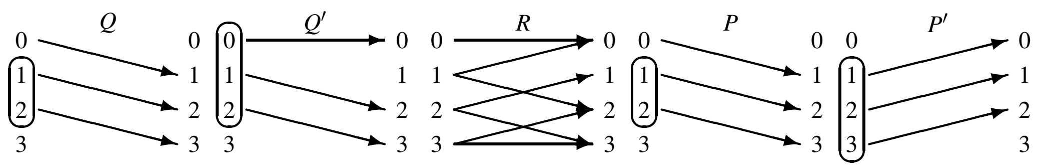
- *The* semantic coverage *of test suite T for the total correctness of program P with respect to specification R is denoted by* $\Gamma^{TOT}_{[R,P]}(T)$ *and defined by:*
  $$\Gamma^{TOT}_{[R,P]}(T)=T\cup\overline{\Theta_T(R,P)}.$$
- *The* semantic coverage *of test suite T for the partial correctness of program P with respect to specification R is denoted by* $\Gamma^{PAR}_{[R,P]}(T)$ *and defined by:*
  $$\Gamma^{PAR}_{[R,P]}(T)=T\cup\overline{\Theta_P(R,P)}.$$

See Figure 9. If we want to talk about semantic coverage without specifying the standard of correctness, we use the notation $\Gamma_{[R,P]}(T)$ defined by:

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

4 Analytical Validation

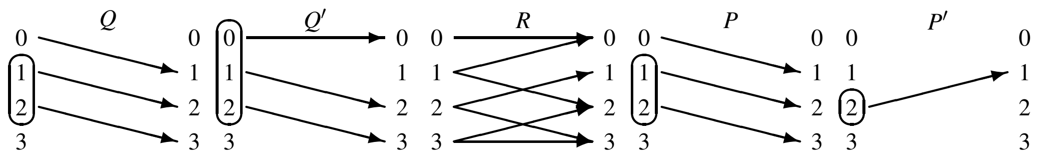
In this section we revisit the requirements put forth in section 1.2 and prove that the formula of semantic coverage proposed above does satisfy all these requirements.



Preserving Correct Behavior
$((R \cap Q) \subseteq (R \cap Q'))$

Preserving Correctness
$(dom(R \cap P) \subseteq dom(R \cap P'))$

Figure 6: Relative Total Correctness



Q' is more-partially-correct than Q
by virtue of being more-totally-correct:
$\Theta_P(R,Q') = \emptyset$, $\Theta_P(R,Q) = \{0\}$

P' is more-partially-correct than P
by virtue of diverging more broadly:
$\Theta_P(R,P') = \emptyset$, $\Theta_P(R,P) = \{0\}$

Figure 7: Relative Partial Correctness

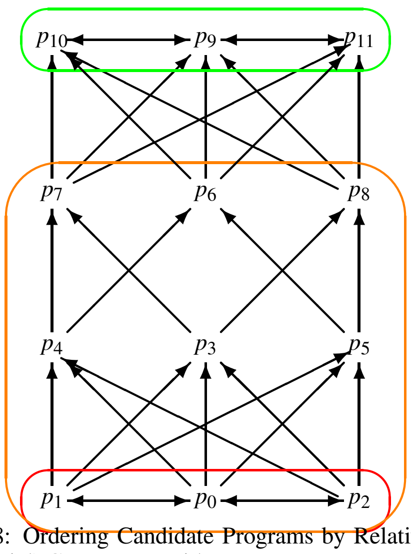


Figure 8: Ordering Candidate Programs by Relative Total (and Partial) Correctness with Respect to R

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. Note that in practice we are also interested in minimizing the size of T, but that is an *efficiency* concern, not an effectiveness concern.

4.2 Rq2: Monotonicity with Respect to Relative Correctness

The effectiveness of a tes 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:*
$\Gamma^{TOT}_{[R,P']}(T) \supseteq \Gamma^{TOT}_{[R,P]}(T)$.

Proof. By hypothesis, and according to Definition 4, we have:
$\Theta_T(R,P') \subseteq \Theta_T(R,P)$.
By taking the complement on both sides, inverting the inequality, and taking the union with T on both sides, we obtain the result sought. **qed**

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

Proof. By hypothesis, and according to Definition 4, we have:
$\Theta_P(R,P') \subseteq \Theta_P(R,P)$.
By taking the complement on both sides, inverting the inequality, and taking the union with T on both sides, we obtain the result sought. **qed**

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; this is true whether we are testing for total correctness and for partial correctness, as shown in the next two Propositions.

Proposition 7. *Given a program P on space S and*

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

Table 2: Candidate Programs for Specification R

p0	{1,2,3}	p1	{1,2,3}	p2	{1,2,3}
p3	{2,3}	p4	{1,3}	p5	{1,2}
p6	{1}	p7	{3}	p8	{2}
p9	{}	p10	{}	p11	{}

Table 3: Detector Sets of Candidate Programs for Total (and Partial) Correctness

two specifications R and R' on S, and a subset T of S. If R' refines R then:
$\Gamma^{TOT}_{[R',P]}(T) \subseteq \Gamma^{TOT}_{[R,P]}(T)$.

Proof. It suffices to prove $\Theta_T(R',P) \supseteq \Theta_T(R,P)$, i.e.:
$dom(R) \setminus dom(R \cap P) \subseteq dom(R') \setminus dom(R' \cap P)$.
Since $dom(R) \subseteq dom(R')$ (by hypothesis $R' \sqsupseteq R$), it suffices to prove:
$dom(R) \setminus dom(R \cap P) \subseteq dom(R) \setminus dom(R' \cap P)$.
We rewrite $\setminus$ using intersection and complement:
$dom(R) \cap \overline{dom(R \cap P)} \subseteq dom(R) \cap \overline{dom(R' \cap P)}$.
We complement both sides, invert the inequality:
$\overline{dom(R)} \cup dom(R' \cap P) \subseteq \overline{dom(R)} \cup dom(R \cap P)$.
Taking the intersection with $dom(R)$ on both size and simplifying, we get:
$dom(R) \cap dom(R' \cap P) \subseteq dom(R) \cap dom(R \cap P)$.
Let s be an element of $dom(R) \cap dom(R' \cap P)$; then $(s,P(s))$ is by definition an element of $RL \cap R'$; by the second clause of Proposition 1, $(s,P(s))$ is an element of R; since it is also by construction an element of P, it is an element of $(R \cap P)$. Whence,
$\Theta_T(R,P) \subseteq \Theta_T(R',P)$.
By complementing both sides of the inequation, inverting it, then taking the union with T on both sides, we find:
$\Gamma^{TOT}_{[R',P]}(T) \subseteq \Gamma^{TOT}_{[R,P]}(T)$. **qed**

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

Proof. In the proof of the previous proposition, we have found that if R' refines R, then
$\Theta_T(R,P) \subseteq \Theta_T(R',P)$.
By taking the intersection with $dom(P)$ on both sires, we find
$\Theta_P(R,P) \subseteq \Theta_P(R',P)$.
By complementing both sides of the inequation, inverting it, then taking the union with T on both sides, we find:

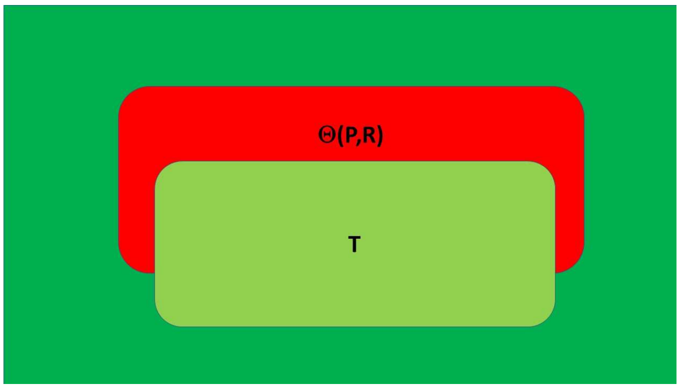


Figure 9: Semantic Coverage of Test T for Program P with respect to R (shades of green)

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

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.*

Proof. By definition of detector sets, we have:
$\Theta_P(R,P) \subseteq \Theta_T(R,P)$.
By complementing both sides and inverting the inequality, we find:
$\overline{\Theta_P(R,P)} \supseteq \overline{\Theta_T(R,P)}$.
By taking the union with T on both sides, we find the result sought. **qed**

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; because mutation coverage and semantic coverage depend on different artifacts (the program and mutant generator for the former, the program, specification, and standard of correctness for the latter) we have no expectation that they be the same.

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 diverge, so as to distinguish between partial correctness and total correctness; we designate this test suite by T. 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 at random.
- *T11, T12, T13, T14, T15*: Five distinct test suites obtained from T by removing 15 elements at random.
- *T16, T17, T18, T19, T20*: Five distinct test suites obtained from T by removing one element at random.

Whereas mutation coverage is usually quantified by the mutation score (the fraction of killed mutants) (Li et al., 2017), 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 show only two of these graphs, given in Figure 10 for the mutation tally and in Figure 11 for the semantic coverage of total correctness with respect to M50; the six graphs are given in the artifact associated with this paper, along with the data used to derive them. Table 4 shows the similarity between the six graphs, where the similarity between any two graphs is computed as the ratio of the number of their common arcs over the total number of arcs. All the data used to build these graphs and analyze them is available online, but cannot be shared explicitly due to the requirement of anonymity.

6 Conclusion

6.1 Summary

In this paper, we define detector sets for partial correctness and total correctness of a program with respetc 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.

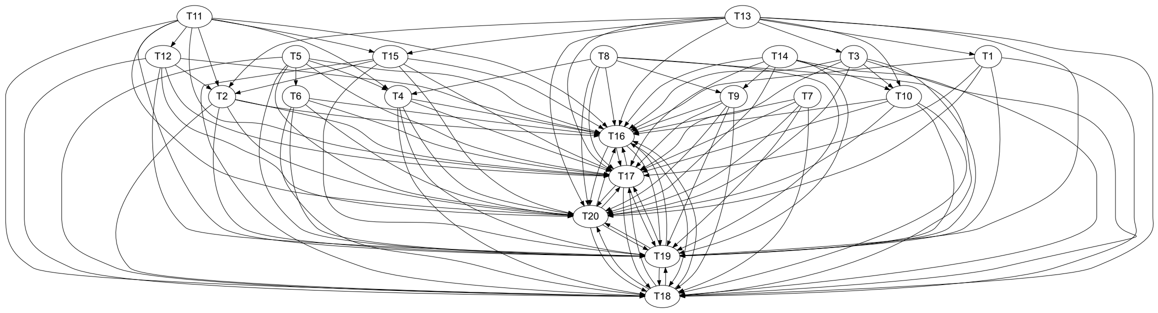


Figure 10: Ordering Test Suites T_i by Mutation Coverage

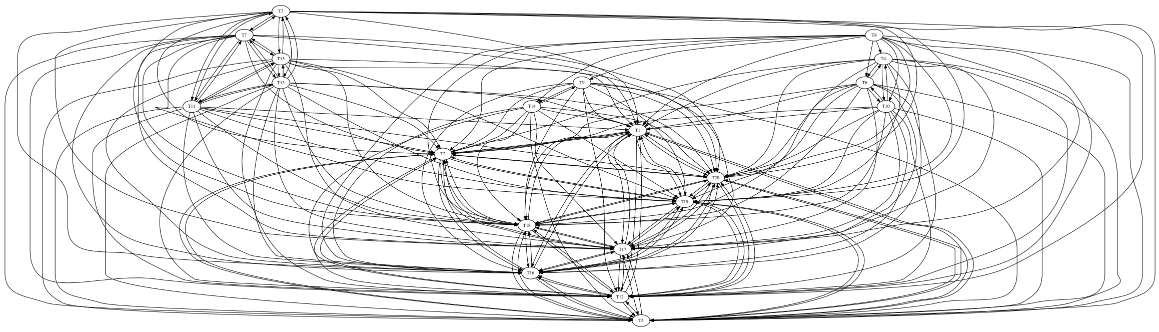


Figure 11: Ordering Test Suites by Semantic Coverage for Total Correctness with respect to M50

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

Table 4: Graph Similarity of Semantic Coverage and Mutation Coverage

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 reserch over the years, and it is impossible to do justice to all the relevant work in this area (Lyu et al., 1994; Lingampally et al., 2007; Hemmati, 2015; Gligoric et al., 2015; Andrews et al., 2006; Someoliayi et al., 2019; Ball, 2004); 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 that it depends exclusively on the program, not its specification, and that it operates by applying mutation operators, wherever they are applicable, without regard to syntactic coverage; 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 these research efforts in a number of ways: perhaps first and foremost, our coverage semantic measure is not a number but a set; as such, it is not totally ordered by numeric inequality, but partially ordered by set inclusion. Second, semantic coverage is not intrinsic to the program, but depends also on the correctness standard used in testing, and the specification with respect to which correctness is judged. Third, semantic coverage is focused on revealing failures rather than diagnosing faults, on the grounds that failures are an objectively observable attribute, but faults are hypothesized causes of observed failures.

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 by analyzing its relationship to existing coverage metrics.

REFERENCES

Aichernig, B., Jobstl, E., and Kegele, M. (2013). Incremental refinement checking for test case generation. In *Tests and Proofs*, pages 1–19.

Andrews, J., Briand, L., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *Proceedings, ICSE.*

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.

Ball, T. (2004). A theory of predicate-complete test coverage and generation. In *International Symposium on Formal Methods for Components and Objects*, pages 1–22. Springer.

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

Brink, C., Schmidt, G., and Kahl, W. (1997). *Relational Methods in Computer Science*. Springer Verlag.

Diallo, N., Ghardallou, W., Frias, M., Jaoua, A., and Mili, A. (2015a). What is a fault? and why does it matter? Technical report, NJIT, Newark, NJ, http://web.njit.edu/~mili/jrn.pdf.

Diallo, N., Ghardallou, W., and Mili, A. (2015b). Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering, NIER track*, Firenze, Italy.

Dijkstra, E. (1976). *A Discipline of Programming*. Prentice Hall.

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.

Gries, D. (1981). *The Science of Programming*. Springer Verlag.

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.

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

Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Procedings, 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*.

Li, X., Wang, Y., and Lin, H. (2017). Coverage based dynamic mutant subsumption graph. In *Proceedings, International Conference on Mathematics, Modeling and Simulation Technologies and Applications*.

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

Lyu, M. R., Horgan, J., and London, S. (1994). A coverage analysis tool for the effectiveness of software testing. *IEEE transactions on reliability*, 43(4):527–535.

Manna, Z. (1974). *A Mathematical Theory of Computation*. McGraw-Hill.

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

Mili, A. (2021). Differentiators and detectors. *Information Processing Letters*, 169.

Mili, A., Frias, M., and Jaoua, A. (2014). On faults and faulty programs. In Hoefner, P., Jipsen, P., Kahl, W., and Mueller, M. E., editors, *Proceedings, RAMICS 2014*, volume 8428 of *LNCS*, pages 191–207.

Mills, H. D., Basili, V. R., Gannon, J. D., and Hamlet, D. R. (1986). *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma.

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

Namin, A. S. and Kakarla, S. (2011). The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings, ISSTA*.

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

Schmidt, G. (2010). *Relational Mathematics*. Number 132 in Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Shin, D., Yoo, S., and Bae, D.-H. (2018). A theoretical and empirical study of diversity-aware mutation adequancy criterion. *IEEE TSE*, 44(10).

Someoliayi, K. E., Jalali, S., Mahdieh, M., and Mirian-Hosseinabadi, S.-H. (2019). Program state coverage: a test coverage metric based on executed program states. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 584–588. IEEE.

Sommerville, I. (2004). *Software Engineering*. Addison Wesley, seventh edition.

Wright, J. V. (1990). A lattice theoretical basis for program refinement. Technical report, Dept. of Computer Science, Åbo Akademi, Finland.