# A Measure of Semantic Coverage

Anonymous, Pending Review

***Abstract*—It is common to assess the quality of a test suite by the extent to which it exercises syntactic features of the program, or the extent to which it exercises use cases of the requirements specification. In this paper we present a measure of test suite effectiveness, which reflects the extent to which a test suite reveals the breadth of failure of a program to meet its specification; we refer to it as *semantic coverage*. We validate it analytically by showing that it is monotonic with respect to several attributes that characterize test suite effectiveness, and we illustrate it on a sample benchmark program.**

***Index Terms*—Test suites, test suite effectiveness, syntactic coverage, semantic coverage, detector set, absolute correctness, relative correctness.**

## I. On the Effectiveness of a Test Suite

### A. Motivation

If the purpose of test suites is to reveal the presence of faults in programs, then it is legitimate to measure the quality 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, such as statements, conditions, branches, paths, etc [20]; 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 [4]; also, not all syntactic paths are feasible (re: infeasible paths, unreachable code, etc.). A better measure of test suite quality is mutation coverage, where the quality of a test suite is measured by 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 [3], [14], it has issues of its own: the same mutation score may mean vastly different things depending on whether the killed mutants are all distinct, all equivalent, or something in between (partitioned into several equivalence classes); if a test suite $T$ kills $N$ mutants, what we can infer about $T$ depends to a vast extent on whether $T$ killed $N$ different mutants or $N$ times the same mutant (i.e. $N$ equivalent versions of the same mutant). Also the same test suite $T$ may yield different mutation scores for different sets of mutants, hence it cannot be considered as an intrinsic attribute of the test suite.

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. In the next section we present and justify a number of criteria that a measure of test suite effectiveness ought to satisfy, and in section I-C we present and justify some design decisions that we resolve to adopt in the process of defining our measure.

Funding source undisclosed, for anonymity

In section II we introduce detector sets, and discuss their significance for the purposes of program testing and program correctness. In section III we use detector sets to define the concept of *semantic coverage*, and in section III-B we validate our definition by showing, analytically, that it meets all the requirements set forth in section I-B. In section IV we illustrate the derivation of semantic coverage on a sample benchmark example, and show its empirical relationship to mutation scores. We conclude in section V by summarizing our findings, critiquing them, comparing them to related work, and sketching directions of further research.

### B. 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 [10], [13], [19].

Hence whereas most traditional measures of test suite coverage depend exclusively on the program under test, our definition depends also on the standard of correctness that we are testing the program for, as well as the specification against which correctness is tested.

The requirements we present below dictate how semantic coverage ought to vary as a function of each of these three parameters: the standard of correctness, the specification, and the program. To understand the discussions below, it is helpful to reason by analogy: the effectiveness of a tool to perform a task increases with the intrinsic power of the tool, but it also increases as the task becomes easier.

- *The Program*. Relative correctness is the property of a program to be more-correct than another with respect to a specification [7]; each fault repair makes the program more-correct, culminating in absolute correctness when all faults are repaired [15]. The same test suite ought to have greater and greater semantic coverage as the program becomes increasingly more-correct with respect to the same specification, as there are fewer and fewer faults left to discover.
- *The Specification*. Specifications are naturally ordered by refinement, whereby more-refined specifications represent stronger/ harder to satisfy requirements [1], [6], [11],

[22], [26]; it is harder to test a program against a more-refined specification than against a less-refined specification, since a more-refined specification represents a stronger requirement to test against. Hence the same test suite ought to have greater semantic coverage for less-refined specifications.

- *The Standard of Correctness*. There is a difference between testing a program for total correctness and testing a program for partial correctness: under total correctness, if we select a test data $t$ and the program fails to terminate on $t$, we conclude that the program fails the test, i.e. is deemed incorrect; under partial correctness, if we select a test data $t$ and the program fails to terminate on $t$, we conclude that the test data selection is wrong (and we choose another test data). Total correctness is a stronger property than partial correctness, hence it is more difficult to test a program for total correctness than for partial correctness. Hence the same test suite $T$ must have greater semantic coverage if it is applied to partial correctness than if applied to total correctness.

Of course, we also want the effectiveness of a test suite $T$ to increase with $T$, i.e. if $T'$ is a superset of $T$ then the effectiveness of $T'$ is greater than that of $T$. In section III-B we present propositions to the effect that the formula of semantic coverage we present in section III satisfies all these requirements.

### C. Design Principles

We resolve to adopt the following design decisions in defining a measure of semantic coverage:

- *Focus on Failure*. We adopt the definitions of fault, error and failure proposed by Avizienis et al [4]: 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 (that sensitizes the fault); a *failure* is the event where the program violates its specification because an error has propagated to the program's output. We quantify the effectiveness of a test suite, not by its ability to reveal faults, but by its ability to reveal failures. The reason is that failures are an objectively verifiable observation: by contrast, resolving that some feature in the source code of the program is the cause of the observed failure is a subjective assumption, as there is no one-to-one mapping between observed failures and faults.
  Hence we resolve to define semantic coverage of a test suite $T$ in terms of the scale of program failures that $T$ can reveal.
- *Partial Ordering*. It is common to think of measurement as the assignment of a numeric value to an artifact, to reflect a particular attribute thereof. But assigning numeric values to an attribute that is not totally ordered causes a loss of precision: it is easy to imagine two test suites that cannot be compared (e.g. they expose unrelated sets of failures), yet if we assign them numbers, we will always find that one test suite is assigned a greater number than the other.
  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 that whenever two test suites have comparable semantic coverages, it is because the test suite with the greater coverage is superior (in a sense to be defined) to the other.
- *Analytical Validation*. There are several reasons why we prefer to rely on analytical argument 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. Second, most existing coverage metrics reflect program attributes only, while our semantic coverage definition depends on the correctness standard and the specification, in addition to the program. Hence we resolve to validate our definition of semantic coverage on the basis of analytical arguments, by arguing that it captures the right attributes and that it satisfies all the properties that we mandate in section I-B.
  Still, we do include an empirical validation step: In section IV we compute the semantic coverage of a set of (20) test suites of a benchmark program for two distinct specifications and two standards of correctness, and we compare the four graphs so derived against the graph that ranks these test suites by mutation coverage; but we do so without the expectation that the graphs be identical, because semantic coverage depends on the program, the specification and the correctness standard, whereas mutation coverage depends on the program and the mutation generator. As we discuss in section IV, the similarity we observe empirically between the graphs exceeds our expectation.

## II. Detector Sets

### A. Relational Mathematics

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 of 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 ($I$), 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 rectangular

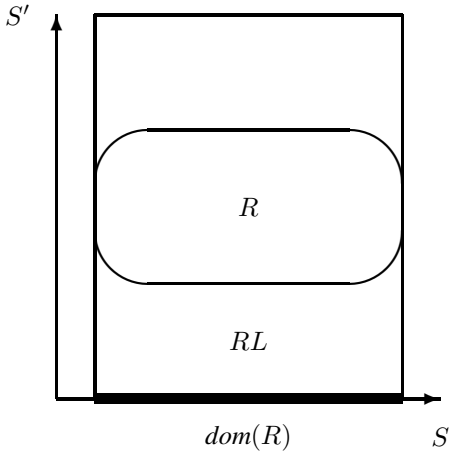


Fig. 1. Relations $R$ and $RL$

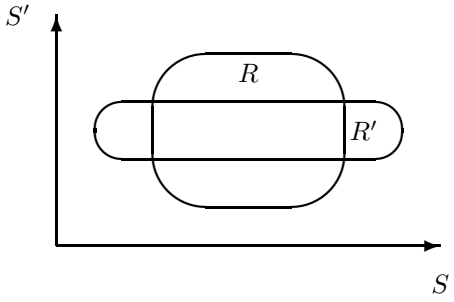


Fig. 2. $R'$ refines $R$: $R' \sqsupseteq R$, $R \sqsubseteq R'$.

relation $RL = dom(R) \times S$. See Figure 1; we use $RL$ as a representation of the domain of $R$ in relational form.

### B. Program Semantics

*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: $RL \subseteq R'L$ and $RL \cap R' \subseteq R$.
Intuitive interpretation: this definition means that $R'$ has a larger domain than $R$, and that $R'$ assigns fewer images than $R$ to the elements of the domain of $R$. This definition of refinement is the relational equivalent to the usual formula of refinement which provides for weaker precondition ($RL \subseteq R'L$) and stronger postcondition ($RL \cap R' \subseteq R$) [8], [10], [11], [22]. See Figure 2.

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 terminates normally in state $s'$; by *terminates normally* we mean that the execution terminates after a finite number of steps, without attempting any illegal operation such as a division by zero, an array reference out of bounds, a reference to a nil pointer, a square root of a negative number, etc... As a consequence of this definition, the domain of $P$ is the set of states (elements of $S$) such that execution of $P$ on $s$ terminates normally.

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 $R \cap PL$.
Except for the fact that they are formulated in relational terms, these definitions are equivalent to traditional definitions of total and partial correctness [8], [10], [13], [19]. Figures 3 and 4 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 terminates.

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

*Proposition 1:* 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 [21]. 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**

The domain of $(R \cap P)$ is the set of initial states for which execution of $P$ produces a final state that satisfies specification $R$; we refer to this as the *competence domain* of $P$ with respect to $R$. Whereas Definition 2 gives criteria of absolute correctness (a program is either correct or incorrect), the following definition gives criteria of *relative* correctness: a program $P'$ may be more-correct than a program $P$, which both are incorrect.

*Definition 3:* Given programs $P$ and $P'$ on space $S$, and specification $R$ on $S$, we say that $P'$ is *more-totally-correct* than $P$ with respect to $R$ if and only if:

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

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

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

The definition of relative total correctness is due to [7]; the definition of relative partial correctness is derived herein by analogy, for the sake of completeness. To contrast the definitions given in Definition 2 with those given herein for relative correctness, we may refer to the former as *absolute correctness*.

A program $P'$ is more-totally-correct than a program $P$ with respect to $R$ if and only if it has a larger competence domain with respect to $R$, i.e. it obeys specification $R$ over a larger set

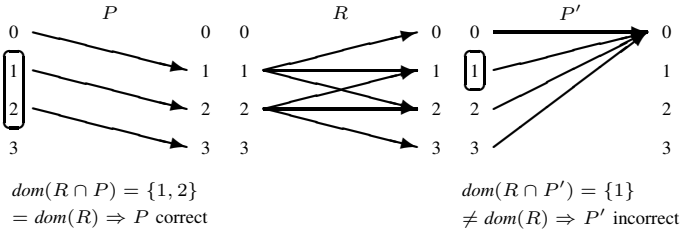


Fig. 3. Total Correctness

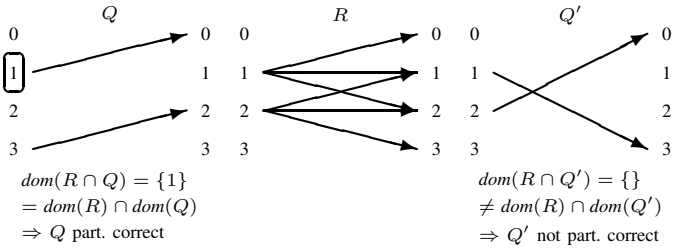


Fig. 4. Partial Correctness

of initial states; see Figure 5. A program $P'$ is more-partially-correct than a program $P$ with respect to $R$ if and only if it has a larger competence domain with respect to $R$, or it fails to terminate normally over a larger set of initial states (i.e. has a smaller domain); see Figure 6. From the standpoint of partial correctness, a program can evade accountability by failing to terminate.

### C. Detector Sets

The detector set of a program $P$ on space $S$ for some standard of correctness (partial or total) with respect to specification $R$ is the set of states that disprove the correctness of $P$ with respect to $R$ (i.e. execution of $P$ on these states yields a final state $s'$ that violates specification $R$).

*Definition 4:* 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 $\Theta(R, P)$. This definition generalizes the concept of *detector* set introduced in [24] by taking into consideration the definition of correctness and the specification against which correctness is tested. 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 2:* 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 the detector set for total correctness of $R$ and $P$ is empty. Program $P$ is partially correct with respect to specification $R$ if and only if the detector set for partial correctness of $R$ and $P$ is empty.

**Proof.** *For total correctness*. Necessity is trivial, per Proposition 1. 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 1, $P$ is totally correct with respect to $R$.

*For partial correctness*. Necessity is trivial, per Proposition 1. 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 1, $P$ is partially correct with respect to $R$. **qed**

### D. Properties

Since total correctness logically implies partial correctness, any test that disproves partial correctness disproves necessarily total correctness. Whence the following proposition.

*Proposition 3:* The detector set for partial correctness of a program $P$ with respect to specification $R$ is a subset of the detector set for total correctness of $P$ with respect to $R$.

**Proof.** This stems readily from the observation that $\Theta_P(R, P) = \Theta_T(R, P) \cap dom(P)$. **qed**

In addition to the standard of correctness, detector sets depend on the program under and the specification against which the program is being tested. The following two propositions highlight how detector sets vary according to the relative correctness of the program, and the refinement of the specification. Before we present the propositions, we introduce a lemma.

*Lemma 1:* Given three sets $A$, $B$ and $C$, the following conditions are equivalent:

$A \setminus B \subseteq A \setminus C$.

$A \cap C \subseteq A \cap B$.

**Proof.** $A \setminus B \subseteq A \setminus C$ can be written as:

$A \cap \overline{B} \subseteq A \cap \overline{C}$.

Taking the complement on both sides, using DeMorgan's laws, and inverting the inequality yields:

$\overline{A} \cup C \subseteq \overline{A} \cup B$.

Taking the intersection with $A$ on both sides, distributing, and canceling the term $A \cap \overline{A}$, we get:

$A \cap C \subseteq A \cap B$.

Taking the complement again and inverting the inequality, then taking the intersection with $A$ yields the original equation. Hence the two equations are equivalent. **qed**

*Proposition 4:* Given a specificartion $R$ on space $S$ and two programs $P$ and $P'$ on $S$. $P'$ is more-totally-correct than $P$ with respect to specification $R$ if and only if:

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

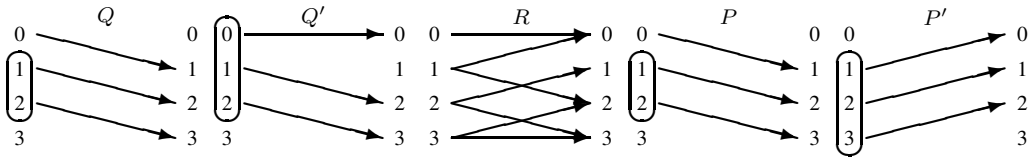


Fig. 5. Relative Total Correctness

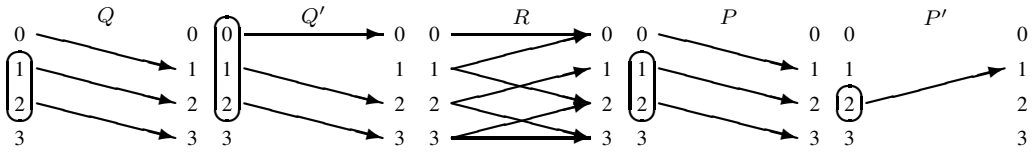


Fig. 6. Relative Partial Correctness

If $P'$ is more-partially-correct than $P$ with respect to specification $R$ then:

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

**Proof.** *For Total Correctness*: Necesity is trivial, per Definition 3. Proof of Sufficiency: By Definition 4, we have:

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

Using Lemma 1, we find:

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

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

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

*For Partial Correctness*: By Definition 3, we have:

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

We take the complement on both sides and invert the inequality, we find (by DeMorgan's laws):

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

By taking the intersection with $dom(R)$ on both sides, we find:

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

**qed**

The intuitive interpretation of Proposition 4 is straightforward: If $P'$ is more-correct (totally or partially) than $P$, then any test that reveals the failure of $P'$ reveals necessarily the the failure of $P$ (if the more-correct program fails, then so does the less-correct program).

Table I summarizes the results of Propositions 2 and 4 (though for relative partial correctness we have proven necessity but not sufficiency of the provided condition).

Whereas Proposition 4 stipulates how the detector set of a program $P$ with respect to $R$ varies according to the program $P$, we now ponder the question of how it varies according to the specification $R$. We consider a program $P$ on space $S$

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

TABLE I
DEFINITIONS OF CORRECTNESS

and two specifications $R$ and $R'$ on $S$ such that $R'$ refines $R$; $R'$ refines $R$ means that $R'$ represents a stronger requirement than $R$; any test that disproves the correctness of $P$ with respect to the less-refined (weaker) specification $R$ disproves (a fortiori) the correctness of $P$ against the (stronger) more-refined specification $R'$. Whence the following proposition.

*Proposition 5:* Given a program $P$ on space $S$ and two specifications $R$ and $R'$ on $S$. If $R'$ refines $R$ then the detector set of $P$ with respect to $R$ is a subset of the detector set of $P$ with respect to $R'$.

**Proof.** *For Total Correctness*. We must prove:

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

Since $dom(R)$ is a subset of $dom(R')$ (by hypothesis, since $R'$ refines $R$), it suffices to prove:

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

By Lemma 1, this is equivalent to:

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

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 the definition of refinement (Definition 1), $(s, P(s))$ is an element of $R$; since it is also by construction an element of $P$, it is an element of $(R \cap P)$. Therefore $s$ is an element of $dom(R \cap P)$.

*For Partial Correctness*. $\Theta_P(R, P) \subseteq \Theta_P(R', P)$ stems

readily from $\Theta_T(R,P) \subseteq \Theta_T(R',P)$ (proved above) and $dom(R) \subseteq dom(R')$ (by hypothesis, since $R'$ refines $R$). **qed**

## III. Semantic Coverage

### A. Definition

In this section we resolve to propose a formula that measures the effectiveness of a test suite $T$ for a program $P$ to be tested for total and partial correctness with respect to specification $R$. As a first step, we ponder the question: what is an ideal test suite? Then we define semantic coverage of an arbitrary (not necessarily ideal) test suite to reflect the extent to which the test suite comes close to the ideal case.

Given a program $P$ on space $S$ and a specification $R$ on $S$, an ideal test suite $T$ is one that is a superset of the detector set of $P$ with respect to $R$ for the selected correctness standard, since such a test suite exposes all the failures of $P$ with respect to $R$. Let $\Theta(R,P)$ be the detector set of program $P$ with respect to $R$; what precludes $T$ from being a superset of $\Theta(R,P)$ is the set of elements of $\Theta(R,P)$ that are outside $T$, i.e.

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

The smaller this set, the better the test suite $T$; since we want a quantity that increases with the effectiveness of $T$ rather than decreases, we take the complement of this quantity, whence the following definition.

*Definition 5:* The semantic coverage of test suite $T$ for the total correctness of program $P$ with respect to specification $R$ on space $S$ 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 $R$ on space $S$ is denoted by $\Gamma^{PAR}_{[R,P]}(T)$ and defined by:

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

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

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

### B. Analytical Validation

In this section we revisit the requirements put forth in section I-B and prove that the formula of semantic coverage proposed above does satisfy all these requirements. For the most part, the propositions presented in this section stem immediately from the properties of detector sets discussed in section II-D.

*Proposition 6:* 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 Proposition 3, 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**

*Proposition 7:* 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.** If $P'$ is more-totally-correct than $P$ with respect to $R$ then, according to Proposition 4,

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

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 8:* 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.** If $P'$ is more-partially-correct than $P$ with respect to $R$ then, according to Proposition 4,

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

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**

*Proposition 9:* Given a program $P$ on space $S$ and two specifications $R$ and $R'$ on $S$, and a subset $T$ of $S$. If $R'$ refines $R$ then:

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

**Proof.** Since $R'$ refines $R$ then, by Proposition 5,

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

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 10:* 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.** Since $R'$ refines $R$ then, by Proposition 5,

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

By complementing both sides of the inequation, inverting it, then taking the union with $T$ on both sides, we find:

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

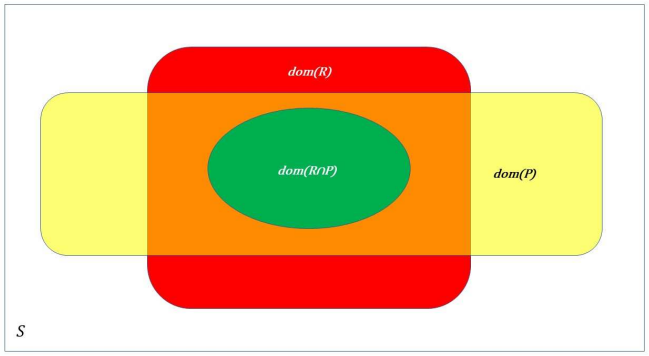


Fig. 7. Detector Sets for Partial Correctness (orange) and Total Correctness (red+orange)

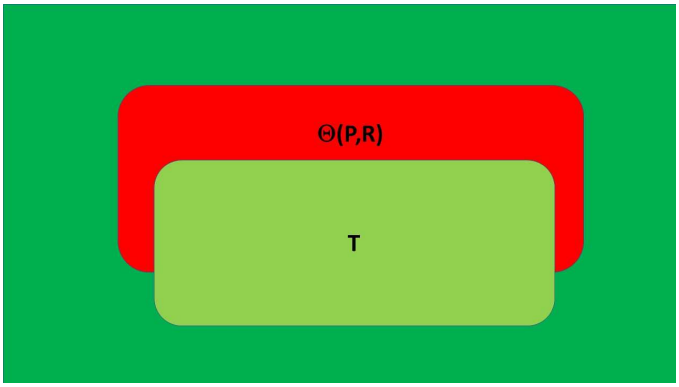


Fig. 8. Semantic Coverage of Test $T$ for Program $P$ with respect to $R$ (shades of green)

## IV. Illustration

In this section we report on an experiment in which we evaluate the semantic coverage of a set of test suites, and compare our findings to an existing measure of coverage, namely *mutation coverage*. To this effect, we consider the Java benchmark program of *jTerminal*[1], an open-source software product routinely used in mutation testing experiments [23]. We apply the mutant generation tool *LittleDarwin* in conjunction with a test generation and deployment class that includes 35 test cases [23]; we augment the benchmark test suite with two additional tests, intended specifically to *trip* the base program *jTerminal*, by causing it to diverge (i.e. fail to terminate normally). The purpose of these tests is to enable us to distinguish between partial correctness and total correctness. We let $T$ designate the augmented test suite codified in this test class. Execution of LittleDarwin on jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants:

`m1, m2, m7, m8, m9, m10, m11, m12, m13, m14, m15, m16, m17, m18, m19, m21, m22, m23, m24, m25, m26, m27, m28, m44, m45, m46, m48, m49, m50, m51, m52, m53, m54, m55, m56, m57, m58, m59, m60, m61, m62, m63, m83, m88, m89, m90, m92, m93.`

Some of these mutants are equivalent to each other, i.e. they produce the same output for each of the 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: $\mu =$

`m1, m2, m7, m11, m13, m15, m19, m21, m22, m23, m24, m25, m27, m28, m44, m45, m46, m48, m49, m50, m51, m52, m53, m55, m56, m57, m60, m63, m92, m93.`

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 15 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 5 elements at random.
- *T16, T17, T18, T19, T20*: Five distinct test suites obtained from $T$ by removing one element at random.

To have a baseline against which we compare our measures of semantic coverage, we rank the test suites T1... T20 by mutation coverage [16]; but we do not equate mutation coverage with the ratio of killed mutants over generated mutants; rather we rank test suites by comparing, in terms of set inclusion, the sets of mutants they kill. The result is shown in Figure 9.

To compute the semantic coverage of these test suites, we need to define specifications against which the test is carried out; for the sake of this experiment, we choose mutants M25 and M50 as sample specifications, and we compute the semantic coverage of test suites T1... T20 for total correctness and partial correctness with respect to M25 and M50. For each specification and standard of correctness, we compute the semantic coverage of each of the twenty test suites, which we compare by inclusion. The result is shown in Figures 10, 11, 12 and 13 for, respectively, the partial correctness with respect to M25 and M50 then the total correctness with respect to M25 and M50.

While we have no expectation that the graph of mutation coverage (Figure 9) and the graphs of semantic coverage be the same, since the former is intrinsic to the program whereas the latter also depend on the specification and the standard of correctness, we are interested in considering the similarity between the mutation coverage graph and the graphs of semantic coverage. To assess similarity between graphs, we use the imperfect but simple metric of ratio between the number of common arcs over the total number of arcs. The results are shown in Table II. Interestingly, the graphs of semantic coverage have greater similarity between themselves than they have with mutation coverage.

[1]Available online at `http://www.grahamedgecombe.com/projects /jterminal`

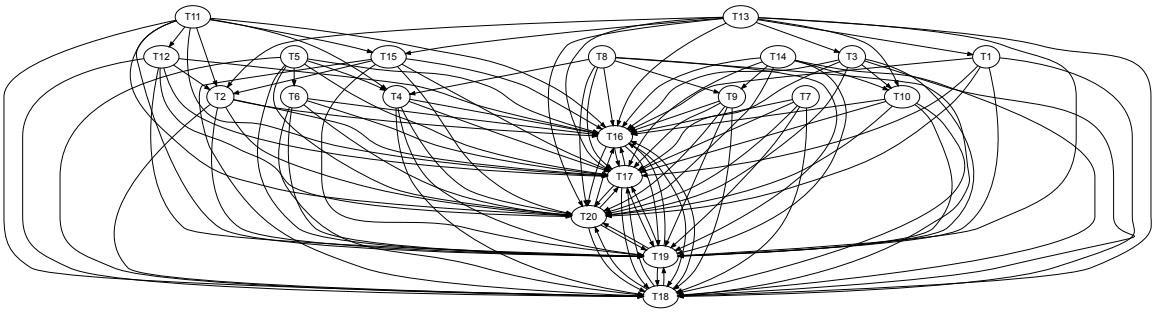


Fig. 9. Ordering Test Suites $T_i$ by Mutation Coverage (inclusion relations of killed mutant sets)

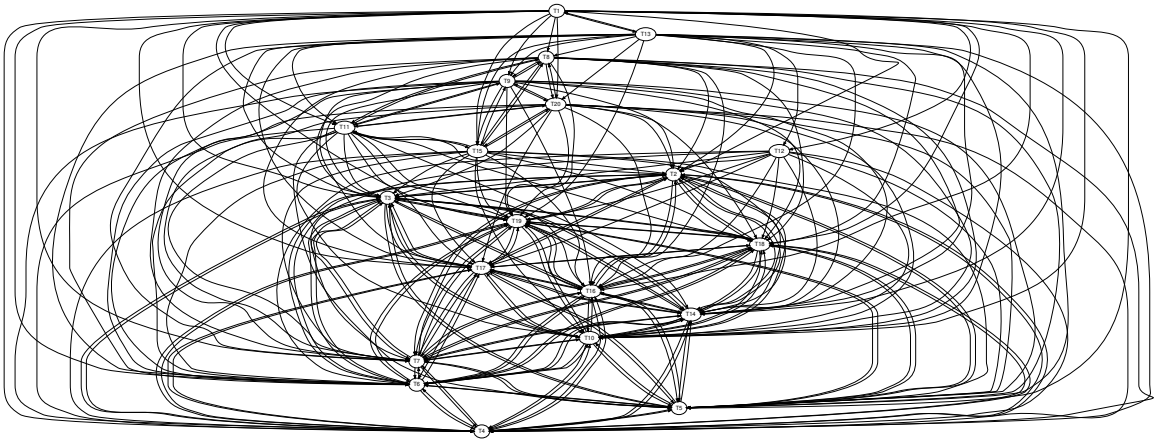

Fig. 10. Ordering Test Suites by inclusion relations of $\Gamma^{PAR}_{[M25,P]}(T_i)$

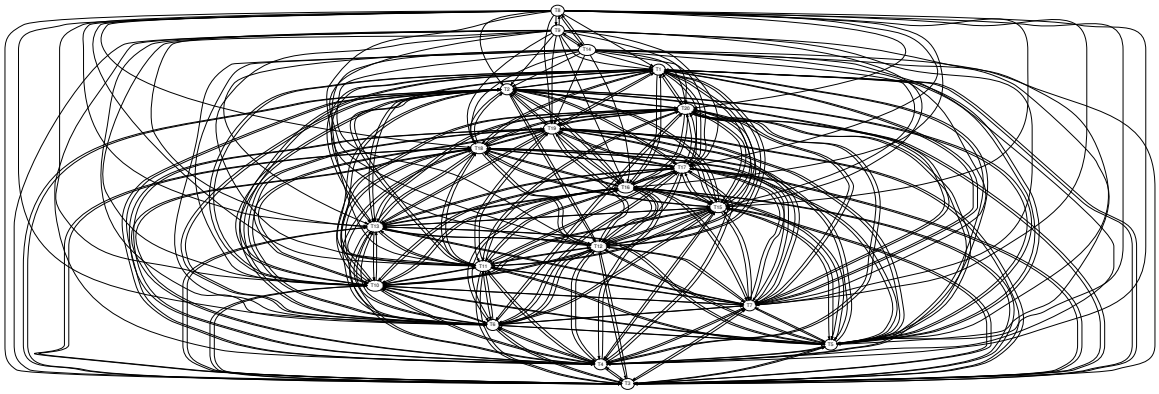

Fig. 11. Ordering Test Suites by inclusion relations of $\Gamma^{PAR}_{[M50,P]}(T_i)$

| Graph Similarity | Mutation | $\Gamma^{PAR}_{[M25,P]}(T)$ | $\Gamma^{PAR}_{[M50,P]}(T)$ | $\Gamma^{TOT}_{[M25,P]}(T)$ | $\Gamma^{TOT}_{[M50,P]}(T)$ |
|---|---|---|---|---|---|
| Mutation | 1 | 0.34 | 0.35 | 0.34 | 0.5 |
| $\Gamma^{PAR}_{[M25,P]}(T)$ | 0.34 | 1.0 | 0.66 | 1.0 | 0.46 |
| $\Gamma^{PAR}_{[M50,P]}(T)$ | 0.35 | 0.66 | 1.0 | 0.66 | 0.62 |
| $\Gamma^{TOT}_{[M25,P]}(T)$ | 0.34 | 1.0 | 0.66 | 1.0 | 0.46 |
| $\Gamma^{TOT}_{[M50,P]}(T)$ | 0.50 | 0.46 | 0.62 | 0.46 | 1.0 |

TABLE II
GRAPH SIMILARITY OF SEMANTIC COVERAGE AND MUTATION COVERAGE

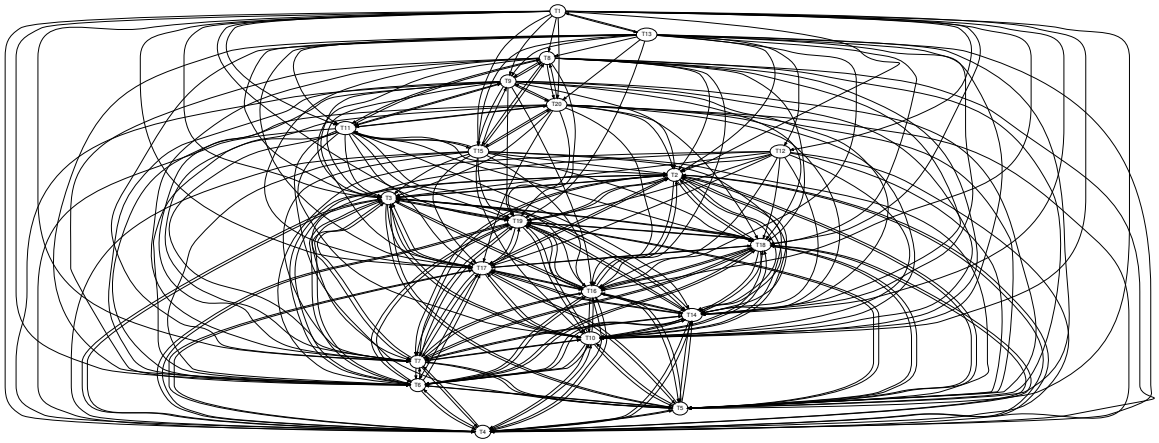

Fig. 12. Ordering Test Suites by inclusion relations of $\Gamma^{TOT}_{[M25,P]}(T_i)$

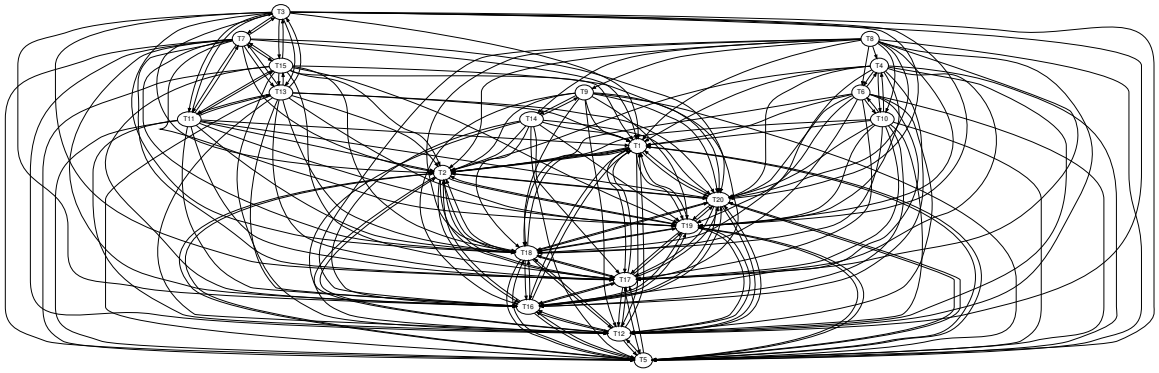

Fig. 13. Ordering Test Suites by inclusion relations of $\Gamma^{TOT}_{[M50,P]}(T_i)$

## V. Conclusion

### A. Summary and Assessment

In this paper we discuss two concepts in software testing: the *detector set* of a program $P$ with respect to a specification $R$ is the set of tests that disprove the correctness of $P$ with respect to $R$. Using detector sets, we introduce the *semantic coverage* of a test suite $T$ for (total or partial) correctness of a program $P$ with respect to specification $R$ as the union of $T$ with the complement of the detector set of $P$ with respect to $R$. In other words, the semantic coverage of a test suite $T$ is the union of the test data on which $P$ is tested ($T$) with the test data on which $P$ does not have to be tested ($\overline{\Theta(R,P)}$).

We find that the detector set of a program $P$ with respect to a specification $R$ grows larger when we transition from partial correctness to total correctness, when $P$ grows less correct with respect to $R$, and when $R$ grows more-refined. All of this makes sense when we consider that the purpose of a detector set is to disprove the correctness of $P$ with respect to $R$ (expose program failures).

On the other hand, we find that the semantic coverage of a test suite $T$ for program $P$ with respect to specification $R$ grows larger when $T$ grows larger, when we transition from total correctness to partial correctness, when $P$ grows more-correct, and when $R$ grows less-refined. All of this makes sense when we consider that semantic coverage reflects the effectiveness of a test suite to test a program for correctness against a specification. Consider that a tool is all the more effective that it is intrinsically more powerful (larger $T$) and the task on which it is deployed is easier (smaller detector set $\Theta(R,P)$).

For illustration, we compute the semantic coverage of twenty test suites of a benchmark program with respect to two sample specifications for total and partial correctness; and we compare the way semantic coverage ranks tets suites with the way mutation coverage does.

We validate the proposed formula of semantic coverage analytically by showing that it is monotonic with respect to several attributes that reflect test suite effectiveness. We also validate it empirically, by showing on a sample benchmark example that there is much similarity between the ordering derived from semantic coverage with the ordering derived from mutation coverage.

### B. Threats to Validity

The empirical validation is clearly incomplete, as it is based on a single benchmark example, but it is intended as a complement to the analytical validation, rather than a substitute thereof. The main difficulty of the proposed coverage metric is that it assumes the availability of a specification, and it is difficult to estimate in practice; but our purpose is to propose a measure of coverage that can be used for reasoning about test suites or for comparing test suites. Its spplicability for such purposes is the subject of future research.

### C. 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 [2], [5], [9], [12], [17], [18], [25]; 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 [2], [14].

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.

### D. 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 validation of our definition of semantic coverage, by comparing it to (yet to be defined/ identified) objective measures of test suite effectiveness.

## References


[1] B. Aichernig, E. Jobstl, and M. Kegele, "Incremental refinement checking for test case generation," in *Tests and Proofs*, 2013, pp. 1–19.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[3] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings, ICSE*, 2005.

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[5] T. Ball, "A theory of predicate-complete test coverage and generation," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2004, pp. 1–22.

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

[7] N. Diallo, W. Ghardallou, and A. Mili, "Correctness and relative correctness," in *Proceedings, 37th International Conference on Software Engineering, NIER track*, Firenze, Italy, May 20–22 2015.

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

[9] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–33, 2015.

[10] D. Gries, *The Science of Programming*. Springer Verlag, 1981.

[11] E. C. Hehner, *A Practical Theory of Programming*. Prentice Hall, 1992.

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

[13] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.

[14] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Procedings, 36th International Conference on Software Engineering*. ACM Press, 2014.

[15] B. Khaireddine, M. Martinez, and A. Mili, "Program repair at arbitrary fault depth," in *Proceedings, ICST 2019 Tools Track*, Xi'An, China, April 2019.

[16] X. Li, Y. Wang, and H. Lin, "Coverage based dynamic mutant subsumption graph," in *Proceedings, International Conference on Mathematics, Modeling and Simulation Technologies and Applications*, 2017.

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

[18] M. R. Lyu, J. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," *IEEE transactions on reliability*, vol. 43, no. 4, pp. 527–535, 1994.

[19] Z. Manna, *A Mathematical Theory of Computation*. McGraw-Hill, 1974.

[20] A. P. Mathur, *Foundations of Software Testing*. Pearson, 2014.

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

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

[23] A. Parsai and S. Demeyer, "Dynamic mutant subsumption analysis using littledarwin," in *Proceedings, A-TEST 2017*, Paderborn, Germany, September 4-5 2017.

[24] D. Shin, S. Yoo, and D.-H. Bae, "A theoretical and empirical study of diversity-aware mutation adequancy criterion," *IEEE TSE*, vol. 44, no. 10, October 2018.

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

[26] J. V. Wright, "A lattice theoretical basis for program refinement," Dept. of Computer Science, Åbo Akademi, Finland, Tech. Rep., 1990.