Subsumption, Correctness and Relative Correctness: Implications for Software Testing

Samia AlBlwi[1], Imen Marsit[2], Besma Khaireddine[3], Amani Ayad[4], JiMeng Loh[1], and Ali Mili[1:{0000−0002−6578−5510}]

1: NJIT, NJ, USA, 2: University of Sousse, Tunisia, 3: University of Tunis, Tunisia, 4: Kean University, NJ, USA
sma225@njit.edu, imen.marsit@gmail.com, besma.khaireddine@gmail.com, amanayad@kean.edu, loh@njit.edu, mili@njit.edu

Abstract

Context. Several Research areas emerged and have been proceeding independently when in fact they have much in common. These include: mutant subsumption and mutant set minimization; relative correctness and the semantic definition of faults; differentiator sets and their application to test diversity; generate-and–validate methods of program repair; test suite coverage metrics.
Objective. Highlight their analogies, commonalities and overlaps; explore their potential for synergy and shared research goals; unify several disparate concepts around a minimal set of artifacts.
Method. Introduce and analyze a minimal set of concepts that enable us to model these disparate research efforts, and explore how these models may enable us to share insights between different research directions, and advance their respective goals.
Results. Capturing absolute (total and partial) correctness and relative (total and partial) correctness with a single concept: detector sets. Using the same concept to quantify the effectiveness of test suites, and prove that the proposed measure satisfies appealing monotonicity properties. Using the measure of test suite effectiveness to model mutant set minimization as an optimization problem, characterized by an objective function and a constraint.

Generalizing the concept of mutant subsumption using the concept of differentiator sets. Identifying analogies between detector sets and differentiator sets, and inferring relationships between subsumption and relative correctness.
Conclusion. This paper does not aim to answer any pressing research question as much as it aims to raise research questions that use the insights gained from one research venue to gain a fresh perspective on a related research issue.

Keywords: mutant subsumption; mutant set minimization; relative correctness; absolute correctness; total correctness; partial correctness; program fault; program repair; differentiator set; detector set.

1. Introduction: Distinctions and Differences

1.1. Five Research Directions

We consider five active research directions in software engineering:

- In [34, 35] Kurtz et al. introduce the concept of mutant subsumption as a criterion for eliminating redundant mutants and minimizing mutant sets; this concepts is subsequently investigated from various angles [22, 50, 53, 37, 30, 28, 34, 55, 49], and tools are proposed to support its use [50].
- In [13] Diallo et al. introduce the concept of *relative correctness* and use it to give a semantic definition of a software fault; this concept is subsequently used by Khaireddine et al. as a basis for modeling program repair [33, 17], and for quantifying program faultiness [32].
- In [51] Shin et al. introduce the concept of *differentiator test* as a test datum that distinguishes a mutant from a base program, and use the concept to characterize and investigate test diversity in mutation testing; they also use differentiator tests, accessorily, to define *detector tests*, which disprove the correctness of a program with respect to a specification. In [42], Mili generalizes the concepts of differentiator sets and detector sets by considering the possibility of program divergence (i.e. failure to terminate normally), by considering non-deterministic specifications, and by distinguishing between partial correctness and total correctness.
- In [16], Gazzola et al. present a survey of program repair, a discipline that has garnered a great deal of attention over the past two decades, and has given rise to a stream of increasingly sophisticated tools.
- One of the most important decisions we make in software testing is the selection of test data; and one of the most important factors in this decision is the way in which we define and quantify the effectiveness of test data sets (aka *test suites*). Virtually all of the existing measures of test suite quality equate effectiveness with the test suite's ability to detect faults [18, 26, 37, 36, 10, 27, 5, 54, 1, 20, 56, 11], but we argue that there is a different (and perhaps better) criterion for test suite effectiveness: the ability to expose program failures.

1.2. Analogies and Overlaps

Even though these directions of research have emerged and evolved independently, and were driven by different research goals, they have much in common. In this paper we highlight the analogies and overlaps between these, and we explore the potential mutual insights that they offer, as well as the research questions that they raise. In particular, we show that these research areas can all be modeled by two simple and related concepts: *Detector Set*, i.e. the set of program inputs that disprove the correctness of a program with respect to a specification; *Differentiator Set*, i.e. the set of program inputs that disprove the semantic equivalence of two programs.

- *Detector Sets and Correctness.* Given a program P and a specification R, the detector set of P with respect to R is the set of inputs x for which execution of P on x disproves the correctness of P with respect to R. Since there are two standards of correctness (partial and total [21, 40]), we introduce two versions of detector sets: detector sets for partial correctness, and detector sets for total correctness. Also, we use detector sets to define two properties: *Absolute Correctness*, the property of a program to be correct with respect to a specification, and *Relative Correctness*, the property of a program to be more-correct than another with respect to a specification. This is the subject of section 3.

- *Differentiator Sets and Subsumption.* Given two programs P and Q, the differentiator set of P and Q is the set of inputs x such that executions of P and Q on x yield different outcomes. Though it may sound straightforward, this definition depends actually on what we consider to be the outcome of an execution, under what condition we consider that two execution outcomes are comparable, and under what condition we consider that two comparable outcomes are distinct: If a program execution fails to terminate or aborts due to an illegal operation (e.g. division by zero), do we consider that it has no outcome or that failure to terminate normally is itself an outcome? How do we compare outcomes in the presence of such possibilities? In this paper we introduce three definitions of differentiator sets, that depend on how we answer these questions; these yield three different definitions of what it means to kill a mutant, hence three different definitions of mutant subsumption. The possibility of divergence (failure to terminate, failure to avoid aborts, etc) is all the more important in the study of mutation testing that several mutation operators are prone to cause divergence even where the base program excutes normally. This is the subject of section 4.

- *Density of Subsumption Graphs.* Using differentiator sets, we can model mutant subsumption as an inclusion relation between their differentiator sets. This, in turn, enables us to derive a statistical model that estimates the probability that any two mutants are in a subsumtion relation by analyzing the probability that two random non-empty subsets of a given set are in an inclusion relation. This model enables us, in turn, to estimate the number of arcs in a subsumption graph, as well as the number of mutants that are maximal by subsumption; this number is important because it represents the size of the minimal mutant set. This is the subject of section 4.4.

- *Subsumption as Relative Correctness.* By considering how relative correctness [43, 13] can be characterized by means of detector sets, and subsumption [34, 35] can be characterized by means of differentiator sets, we find that these two properties are actually equivalent. This is interesting, given that these properties were introduced independently, albeit simultaneously (in 2014), and have so far evolved separately. Given that subsumption is introduced in the context of mutation testing and relative correctness is introduced to define software faults, this relationship can be seen through the lens of the ongoing debate on whether mutants are faithful representatives of faults [5, 4, 48, 29]. This is the subject of section 5

- *Defining Test Suite Effectiveness.* Of all the decisions we make in softwqare testing, none is as consequential as the selection of test data; the selection of test data is, in turn, profoundly influenced/ determined by the criteria that we use to assess test suite effectiveness, whence the way in which we compare candidate test suites. We consider two possible criteria for assessing the effectiveness of a test suite:

  – The ability to detect faults, or

  – The ability to expose failures.

  The existence of these two criteria raises three questions:

  – Are these two criteria equivalent? if not,

  – Are these two criteria statistically correlated? if not,

  – Is one of them better than the other, for the purposes of test data selection?

  A fourth question may be worth considering: which of these two criteria are current measures reflecting? In section 6 we discuss the concept of *semantic coverage*, which equates the effectiveness of a test suite with its ability to expose the failures of a program, and analyze its attributes.

- *Defining Mutant Set Effectiveness.* To quantify the effectiveness of a mutant set we start by pondering the question: What is the purpose of a mutant set? If we postulate that the purpose of a mutant set is to vet test suites, then the effectiveness of a mutant set can be quantified as a function of the effectiveness of the test suites that it vets; since we quantify the effectiveness of a test suite by its semantic coverage, we resolve to quantify the effectiveness of a mutant set by the semantic coverage of the test suites that it vets. This is the subject of section 7.

- *Mutant Set Minimization as an Optimization Problem.* Subsumption was introduced as a means to minimize the cardinality of a mutant set [34], without reducing its effectiveness to vet test suites. Mutant set minimization is essentially an optimization problem; as such, it ought to be defined by two parameters, namely the objective function to minimize, and the constraint under which this minimization is attempted. While the published literature is clear about the objective function (re: the cardinality of the mutant set), it does not include an explicit definition of the optimization constraint. In section 7.2, we use the definition of mutant set effectiveness of section 7 to formulate the constraint of mutant set minimization as the condition that the minimal mutant set has the same effectiveness as the original set; we show that removing a subsumed mutant has no impact on the effectiveness of the mutant set, as defined.

Figure 1 shows an outline of this paper: In section 2 we introduce some elementary relational mathematics that we use in the remainder of the paper. In section 3 we introduce detector sets and use them to define four correctness properties: absolute partial correctness, absolute total correctness, relative partial correctness and relative total correctness. In section 4 we introduce differentiator sets and use them to define three subsumption properties, which vary according to how we interpret the outcome of a program execution: Basic subsumption, Strict subsumption, and Broad subsumption. In section 5 we show that some forms of relative correctness are equivalent to some forms of subsumption and in section 6 we use detector sets to define a measure of test suite effectiveness. In section 7 we use the results of sections 4 and 6 to recast mutant set minimization as an authentic optimization problem: we do so by presenting its objective function and the constraints under which this objective function is optimized. In section 8 we summarize our results, critique them, and discuss future research directions.

2. Relational Mathematics

2.1. Sets

We represent sets by C-like variable declarations; if we define a set S by the variable declarations:

```
xType x; yType y;
```

then S is the cartesian product of the sets of values that the types `xType` and `yType` represent; elements of S are denoted by lower case s, and are referred to as *states*. Given an element s of S, we may refer to the x-component (resp. y-component) of s as $x(s)$ (resp. $y(s)$). But we may, for the sake of convenience, refer to the x component of states s, s', s'' (e.g.) simply as x, x', x''.

2.2. Operations on Relations

A relation on set S is a subset of the cartesian product $S \times S$; special relations on set S include the *universal relation* $L = S \times S$, the *identity relation* $I = \{(s,s')|s \in S \wedge s' = s\}$ and the *empty relation* $\phi = \{\}$. Operations on relations include the set theoretic operations of union ($\cup$), intersection ($\cap$), difference($\setminus$) and complement ($\overline{R} = L \setminus R$). They also include the *product* of two relations, denoted by $R \circ R'$ (or RR', for short) and defined by

$$R \circ R' = \{(s,s')|\exists s" : (s,s") \in R \wedge (s",s') \in R'\}.$$

The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$. The *domain* of relation R is denoted by $dom(R)$ and defined by $dom(R) = \{s|\exists s' : (s,s') \in R\}$. The *pre-restriction* of relation R to set T is the relation denoted by ${}_{T\setminus}R = \{(s,s')|s \in T \wedge (s,s') \in R\}$.

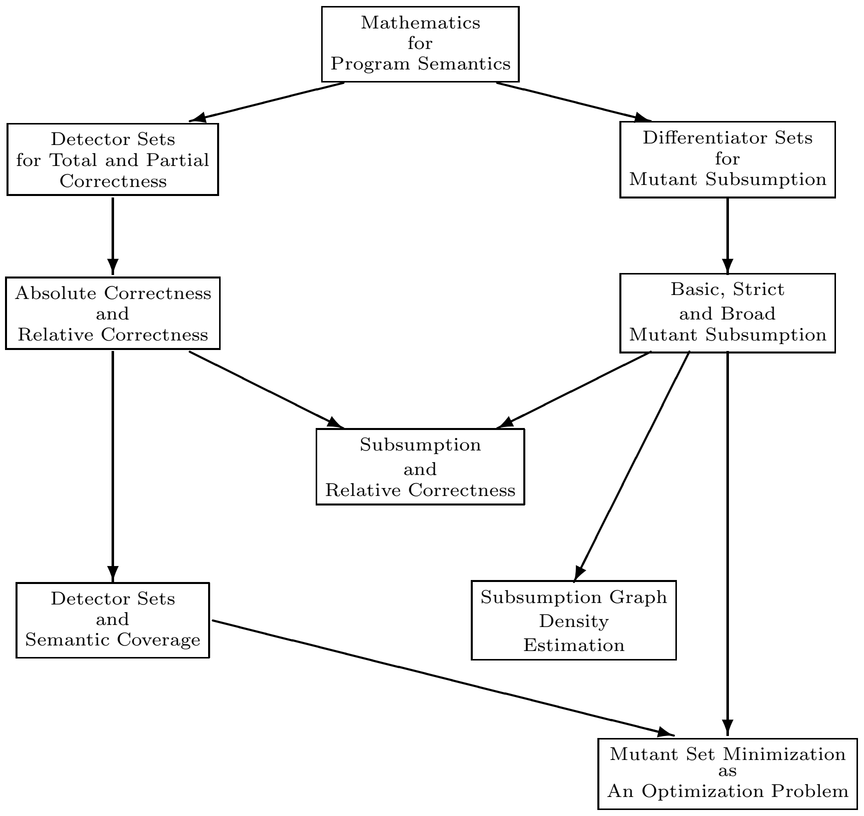

Mathematics for Program Semantics
Detector Sets for Total and Partial Correctness
Differentiator Sets for Mutant Subsumption
Absolute Correctness and Relative Correctness
Basic, Strict and Broad Mutant Subsumption
Subsumption and Relative Correctness
Detector Sets and Semantic Coverage
Subsumption Graph Density Estimation
Mutant Set Minimization as An Optimization Problem

Figure 1: Paper Outline

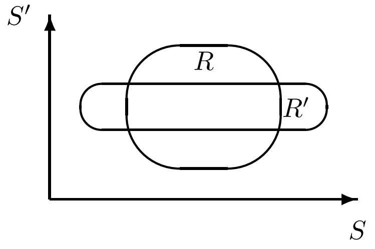


Figure 2: R' refines R: $R' \sqsupseteq R$, $R \sqsubseteq R'$.

2.3. Properties of Relations

A relation R is said to be *reflexive* if and only if $I \subseteq R$; relation R is said to be *symmetric* if and only if $R = \widehat{R}$; relation R is said to be *transitive* if and only if $RR \subseteq R$; relation R is said to be *asymmetric* if and only if $R \cap \widehat{R} = \phi$; relation R is said to be antisymmetric if and only if $R \cap \widehat{R} \subseteq I$. A relation R is said to be an *equivalence relation* if and only if it is reflexive, symmetric and transitive. A relation R is said to be a *partial ordering* if and only if it is reflexive, transitive and antisymmetric. A relation R is said to be a *strict partial ordering* if and only if it is transitive and asymmetric.

A relation 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 *total* if and only if $RL = L$. A relation R is said to be a *vector* if and only if $RL = R$; a vector V on set S is a relation of the form $V = A \times S$ for some non-empty subset A of S. We may use vectors to define pre-restrictions and post-restrictions: Given a relation R and a vector $V = A \times S$, the pre-restriction of R to A can be written as $V \cap R$ and the post-restriction of R to A can be written $R \cap \widehat{V}$. A relation R' is said to *refine* a relation R if and only if

$$RL \cap R'L \cap (R \cup R') = R;$$

this is denoted by $R' \sqsupseteq R$ or $R \sqsubseteq R'$. The following Proposition, which we present without proof, gives an equivalent formulation of refinement.

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

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 ($RL \cap R' \subseteq R$) [23, 47, 21, 15]. Figure 2 illustrates the property of refinement.

3. Detector Sets and Correctness

3.1. Program Functions

We can define the semantics of a program by means of a function from an input space to an output space, or by means of a function from initial states to final states. For the sake of simplicity, we adopt the latter model, as it enables us to work with homogeneous relations, without loss of generality. This model encompasses the case where we want to think of a program as mapping an input space into an output space: It suffices to add to the state space of the program a variable that represents the input stream (in the parlance of C++) and a variable that represents the output stream. Then the mapping that the program defines from inputs to outputs is simply the set of pairs of the form (is, os') where is is the value of the input stream in the initial state, and os' is the value of the output stream in the final state. Hence focusing on the (initial state, final state) model causes no loss of generality.

We consider a program P on space S; execution of P on an initial state s may terminate in a final state s' after a finite number of steps; conversely, it may enter an infinite loop or attempt an illegal operation such as a division by zero, an array reference out of bounds, reference to a nil pointer, the square root of a negative number, the log of a non-positive number, etc. When execution of P on s terminates normally in a final state s', we say that it *converges* on s, else we say that it *diverges*.

Given a program P on space S, the *function* of program P, which we also denote by P, is the set of pairs (s, s') such that if execution of P starts in state s, it converges in final state s'. Consequently, the domain of P is the set of initial states on which execution of P converges.

3.2. Absolute Correctness

Absolute correctness is a property between a program and a specification; we discuss it in this section. Following decades-old tradition, we distinguish between two forms of program correctness: total correctness and partial correctness [40]. A specification on space S is a relation on S. For the sake of our discusions herein, we consider that programs are deterministic, but specifications may be non-deterministic. The following definition, due to [46], introduces absolute total correctness.

Definition 1. *Program P on space S is said to be* totally correct *with respect to specification R on S if and only if:*

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

The domain of $(R \cap P)$ is called the *competence domain* of P with respect to R; it is the set of initial states for which P behaves according to R. Figure 3 shows a simple example of a (non-deterministic) specification R and two programs P and P' such that P is correct with respect to R and P' is not; the competence domains of P and P' are shown by the ovals. Even though it looks different, this definition is equivalent, modulo differences in notation, with traditional definitions of total correctness [40, 21, 23]. We present a brief argument to this effect: Given that $dom(R \cap P) \subseteq dom(R)$ is a set theoretic tautology, the condition of Definition 1 is equivalent to:

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

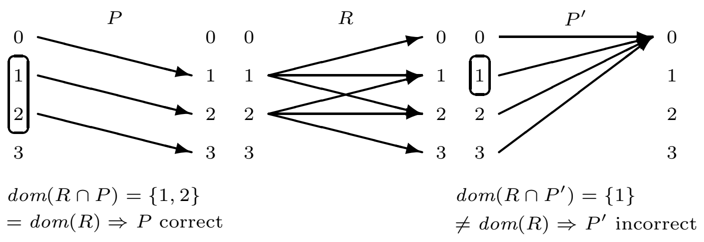

P
R
P'
0
1
2
3
$dom(R \cap P) = \{1, 2\}$
$= dom(R) \Rightarrow P$ correct
$dom(R \cap P') = \{1\}$
$\neq dom(R) \Rightarrow P'$ incorrect

Figure 3: Total Correctness

By set theory, this can be interpreted as:
$\forall s : s \in dom(R) \Rightarrow s \in dom(R \cap P)$.
By Definition of domain, this can be written as:
$\forall s : s \in dom(R) \Rightarrow (\exists s' : (s, s') \in (R \cap P)$.
Since P is deterministic, we can replace s' by $P(s)$, hence:
$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in R$.
If we interpret:

- $s \in dom(R)$ as: s satisfies the precondition implied by R,
- $s \in dom(P)$ as: execution of P on s converges,
- $(s, P(s)) \in R$ as: $P(s)$ satisfies the postcondition implied by R,

then we find that this is exactly the traditional definition of total correctness [40, 21, 15].

The following definition, due to [44], mimics the style of Definition 1, to define partial correctness.

Definition 2. *We say that P is* partially correct *with respect to R if and only if:*

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

A similar argument to what we offered above can establish that our definition is equivalent to traditional definitions of partial correctness [25, 15, 40, 21]. See Figure 4: Program Q is partially correct with respect to R because for any initial state of $dom(R)$ for which it converges, program Q delivers a final state that satisfies specification R; by contrast, program Q' is not partially correct with respect to R, even though it terminates normally for all initial states in $dom(R)$, because it does not satisfy specification R; neither Q nor Q' is totally correct with respect to R.

Whereas the distinction between partial correctness and total correctness has been a key feature of the program correctness literature [40, 21, 15, 23], it has not been given much consideration in software testing. Yet, testing a program for partial correctness is different from testing it for total correctness, as we discuss below:

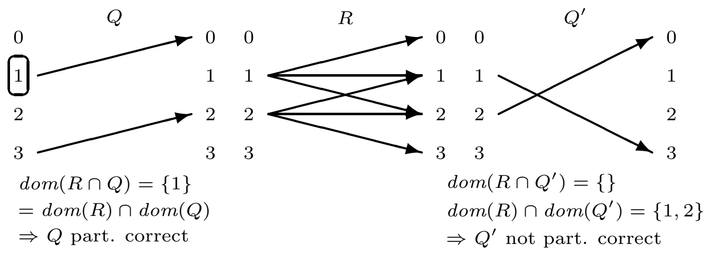

Q
R
Q'
$dom(R \cap Q) = \{1\}$
$= dom(R) \cap dom(Q)$
$\Rightarrow Q$ part. correct
$dom(R \cap Q') = \{\}$
$dom(R) \cap dom(Q') = \{1, 2\}$
$\Rightarrow Q'$ not part. correct

Figure 4: Partial Correctness

- *Interpreting Test Outcomes.* If we test a program P for correctness with respect to specification R on some initial state $s \in dom(R)$ and P fails to converge on s, then what conclusion we draw depends on whether we are testing P for total correctness or for partial correctness. Under total correctness we conclude that P fails on s; under partial correctness we conclude that s is a wrong test.

- *Test Data Selection.* The essence of test data selection is to approximate an infinite (or prohibitively large) input domain by a finite and small subset, which serves as a proxy thereof. If we are testing a program P for total correctness with respect to R, the infinite set we are trying to approximate is $dom(R)$; for partial correctness, the set we are trying to approximate is $dom(R) \cap dom(P)$.

3.3. Detector Sets

Given a program P on space S and a specification R on S, the detector set of P with respect to R is the set of initial states in S which disprove the correctness of P with respect to R; given that there are two standards of correctness (partial and total), we get two versions of detector sets.

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

- *The* detector set of P with respect to R for total correctness *is denoted by $\Delta^{TOT}(R, P)$ and defined by:*
  $\Delta^{TOT}(R, P) = dom(R) \cap \overline{dom(R \cap P)}$.

- *The* detector set of P with respect to R for partial correctness *is denoted by $\Delta^{PAR}(R, P)$ and defined by:*
  $\Delta^{PAR}(R, P) = dom(R) \cap dom(P) \cap \overline{dom(R \cap P)}$.

Figure 1 shows in red the detector sets of program P with respect to specification R as a function of $dom(P)$, $dom(R)$ and $dom(R \cap P)$. Since total correctness is a stronger property than partial correctness, it is a harder property to prove, hence an easier property to disprove: Indeed, the detector set of P for total correctness is a superset of the detector set of P for partial correctness (because it is a larger set, it offers more opportunities to disprove total correctness than

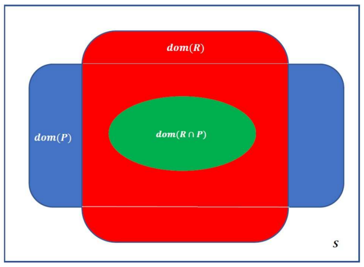
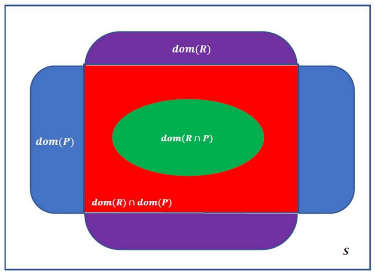
	
Total Correctness: $\Delta^{TOT}(R, P)$	Partial Correctness: $\Delta^{PAR}(R, P)$

Table 1: Detector Sets for Total and Partial Correctness

partial correctness). Detector sets are useful and relevant when we discuss program testing; the simple Propositions below show that they are useful when we discuss program correctness verification as well.

Proposition 2. *Program P is totally correct with respect to specification R if and only if its detector set with respect to R for total correctness if empty.*

Proof. Necessity stems trivially from Definition 1. Sufficiency can be proved by observing that whenever an intersection of two sets is empty, each set is a subset of the complement of the other. From $dom(R) \cap \overline{dom(R \cap P)} = \emptyset$, we infer $dom(R) \subseteq dom(R \cap P)$; this in conjunction with the tautology $dom(R \cap P) \subseteq dom(R)$, yields that P is totally correct with respect to R. **qed**

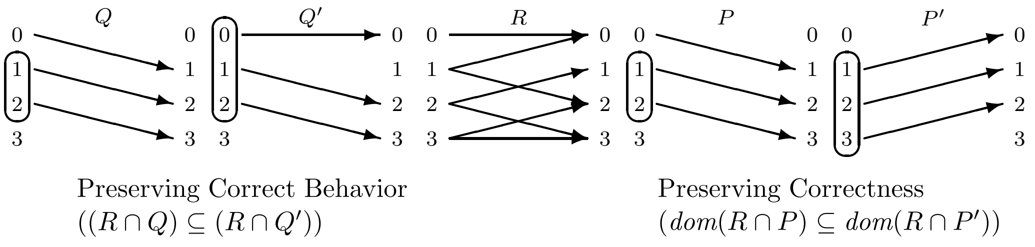
A similar proof yields the following Proposition, which we present without proof.

Proposition 3. *Program P is partially correct with respect to specification R if and only if its detector set with respect to R for partial correctness if empty.*

Of course, if and only if a program is correct, the set of inputs that disprove its correctness ought to be empty.

3.4. Relative Correctness

Whereas correctness is a property that involves a specification and a program, relative correctness involves a specification, say R, and two candidate programs, say P and P', and ranks P and P' according to how close they are to being correct. The following definition introduces the concept of relative correctness. Since to be correct means to have an empty detector set (per Propositions 2 and 3), it is natutal to define relative correctness by means of inclusion of detector sets; whence the following definitions.



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

Figure 5: Relative Total Correctness

Definition 4. *Given a specification R on space S and two programs P and P' on S, we say that P' is* more-totally-correct *than P with respect to R if and only if:*

$$\Delta^{TOT}(R, P') \subseteq \Delta^{TOT}(R, P).$$

Definition 5. *Given a specification R on space S and two programs P and P' on S, we say that P' is* more-partially-correct *than P with respect to R if and only if:*

$$\Delta^{PAR}(R, P') \subseteq \Delta^{PAR}(R, P).$$

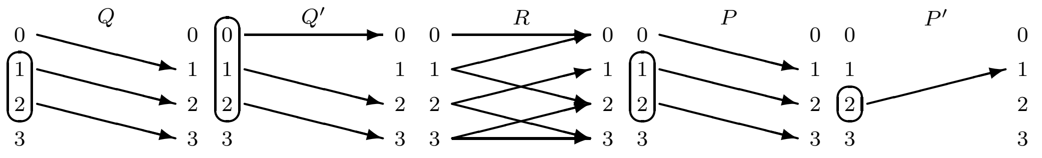
Figure 5 illustrates relative total correctness by showing a specification (R) and two sets of programs: Q' is more-correct than Q with respect to R by virtue of imitating the correct behavior of Q; P' is more-correct than P with respect to R by virtue of a different correct behavior. Relative total correctness culminates in absolute total correctness in the following sense: a totally correct program is more-totally-correct than any candidate program. Figure 6 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 fails to converge evades accountability, and is considered partially correct, by default).

Note the following relation between the detector sets of a program P with respect to a specification R:

$$\Delta^{PAR}(R, P) = dom(P) \cap \Delta^{TOT}(R, P).$$

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

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



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

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

Figure 6: Relative Partial Correctness

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

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 7 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, to allow interested readers to check Figure 7. 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, hence their detector set is all of $dom(R)$).

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, as shown in Figure 7.

Table 4 summarizes and organizes the definitions of correctness to help contrast them. In [43, 14], total relative correctness is defined, not by comparing

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

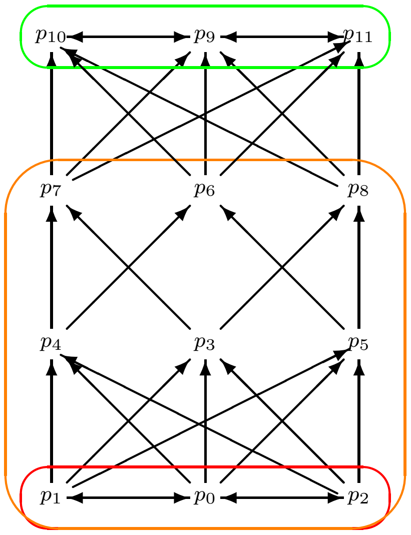


Figure 7: Ordering Candidate Programs by Relative Total (and Partial) Correctness with Respect to R

	Partial Correctness	Total Correctness
Absolute Correctness : P is correct iff:	$\Delta^{PAR}(R,P)=\emptyset$	$\Delta^{TOT}(R,P)=\emptyset$
Relative Correctness : P' is more-correct than P iff:	$\Delta^{PAR}(R,P')\subseteq\Delta^{PAR}(R,P)$	$\Delta^{TOT}(R,P')\subseteq\Delta^{TOT}(R,P)$

Table 4: Definitions of Correctness by Means of Detector Sets

detector sets, as we do in Definition 4, but by comparing competence domains. The following Proposition provides that these definitions are equivalent.

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

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

Proof. Sufficiency can be inferred readily by inverting the inequality (and complementing both sides) then taking the intersection with $dom(R)$ on both sides. Necessity can be proved by using the set theoretic identity to the effect that the two conditions below are equivalent:

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

This lemma can be proved by inverting the first inequality, applying DeMorgan's laws, then taking the intersection with A on both sides. From the hypothesis:
$dom(R) \cap \overline{dom(R \cap P')} \subseteq dom(R) \cap \overline{dom(R \cap P)}$
we infer (by the lemma above):
$dom(R) \cap dom(R \cap P) \subseteq dom(R) \cap dom(R \cap P')$,
which we simplify into:
$dom(R \cap P) \subseteq dom(R \cap P')$,
since $dom(R \cap P)$ and $dom(R \cap P')$ are both subsets of $dom(R)$. **qed**

3.5. Detector Sets and Oracles

Given a program P on space S and a specification R on S, we ponder the question of what test oracle do we use if we want to test program P for correctness with respect to R. We consider the following framework:

```
{Stype s, s0;  // current state, initial state
 read(s);  s0=s;  // read state, save it in s0
 P();          // modifies s, keeps s0 intact
 if (not oracle(so,s)) {testfailure(s0);}}
```

The question we raise is: how should predicate `oracle(,)` be defined, if we are interested to test P for correctness with respect to R? Of course, the answer depends on whether we are talking about partial correctness or total correctness.

Definition 6. *Due to Khaireddine et al [33]. Given a program P on space S and a specification R on S, the* oracle for total correctness *of Program with respect to specification R is denoted by $\Omega^{TOT}_{[R,P]}(s, s')$ and defined by:*

$$\Omega^{TOT}_{[R,P]}(s, s') \equiv \neg s \in dom(R) \vee (s, s') \in R.$$

Khaireddine et al. prove in [33] that if and only if $\Omega^{TOT}_{[R,P]}(s, P(s))$ returns TRUE for all elements s in T then P is totally correct with respect to the pre-restriction of R to T, ${}_{T\backslash}R$. The following Proposition links the links the oracle of total correctness with the detector set of total correctness.

Proposition 5. *Given a program P on space S and a specification R on S, an element s of S is in the detector set for total correctness of P with respect to R if and only if the oracle for total correctness of P with respect to R returns FALSE for the pair $(s, P(s))$:*

$$\forall s \in S : s \in \Delta^{TOT}_{[R,P]} \Leftrightarrow \neg\Omega^{TOT}_{[R,P]}(s, P(s)).$$

Proof. This Proposition stems readily from the Definitions:

$s \in \Delta^{TOT}_{[R,P]}$

$\Leftrightarrow$ {Definition}

$s \in dom(R) \wedge s \notin dom(R \cap P)$

$\Leftrightarrow$ {Determinacy of P}

$s \in dom(R) \wedge s \not(s, P(s)) \in R$

$\Leftrightarrow$ {De Morgan}

$\neg(\neg s \in dom(R) \vee (s, P(s)) \in R)$

$\Leftrightarrow$ {Definition}

$\neg\Omega^{TOT}_{[R,P]}(s, P(s)).$ **qed**

Notwithstanding the issue of how we determine that a program fails to terminate, we can define the oracle of partial correctness as follows:

$$\Omega^{PAR}_{[R,P]}(s, s') \equiv s \notin dom(P) \vee \Omega^{TOT}_{[R,P]}(s, s').$$

Then we can easily prove a result analogous to Proposition 5.

$$\forall s \in S : s \in \Delta^{PAR}_{[R,P]} \Leftrightarrow \neg\Omega^{PAR}_{[R,P]}(s, P(s)).$$

4. Differentiator Sets and Mutant Subsumption

Whereas in the previous section we discuss detector sets and their relationship to relative correctness, in this section we discuss differentiator sets and their relationship to mutant subsumption.

4.1. Execution Outcomes

The differentiator set of two programs P and Q on space S is the set of initial states for which execution of P and execution of Q yield distinct outcomes. Whenever P and Q both converge for some initial state s, then their outcomes are the final states, $P(s)$ and $Q(s)$ that they yield; determining whether they have the same outcome amounts to checking $P(s)$ and $Q(s)$ for equality. But if one or both programs diverge, determining whether they have the same outcome or different outcomes becomes less clear-cut, more debatable. Given the possibility of divergence, we must consider the following questions, on which the definition of differentiator set depends:

Space S	**Program** P	**Mutation**	**Divergence**
`int i, x;`	`i=100; x=0;` `while(i!=0) {x=x+1; i=i-1;}`	`i-1` → `i+1`	Failure to Terminate
`int a[100], x;` `int i;`	`i=0; x=0.0;` `while (i<100) {x=x+a[i];i=i+1;}`	`<` → `<=`	Array Reference Out of Bounds
`int x;` `float y;`	`x=100; y=0;` `while (x>0) {y=y+1./x; x=x-1;}`	`>` → `>=`	Division By Zero
`float x,y;`	`cin >> x;` `if (x>=0) {y=sqrt(1+x);}` `else {y=sqrt(1-x);}`	`1-x` → `1+x`	Illegal Arithmetic Operation

Table 5: Examples of Mutation Operations Causing Divergence

- *What is the outcome of a program's execution*? In particular, is divergence an outcome or the absence of an outcome?
- *When are two outcomes comparable*? In particular, is divergence comparable to the outcome of a program that converges?
- *When are two comparable outcomes identical or distinct*? In particular, is divergence a different outcome from any convergent outcome? Are two divergent outcomes identical or incomparable?

How we define differentiator sets depends on how we answer these questions. Though these questions may sound like mundane academic exercises, we argue that divergence is in fact a common occurrence in mutation testing; indeed many mutation operators are prone to cause divergence even when the base program converges; this includes mutation operators that are applied to guards that programmers routinely include to avoid illegal operations. Table 5 shows examples of common mutation operators which cause common program patterns to diverge.

4.2. Differentiator Sets

The differentiator set of two programs P and Q on space S is the set of initial states s such that the execution of programs P and Q on s yields different outcomes [51, 42]. In light of the foregoing discussions, we adopt three definitions of differentiator sets, which reflect three sensible interpretations of what it means for two program executions to yield distinct outcomes.

- *Basic Interpretation.* We assume that programs P and Q converge for all initial states in S, and their outcome is their final state. Their *basic differentiator set* (which we denote by $\delta_0(P, Q)$) is the set of initial states for which their final states are distinct.
- *Strict Interpretation.* We do not assume that P and Q converge for all initial states, but we restrict their differentiator set to those initial states

for which they both converge and produce distinct outcomes; we denote their *strict differentiator set* by $\delta_1(P, Q)$.

- *Broad Interpretation.* We do not assume that P and Q converge for all initial states, but we restrict their differentiator set to those initial states for which they both converge and produce distinct outcomes along with the initial states for which only one of them converges; we assume that a program that diverges has a different outcome from a program that converges, regardless of the final state of the latter; we denote the *broad differentiator set* by $\delta_2(P, Q)$.

The following definition gives explicit formulas of differentiator sets under the three interpretations given above. To understand these definitions, it suffices to note the following:

- The set of initial states for which program P (resp. Q) converges is $dom(P)$ (resp. $dom(Q)$).
- The set of initial states for which the final states of P and Q are identical is $dom(P \cap Q)$.
- The following inequalities hold by set theory:
  $dom(P \cap Q) \subseteq dom(P) \cap dom(Q) \subseteq dom(P) \cup dom(Q)$.

Definition 7. *The definition of a* differentiator set *of two programs P and Q depends on how we define the outcome of a program, under what condition we consider that two outcomes are comparable, and under what condition we consider that two comparable outcomes are identical or distinct.*

- *The* basic differentiator set *of two programs P and Q is defined as:*

$$\delta_0(P, Q) = \overline{dom(P \cap Q)}.$$

- *The* strict differentiator set *of two programs P and Q is defined as:*

$$\delta_1(P, Q) = dom(P) \cap dom(Q) \cap \overline{dom(P \cap Q)}.$$

- *The* broad differentiator set *of two programs P and Q is defined as:*

$$\delta_2(P, Q) = (dom(P) \cup dom(Q)) \cap \overline{dom(P \cap Q)}.$$

Figure 8 illustrates the three definitions of differentiator sets (represented in red in each case). Whenever we want to refer to a differentiator set of programs P and Q without specifying the interpretation, we use the notation $\delta(P, Q)$. Note that having three different definitions of differentiator sets means that we now have three distinct definitions of what it means to kill a mutant:

- Test suite T kills mutant M of program P in the *basic sense* if and only if: $T \cap \delta_0(P, M) \neq \emptyset$.

- Test suite T kills mutant M of program P in the *strict sense* if and only if: $T \cap \delta_1(P, M) \neq \emptyset$.
- Test suite T kills mutant M of program P in the *broad sense* if and only if: $T \cap \delta_2(P, M) \neq \emptyset$.

For illustration of differentiator sets under the basic interpretation, we consider space S defined by a single integer variable, and we consider two programs that converge for all initial states:

```
P:  {s=pow(s,4)+35*s*s+24;}
Q:  {s=10*pow(s,3)+50*s;}
```

The functions of these programs are:
$P = \{(s, s')|s' = s^4 + 35s^2 + 24\}$.
$Q = \{(s, s')|s' = 10s^3 + 50s\}$.
Their intersection is:
$P \cap Q = \{(s, s')|s^4 + 35s^2 + 24 = 10s^3 + 50s \wedge s' = s^4 + 35s^2 + 24\}$.
The domain of their intersection is:
$\mathit{dom}(P \cap Q) = \{s|s^4 + 35s^2 + 24 = 10s^3 + 50s\}$.
Solving this equation in the fourth degree, we find:
$\mathit{dom}(P \cap Q) = \{s|1 \leq s \leq 4\}$.
Taking the complement, we find:
$\delta_0(P, Q) = \overline{\mathit{dom}(P, Q)} = \{s|s < 1 \vee s > 4\}$.
For illustration of differentiator sets under the strict and broad interpretation, we consider the following programs P and Q on space S defined by an integer variable s, where `abort()` is a program that diverges for any execution, such as {`int x=1/0.;`} or {`while (true) {}`}.

```
P:  {if (s<0) {abort();}
     else {s=pow(s,4)+35*s*s+24;}}
Q:  {if (s>5) {abort();}
     else {s=10*pow(s,3)+50*s;}}
```

Note that P fails to converge for all s less than zero (since it enters an infinite loop) and Q fails to converge for all s greater than 5 (for the same reason). The functions of these programs are:
$P = \{(s, s')|s \geq 0 \wedge s' = s^4 + 35s^2 + 24\}$.
$Q = \{(s, s')|s \leq 5 \wedge s' = 10s^3 + 50s\}$.
From these definitions, we compute the following parameters:
$\mathit{dom}(P) = \{s|s \geq 0\}$.
$\mathit{dom}(Q) = \{s|s \leq 5\}$.
$P \cap Q = \{(s, s')|0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s \wedge s' = 10s^3 + 50s\}$.
$\mathit{dom}(P \cap Q) = \{s|0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s\}$.
By solving the equation $(s^4 + 35s^2 + 24 = 10s^3 + 50s)$, we find:
$\mathit{dom}(P \cap Q) = \{s|1 \leq s \leq 4\}$.
Whence:
$\delta_1(P, Q) = \{0, 5\}$.
$\delta_2(P, Q) = \{s|s \leq 0 \vee s \geq 5\}$.
Interpretation:

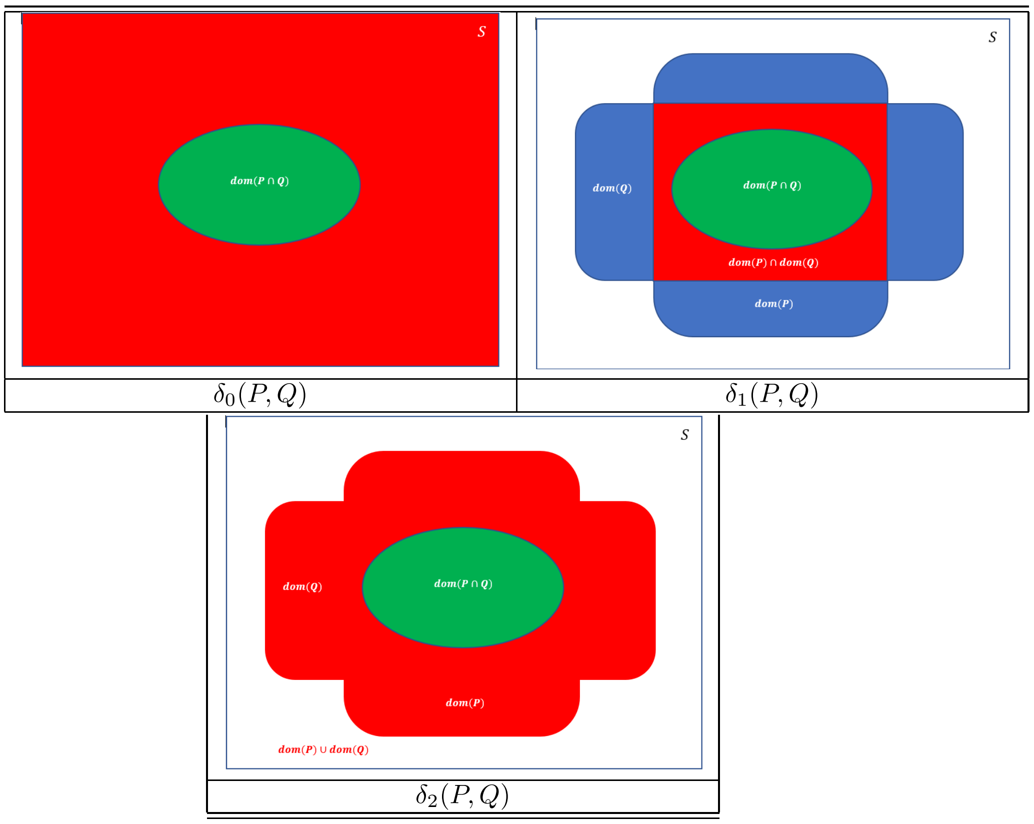


Figure 8: Three Definitions of Differentiator Sets (shown in red)

- *Strict Differentiator Set.* The set of initial states that expose the difference between P and Q is $\{0,5\}$ because the interval $[0..5]$ includes all the initial states where both P and Q are defined ($dom(P) \cap dom(Q)$), and programs P and Q return the same results for initial states in the interval $[1..4]$ ($dom(P \cap Q)$).

- *Broad Differentiator Set.* Any initial state outside the interval $[1..4]$ exposes the difference between P and Q, either because they both converge but give different results (for initial states 0 and 5) or because one of them converges while the other diverges (for s greater than 5, P converges but Q does not; for s negative, Q converges but P does not).

As further illustration of differentiator sets, we consider the space S defined by a single variable s of type integer, and we consider the following programs:

P `{s=pow(s,5)+pow(s,4)+9;}`.

Q `{s=2*pow(s,5)+pow(s,4)-5*pow(s,3)+4*s+9;}`.

Assuming perfect arithmetic, these two programs are total on space S; their competence domain is:

$dom(P \cap Q)$

$=$ {substitutions}

$\{s|s^5+s^4+9=2s^5+s^4-5s^3+4s+9\}$

$=$ {simplification}

$\{s|s^5-5s^3+4s=0\}$

$=$ {factoring}

$\{s|s(s-1)(s-2)(s+1)(s+2)=0\}$

$=$ {simplification}

$\{-2,-1,0,1,2\}$.

Hence the basic differentiator set of P and Q is:

$$\delta_0(P,Q)=\{s|s<-2 \vee s>2\}.$$

See Figure 9 for illustration.

To illustrate strict and broad differentiator sets, we consider non-total versions of P and Q:

P'
```
{if (s<-10 || s>5) {abort();}
else {s=pow(s,5)+pow(s,4)+9;}}.
```

Q'
```
{if (s<-5 || s>10) {abort();}
else{s=2*pow(s,5)+pow(s,4)-5*pow(s,3)+4*s+9;}}.
```

The functions of these programs are given as:

$$P'=\{(s,s')|-10 \leq s \leq 5 \wedge s'=s^5+s^4+9\},$$

$$Q'=\{(s,s')|-5 \leq s \leq 10 \wedge s'=2s^5+s^4-5s^3+4s+9\}.$$

The domain of the intersection of these two functions is the same as that of P and Q, namely:

$$dom(P' \cap Q')=\{-2,-1,0,1,2\}.$$

On the other hand,

$$dom(P') \cap dom(Q')=\{s|-5 \leq s \leq 5\},$$

$$dom(P') \cup dom(Q')=\{s|-10 \leq s \leq 10\},$$

From this, we can derive easily:

$$\delta_1(P',Q')=\{-5,-4,-3,3,4,5\},$$

$$\delta_2(P',Q')=\{-10,-9,-8,-7,-6,-5,-4,-3,3,4,5,6,7,8,9,10\}.$$

We leave it to the reader to check that these are indeed the sets of initial states for which programs P' and Q' have distinct outcomes for the selected definitions of outcomes and outcome comparisons. Figure 10 helps in this analysis.

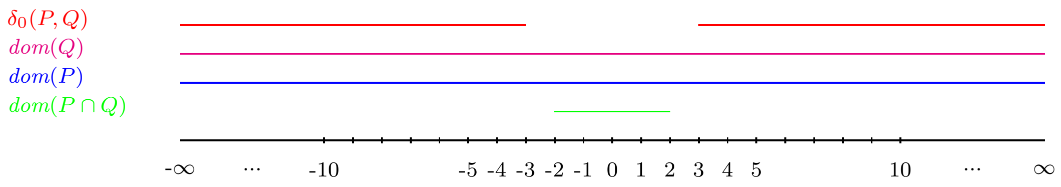


Figure 9: Basic differentiator Set for P and Q

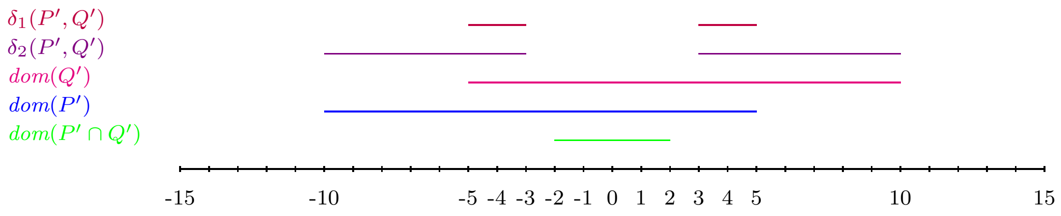


Figure 10: Strict and Broad differentiator Sets for P' and Q'

4.3. Differentiator Sets and Mutant Subsumption

In [34, 35] Kurtz et al. define mutant subsumption as follows: *Given a program P and two mutants thereof M and M', we say that M' subsumes M (in the sense of true subsumption) with respect to P if and only if:*

P1 There exists a test s such that P and M' compute different outcomes on s.

P2 For every possible test on P, if M' computes a different outcome from P, then so does M.

The following Proposition formulates the condition of true subsumption in terms of basic differentiator sets.

Proposition 6. *Given a program P on space S and two mutants M and M' on S, such that P, M and M' converge for all s in S, M' subsumes M with respect to P in the sense of true subsumption if and only if:*

$$\emptyset \subset \delta_0(P, M') \subseteq \delta_0(P, M).$$

Proof. We prove in turn that P1 is equivalent to $\emptyset \subset \delta_0(P, M')$ and that P2 is equivalent to $\delta_0(P, M') \subseteq \delta_0(P, M)$. We use the simple lemma that the set of states for which two functions F and G produce the same outcome ($F(s) = G(s)$) is $dom(F \cap G)$.
P1 is interpreted as:
$\exists s : P(s) \neq M'(s)$
$\Leftrightarrow$ {Interpretation}
$\emptyset \subset \{s | P(s) \neq M'(s)\}$
$\Leftrightarrow$ {Set theory}

$\emptyset \subset \overline{\{s|P(s) = M'(s)\}}$
$\Leftrightarrow$ {Lemma above}
$\emptyset \subset \overline{dom(P \cap M')}$
$\Leftrightarrow$ {Definition of δ_0}
$\emptyset \subset \delta_0(P, M')$.
P2 is interpreted as:
$\forall s : P(s) \neq M'(s) \Rightarrow P(s) \neq M(s)$
$\Leftrightarrow$ {Set theory}
$\{s|P(s) \neq M'(s)\} \subseteq \{s|P(s) \neq M(s)\}$
$\Leftrightarrow$ {Inverting the inclusion}
$\{s|P(s) = M(s)\} \subseteq \{s|P(s) = M'(s)\}$
$\Leftrightarrow$ {Lemma above}
$dom(P \cap M) \subseteq dom(P \cap M')$
$\Leftrightarrow$ {Definition of δ_0}
$\delta_0(P, M') \subseteq \delta_0(P, M)$. **qed**

This Proposition provides that the original ([34, 35]) definition of *true subsumption* can be defined by means of *basic differentiator sets*; but this definition applies only if the base program and its mutants converge for all initial states. If we want to make provisions for the possibility that the base program (P) and its mutants (M, M') may diverge, we can use *strict differentiator sets* and *broad differentiator sets*. Whence the following definitions.

Definition 8. *Given a program P on space S and mutants M and M' of P, we say that mutant M'* subsumes mutant M *in the sense of* basic subsumption *if and only if:*
$$\emptyset \subset \delta_0(P, M') \subseteq \delta_0(P, M).$$

Definition 9. *Given a program P on space S and mutants M and M' of P, we say that mutant M'* subsumes mutant M *in the sense of* strict subsumption *if and only if:*
$$\emptyset \subset \delta_1(P, M') \subseteq \delta_1(P, M).$$

Definition 10. *Given a program P on space S and mutants M and M' of P, we say that mutant M'* subsumes mutant M *in the sense of* broad subsumption *if and only if:*
$$\emptyset \subset \delta_2(P, M') \subseteq \delta_2(P, M).$$

Of course, having three different definitions of mutant subsumption may lead to three different subsumption graphs, whence potentially three different minimal mutant sets.

In [34, 35], Kurtz et al. introduce *dynamic subsumption* as follows: Given a program P, a test suite T and two mutants of P, M and M', we say that M' *dynamically subsumes* M with respect to P for test suite T if and only if:

D1 There exists a test t in T such that P and M' compute different outcomes on t.

D2 For every possible test t in T, if M' computes a different outcome from P, then so does M.

We argue that once we admit the possibility that programs and mutants may diverge for some inputs, then there is really no difference between true subsumption and dynamic subsumption: dynamic subsumption with respect to program P and test data T is the same as subsumption with respect to the program whose function is ${}_{T\backslash}P$, the pre-restriction of P to T:

```
if (s in T) {P;} else {abort();}.
```

Given the possibility for programs to define partial (rather than total) functions, dynamic (i.e. test suite dependent) mutant subsumption is reducile to strict subsumption of the program pre-restricted to the test suite.

4.4. Subsumption Graph Topology

Modeling mutant subsumption by an inclusion relationship between differentiator sets opens an opportunity that was hitherto unavailable: Assuming that the differentiator sets of mutants are statistically independent, we can estimate the probability that two mutants are in a subsumption relationship by computing the probability that two random non-empty subsets of a set are in an inclusion relationship. This probability can, in turn be used to estimate the number of arcs in a subsumption graph, and can be used to estimate the number of maximal nodes in the subsumption graph, all without having to draw the graph. estimating the number of maximal nodes in a subsumption graph is important in practice, because it enables us to determine whether the effort of building the subsumption graph is even worthwhile: If we find that the number of maximal nodes is not much smaller than the total number of mutants, then the whole subsumption exercise may be futile.

If subsumption is judged by observing the behavior of mutants on a test suite T, then all the differentiator sets are by construction subsets of T. Given a set T and K non-empty subsets thereof (which represent, for our puposes, the differentiator sets of K mutants), we ponder the question: what is the probability that any two subsets among K are in an inclusion relationship? This represents, for our purposes, the probability that two mutants are in a subsumption relation. Let D be the random variable that takes its values in subsets of T. The probability that D takes any particular value E is given by the inverse of the number of non-empty subsets of T:

$$prob(D = E) = \frac{1}{2^T - 1}.$$

The probability that D takes the value of a subset of size n is:

$$prob(|D| = n) = \frac{\binom{T}{n}}{2^T - 1}.$$

Given a subset E of size n, the probability that another subset E' is a subset of E is:

$$prob(E' \subseteq E) = \frac{2^n - 1}{2^T - 1},$$

where the numerator is the number of subsets of E and the denominator is the total number of subsets of T. The probability that two subsets E and E' of T are in a subset relation is:

$prob(E' \subseteq E)$

$=$ {conditional probability}

$\sum_{n=1}^{T} prob(E' \subseteq E|\ |E|=n) \times prob(|E|=n)$

$=$ {substitutions}

$\sum_{n=1}^{T} \frac{2^n-1}{2^T-1} \times \frac{\binom{T}{n}}{2^T-1}.$

$=$ {simplification}

$\sum_{n=1}^{T} \frac{2^n-1}{(2^T-1)^2} \times \binom{T}{n}.$

$=$ {factorization}

$\frac{1}{(2^T-1)^2} \times \sum_{n=1}^{T} 2^n \binom{T}{n}$

$-\frac{1}{(2^T-1)^2} \times \sum_{n=1}^{T} \binom{T}{n}.$

$=$ {highlighting the binomial formula}

$\frac{1}{(2^T-1)^2} \times \sum_{n=1}^{T} 2^n \times 1^{T-n} \binom{T}{n}$

$-\frac{1}{(2^T-1)^2} \times \sum_{n=1}^{T} \binom{T}{n}.$

$=$ {simplifying}

$\frac{3^T}{(2^T-1)^2} - \frac{2^T}{(2^T-1)^2}.$

For large (or even moderate) values of T, this can be approximated by $(\frac{3}{4})^T$. Under the assumption of statistical independence cited above, the expected number of arcs in a subsumption graph of K nodes can be approximated by:

$$(\frac{3}{4})^T \times K(K-1).$$

In practice, even with moderate values of T, this expected number is very small.

Using the probability estimate $p = \frac{3}{4}^T$, we can estimate the probability that any subset of T is maximal: A given subset is maximal if and only if all $(K-1)$ other subsets are not supersets there of; hence,

$$prob(maximality) = (1 - (\frac{3}{4})^T)^{K-1}.$$

Whence we derive the expected number of maximal mutants in a subsumption (/ relative correctness) graph that stems from K mutants and a test suite of size T:

$$K \times (1 - (\frac{3}{4})^T)^{K-1}.$$

When we consider published subsumption graphs [34, 35, 22, 37, 50, 53, 55], we find that they have far more arcs than our estimates, and far fewer maximal mutants than our estimates; this may be due to our assumption of statistical independence of differentiator sets. Hence this matter is left for future investigation; if the differentiator sets of the mutants of a base program are not statistically independent, then it is very intriguing to elucidate what statistical relationship represents their dependence.

5. Relative Correctness and Subsumption

So far we have used detector sets to define (partial and total) correctness, and we have used differentiator sets to define (basic, strict, and broad) subsumption. In this section we present two simple Propositions that relate correctness and subsumption.

Proposition 7. *We consider a program P on space S and two mutants M and M' such that P, M and M' converge for all states in S. Then M' subsumes M in the sense of basic subsumption if and only if M' is more-totally-correct than M with respect to (the function of) P, and M' is not (absolutely) totally correct with respect to (the function of) P.*

Proof. Since P converges for all s in S, $dom(P) = S$. Hence the detector set of M and M' for total correctness with respect to the function of P (viewed as a specification) is:
$dom(P) \cap \overline{dom(P \cap M)} = S \cap \overline{dom(P \cap M)} = \overline{dom(P \cap M)}$.
This is the same as the basic differentiator set of P and M. The two clauses of the basic subsumption relation of M' over M with respect to P can be written as (once we replace differentiator sets by the corresponding detector sets):

- P1 $\emptyset \subset \Delta^{TOT}(P, M')$, which according to Proposition 2 is equivalent to: M' is not (absolutely) totally correct with respect to (the function of) P.
- P2 $\Delta^{TOT}(P, M') \subseteq \Delta^{TOT}(P, M)$, which according to Definition 4 is equivalent to: M' is more-totally-correct than M with respect to (the function of) P.

qed

This Proposition means that the subsumption graphs seen in [34, 35, 22, 50, 53, 37, 28, 34, 55], represent essentially the same relation as the graphs of relative correctness seen in [12, 13, 32, 31]; the only meaningful/interesting difference is that, while in relative correctness we are interested in the top of the graph, which represents the absolutely correct programs, in subsumption we are interested in the layer immediately below the absolutely correct programs, which represents the *maximally stubborn* mutants [57].

Proposition 8. *We consider a program P on space S and two mutants M and M' of P. Then M' subsumes M in the sense of strict subsumption if and only if M' is more-partially-correct than M with respect to (the function of) P, and M' is not (absolutely) partially correct with respect to (the function of) P.*

m0: {s=pow(s,3)+4;}	m4: {s=pow(s,3)+s+1;}	m8: {s=pow(s,3)+s*s-4*s+8;}
m1: {s=pow(s,3)+5;}	m5: {s=pow(s,3)+s;}	m9: {s=2*pow(s,3)-6*s*s+11*s-3;}
m2: {s=pow(s,3)+6;}	m6: {s=pow(s,3)+s*s-5*s+9;}	m10:{s=3*pow(s,3)-12*s*s+22*s-9;}
m3: {s=pow(s,3)+s+2;}	m7: {s=pow(s,3)+s*s-3*s+5;}	m11:{s=4*pow(s,3)-18*s*s+33*s-15;}

Table 6: Mutants of Program P

Proof. If and only if M' subsumes M in the strict sense with respect to P, we can write:
$\emptyset \subset \delta_1(P, M') \subseteq \delta_1(P, M)$.
Interestingly, the strict differentiator set of P and M is the same as the detector set of M with respect to (the function) of P for partial correctness. We rewrite this condition as:
$\emptyset \subset \Delta^{PAR}(P, M') \subseteq \Delta^{PAR}(P, M)$.
According to Proposition 3, the first inequation is equivalent to: M' is not partially correct with respect to (the function of) P. According to Definition 5, the second inequation is equivalent to: M' is more-partially-correct than M with respect to (the function of) P. **qed**

For illustration, we consider the following program P on space S defined as the set of integers:

```
p:  {if ((s<1) || (s>3)) {abort();}
    else {s=3+s*s*s;}}
```

The function of this program is:

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

This is the same relation as specification R presented in section 3.4, though we change its name from R (a specification used as a reference for relative correctness) to P (a program used as a reference for mutant subsumption). Also, for mutants of P, we take the programs that we used in section 3.4, though we rename them as (`m0`, `m1`, `m2`, ..., `m11`), listed in Table 6, and we rank them by subsumption with respect to P. These programs are not derived from P by any known mutation operator we know of, but we use them for the purpose of illustration.

Since program P does not converge for all s, we cannot use basic subsumption; we will use strict subsumption instead. Table 7 shows the strict differentiator set of each mutant with respect to P, and Figure 11 shows the strict subsumption graph of the mutants `m0`, `m1`, `m2`, ... `m11` with respect to program P. Not surprinsingly, this is the exact same graph as that of Figure 7, except for the different names (`mi` vs `pi`). In Figure 11 we highlight (in blue) the maximally subsuming mutants; they are the most (relatively) correct among mutants that are not absolutely correct. Their strict differentiator sets are singletons. Whereas relative correctness is based on comparing detector sets, subsumption is based on comparing differentiator sets.

m0	$\{1,2,3\}$	m1	$\{1,2,3\}$	m2	$\{1,2,3\}$
m3	$\{2,3\}$	m4	$\{1,3\}$	m5	$\{1,2\}$
m6	$\{1\}$	m7	$\{3\}$	m8	$\{2\}$
m9	$\{\}$	m10	$\{\}$	m11	$\{\}$

Table 7: Strict Differentiator Sets of Mutants

$$\begin{tikzcd}
m_{10} \arrow[r, leftrightarrow] & m_9 \arrow[r, leftrightarrow] & m_{11} \\
m_7 \arrow[u] \arrow[ur] \arrow[urr] & m_6 \arrow[ul] \arrow[u] \arrow[ur] & m_8 \arrow[ull] \arrow[ul] \arrow[u] \\
m_4 \arrow[u] \arrow[ur] & m_3 \arrow[ul] \arrow[ur] & m_5 \arrow[ul] \arrow[u] \\
m_1 \arrow[u] \arrow[ur] \arrow[urr] \arrow[r, leftrightarrow] & m_0 \arrow[ul] \arrow[u] \arrow[ur] \arrow[r, leftrightarrow] & m_2 \arrow[ull] \arrow[ul] \arrow[u]
\end{tikzcd}$$

Figure 11: Ordering Mutants by Subsumption with respect to P

6. Test Suite Effectiveness

6.1. Defining Effectiveness

The effectiveness of an artifact can only be defined in reference to a *purpose* of the artifact, and must reflect the artifact's fitness to the declared purpose. Hence before we discuss how to quantify the effectiveness of a test suite, we must agree on what is the purpose of a test suite. We propose the following axiom:

Axiom 1. *Given a program P on space S and a specification R on S, the purpose of a test suite T (a subset of S) is:*

- *To expose the failures of P with respect to R if P is incorrect with respect to R, or*
- *To gives us confidence in the correctness of P with respect to R, if P is correct.*

These two clauses are logically equivalent, since what gives us confidence in the correctness of P if it passes the test T successfully is the assurance that if P were incorrect, test suite T would have revealed its incorrectness by exposing its failures.

Whereas, according to this axiom, the purpose of a test suite is to expose program failures, most metrics in use nowadays to assess the effectiveness of a test suite are focused on another attribute: the ability to detect faults [39, 38, 24, 19, 6, 52, 9, 41]. Indeed, metrics used nowadays to quantify the effectiveness of test suites can be divided into two broad categories:

- *Syntactic Coverage.* This family of metrics reflects the extent to which a test suite exercises syntactic features of the program's source code (statements, branches, conditions, lines, paths, etc). The rationale for syntactic coverage is that in order to detect a fault, we need to exercise the code that contains it.
- *Mutation Coverage.* Mutation coverage reflects the ability of a test suite to distinguish the base program from mutants thereof obtained by applying small mutations to its source code. The rationale for mutation coverage is that mutants are faithful proxies of faults [5, 4, 48, 29], hence the ability to kill mutants can be used as an indicator of a test suite's ability to detect faults.

Exercising all the syntactic features of a program is neither necessary nor sufficient to detect all the faults in a program. It is not sufficient since the same faulty feature may be sensitized for some inputs, and not sentizied for other inputs; hence the same fault can go undetected at testing time but cause failures in field usage. Also, exercising all the syntactic features of a program is not necessary either, since strictly speaking only faulty features need to be exercised, to expose all the faults of a program.

While mutation coverage is usually a better measure of test suite effectiveness and is sometimes used as a reference in analyzing other measures [6, 26], it has issues of its own:

- The same mutation score means vastly different things depending on whether the surviving mutants are equivalent to the base program (in which case the effectiveness of the test suite is not in question) or not.
- The same mutation score means vastly different things depending on whether the killed mutants are semantically equivalent to each other (in which case the test suite has just killed several times the same mutants) or are semantically distinct (in which case the test suite has killed that many distinct mutants).
- Whereas the mutation score of a test suite is usually considered as an attribute of the test suite and the program under test, we find in empirical experiments that the mutation score of the same test suite varies widely depending on the mutation operators that are used to generate mutants [2].

Hence before we can interpret or assign a meaning to the mutation score of a test suite, we must consider the following questions: Are the surviving mutants semantically equivalent to the base program or distinct? Are the killed mutants semantically equivalent to each other or distinct? To what extent is the mutation score dependent on the mutation operators? Such questions challenge the validity of the mutation score as an accurate measure of test suite effectiveness.

All the quantitative metrics of test suite effectiveness, whether they are based on syntactic or semantic criteria, have another flaw in common: they define a total ordering on what is essentially a partially ordered set. Indeed, if we measure test suite effectiveness by a number, then any two test suites can be compared, even when they are not actually ordered (none can be considered to be better than the other); two test suites that detect distinct sets of faults or disjoint sets of faults are not comparable (we have no basis for considering that one is better than the other), yet if their effectiveness is measured by numbers, we will always find one to have a greater (or equal) effectiveness than the other. Modeling a partial ordering relation by means of a total ordering relation will necessarily yield a loss of precision.

6.2. Detecting Faults vs. Exposing Failures

There are two ways to quantify the effectiveness of a test suite:

- By equating effectiveness with the ability to detect faults, or
- By equating effectiveness with the ability to expose failures.

There is no one-to-one correspondence between failures and faults: the same failure can be attributed to more than one fault or set of faults. Hence while failures are observable, certifiable effects (whether program P runs correctly on input s with respect to specification R can be checked by oracle $\Omega^{TOT}_{[R,P]}()$), faults are hypothetical, speculative causes to the observed effects; Avizienis et al. [7] define a *fault* as *the adjudged or hypothesized cause of an error*. Hence it is safe to argue that detecting faults and exposing failures are distinct capabilities: a test suite may be good at one and not the other. An empirical study by Ayad et al. [8] finds that these two capabilities have low statistical correlation. This bears out a long standing recognition that faults are not created equal: some cause failures at a much higher rate than others [45].

While we argue in the previous section that most existing measures of test suite effectiveness appear to equate effectiveness with the ability to detect faults, we propose in section 6.5 a measure (under the name *semantic coverage*) that equates effetiveness with the ability to expose failures. Prior to introducing this metric, we discuss in sections 6.3 and 6.4 the requirements that a measure of test suite effectiveness ought to satisfy, then the design principles that we adopt in the design of this metric. In section 6.6 we prove that semantic coverage satisfies all the criteria listed in section 6.3.

6.3. Requirements on Test Suite Effectiveness

We consider a program P on space S and a relation R on S; we let T be a subset of S and we want to define the *semantic coverage* of T to reflect the effectiveness of T to expose possible failures of P with respect to R. As such, semantic coverage depends on four parameters: T, P, R and the standard of correctness with respect to which correctness is tested. The four requirements below mandate how we want semantic coverage to vary according to each of the following parameters.

- *Monotonicity with Respect to T.* Of course, we want the effectiveness of a test suite to be monotonic with respect to set inclusion: If T is a subset of T', then the effectiveness of T ought to be less than or equal that of T'.
- *Monotonicity with respect to the specification.* If specification R is refined by specification R', this means that R' imposes stiffer requirements on candidate programs than R; hence it is more difficult to test a program for correctness with respect to R' than with respect to R. Consequently, the same test suite T ought to have a smaller measure of effectiveness against R' than against R.
- *Monotonicity with respect to the program.* If program P' is more-correct than program P with respect to specification R, then P' has fewer failures for a test suite T to expose than does P. Hence a test suite T ought to have a higher measure of effectiveness for P' than for P.
- *Monotonicity with respect to the standard of correctness.* Total correctness of a program P with respect to a specification R is a stronger property than partial correctness; hence it is a more difficult property to test a program against. Therefore, the same test suite T ought to have lower semantic coverage for total correctness than for partial correctness, given the same program and specification.

6.4. Design Principles

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

- *Focus on Failure.* Given that a failure is an observable effect and a fault is the hypothesized cause of the observed effect, it is clearly better to anchor our definition in failures.
- *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. Consequently, 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 the semantic coverage of some test suite T is superior to that of T', it is because T is better (in some sense) than T'.
- *Analytical Validation.* There are ample reasons why we could not envisage an empirical validation of semantic coverage, some of which are: we know of no measure of test suite effectiveness that can be considered as *ground truth*; whereas all coverage metrics we know of take numeric values, semantic coverage is an element of a partially ordered set, hence precluding any statistical analysis; whereas existing coverage metrics focus on faults and fault detection, semantic coverage is based on failures; whereas existing coverage metrics depend exclusively on the program and test suite, semantic coverage depends also on the specification and standard of correctness, hence precluding meaningful comparison.

  Consequently, we validate semantic coverage analytically, by showing that it meets the monotonicity conditions discussed in section 6.3.

6.5. Semantic Coverage

Given that the declared purpose of a test suite if to expose the failures of an incorrect program (Axiom 1) and given that the detector set of program P with respect to a specification R includes all the inputs that expose the failures of P with respect to R, an ideal test suite is any superset of the detector set. This condition can be formulated as:

$$\Delta(R, P) \subseteq T,$$

where $\Delta(R, P)$ is a stand-in for $\Delta^{PAR}(R, P)$ or $\Delta^{TOT}(R, P)$, depending on whether we are testing P for partial or total correctness. Now that we know what characterizes an ideal test suite, we introduce a measure that reflects to what extent a random (not necessarily ideal) test suite differs from an ideal test suite. The elements that preclude a test suite T from being a superset of $\Delta(R, P)$ are the elements of $\Delta(R, P)$ that are outside T; the fewer such elements, the better the test suite. The set of these elements is $\overline{T} \cap \Delta(R, P)$; since we want a quantity that increases (rather than decreases) with the quality of a test suite, we take the complement of this expression. Whence the following definition.

Definition 11. *Given a program P and a specification R on space S, the* semantic coverage *of test suite T for program P with respect to specification R is denoted by $\Gamma_{R,P}(T)$ and defined by:*

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

Definition 11 represents, in effect, two distinct definitions, depending on whether we are interested to test P for partial correctnes or total correctness:

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

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

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

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

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

- $\Gamma_{R,P}(T) = \emptyset$. The semantic coverage of a test T for program P with respect to specification R is empty if and only if T is empty and the complement of the detector set of P with respect to R is empty; in such a case the detector set of P with respect to R is all of S. In other words, even though any element of S exposes the incorrectness of P with respect to R, T does not reveal that P is incorrect since it is empty. This is clearly characteristic of a useless test suite.

- $\Gamma_{R,P}(T) = S$. If the union of two sets equals S, the complement of each set is a subset of the other set. Whence: $\Delta(R,P) \subseteq T$, which is precisely how we characterize ideal test suites. As a special case, if P is correct with respect to R, then, according to Propositions 2 and 3, the semantic coverage of any test suite T with respect to P and R is S. If there are no failures to expose, then any test suite will have maximal semantic coverage.

See Figure 12; the semantic coverage of test suite T for program P with respect to specification R is the area colored (both shades of) green. The (partially hidden) red rectangle represents the detector set of P with respect to R; the dark green area represents the complement of this set, i.e. in fact all the test data that need not be exercised; the light green area represents test suite T. The semantic coverage of T is the union of the area that need not be tested (dark green) with the area that is actually tested (light green). Test suite T is as good as the red area is small.

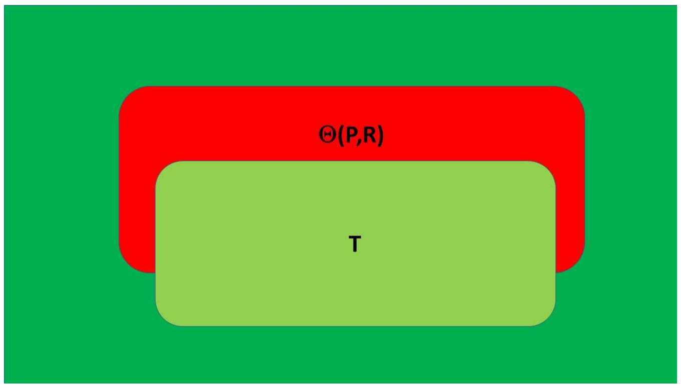


Figure 12: Semantic Coverage of Test T for Program P with respect to R (shades of green)

6.6. Analytical Validation

In this section we revisit the requirements put forth in section 6.3 and prove that the formula of semantic coverage proposed above does satisfy all these requirements.

6.6.1. Monotonicity with Respect to the Test Suite

Definition 11 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.

6.6.2. Monotonicity with Respect to Relative Correctness

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

Proposition 9. *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:
$\Delta^{TOT}(R,P') \subseteq \Delta^{TOT}(R,P)$.
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 10. *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 5, we have:
$\Delta^{PAR}(R,P') \subseteq \Delta^{PAR}(R,P)$.
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**

6.6.3. 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 11. *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. It suffices to prove $\Delta^{TOT}(R',P) \supseteq \Delta^{TOT}(R,P)$, 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 sides 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,
$\Delta^{TOT}(R,P) \subseteq \Delta^{TOT}(R',P)$.
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 12. *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
$\Delta^{TOT}(R,P) \subseteq \Delta^{TOT}(R',P)$.
By taking the intersection with $dom(P)$ on both sires, we find
$\Delta^{PAR}(R,P) \subseteq \Delta^{PAR}(R',P)$.
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**

6.6.4. 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 13. *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:
$\Delta^{PAR}(R,P) \subseteq \Delta^{TOT}(R,P)$.
By complementing both sides and inverting the inequality, we find:
$\overline{\Delta^{PAR}(R,P)} \supseteq \overline{\Delta^{TOT}(R,P)}$.
By taking the union with T on both sides, we find the result sought. **qed**

7. Mutant Set Minimization

Mutant subsumption has been introduced as a way to minimize the cardinality of a mutant set, on the grounds that a subsumed mutant can be removed from a mutant set without affecting its effectiveness. We argue that at its core, mutant set minimization is an optimization problem; yet to the best of our understanding it has not been cast as such in the published literature. Indeed, an optimization problem ought to be defined by two parameters, namely: The objective function to minimize, and the constraint under which the objective function is minimized. Whereas the objective function of mutant set minimization is clear (the cardinality of the mutant set), the constraint under which this function is minimized had not been defined explicitly in the literature. Of course, the implicit assumption is that we are minimizing the cardinality of the mutant set while preserving its effectiveness, but this raises the question: How do we define or quantify the effectiveness of a mutant set?

The effectiveness of an artifact must be defined with respect to the purpose of the artifact, and must reflect the artifact's fitness for that purpose; hence the first question we must consider is: what is the purpose of a mutant set? If we consider that the purpose of a set of mutants is to vet test suites, then the quality of a mutant set can be assessed as a function of the quality of the test suites that it vets. This, in turn, raises the question: What does it mean for a mutant set to vet a test suite? We consider that a mutant set μ vets a test suite T if and only if T kills every mutant in μ, and no subset of T does (i.e. whenever we remove an element of T, at least one mutant in μ survives); see Figure 13. Given that the same mutant set may vet several test suites, we must ponder the question: How do we aggregate the quality of all the test suites

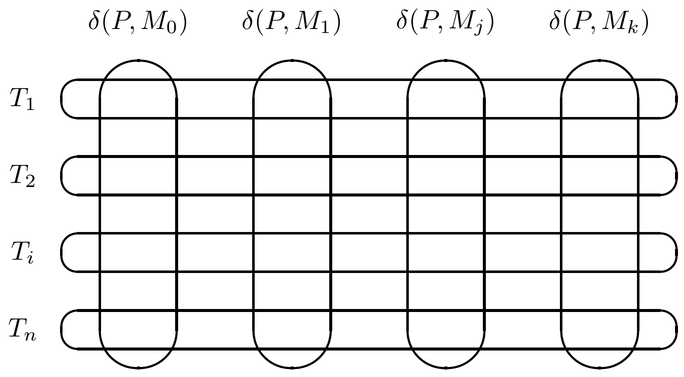


Figure 13: Test Suites $\{T_i\}$ Vetted by a Mutant Set$\mu = \{M_j\}$

vetted by a mutant set μ into a synthetic quality metric of the whole set? This is the subject of the next section.

7.1. Mutant Set Effectiveness

We resolve to quantify the effectiveness of a mutant set through the semantic coverage of the test suites that it vets. We denote by $\theta(\mu)$ the set of minimal test suites vetted by mutant set μ:

$$\theta(\mu) = \{T | \forall M \in \mu : T \cap \delta(P, M) \neq \emptyset\} \\ \cap \\ \{T | \forall T' \subset T, \exists M \in \mu : T' \cap \delta(P, M) = \emptyset\}.$$

Figure 13 illustrates the relation of orthogonality between the differentiator sets and test suites, and helps follow the subsequent discussions. We want to characterize the effectiveness of mutant set μ as an aggregate of the semantic coverage of the elements of $\theta(\mu)$. Since the test suites vetted by a mutant set may have different levels of semantic coverage, we resolve to capture the effectiveness of a mutant set by a lower bound and an upper bound, as follows:

Definition 12. Mutant Set Effectiveness.

- Assured Effectiveness, $\eta_A(\mu)$. *This is a lower bound of the semantic coverage of all the test suites vetted by* μ.

$$\eta_A(\mu) = \bigcap_{T \in \theta(\mu)} \Gamma_{R,P}(T).$$

- Potential Effectiveness, $\eta_P(\mu)$. *This is an upper bound of the semantic coverage of all the test suites vetted by* μ. *It represents the potential semantic coverage of a test suite that is vetted by* μ.

$$\eta_P(\mu) = \bigcup_{T \in \theta(\mu)} \Gamma_{R,P}(T).$$

It stems from this definition that the semantic coverage of any test suite that is vetted by μ is bounded by the assured effectiveness and the potential effectiveness of μ.

$$\forall T \in \theta(\mu) : \eta_A(\mu) \subseteq \Gamma(T) \subseteq \eta_P(\mu).$$

The following Proposition stems readily from the definition of semantic coverage.

Proposition 14. *Given a program P on space S and a specification R on S, we let μ be a set of mutants of P, then the assured effectiveness of μ can be written as:*

$$\eta_A(\mu) = \overline{\Delta(R,P)} \cup \bigcap_{T \in \theta(\mu)} T.$$

Also, the potential effectiveness of μ can be written as:

$$\eta_P(\mu) = \overline{\Delta(R,P)} \cup \bigcup_{T \in \theta(\mu)} T.$$

Proof. This proposition stems readily from the fact that $\Delta(R,P)$ is independent of T, hence can be factored out of the effectiveness formulas. **qed**

7.2. Mutant Set Minimization as An Optimization Problem

The following Proposition provides that removing a subsumed mutant in a mutant set preserves its potential effectiveness.

Proposition 15. *Let μ be a set of mutants of program P and let μ' be $\mu' = \mu \cup \{M'\}$ for some mutant M' that is subsumed by some mutant M of μ. Then μ and μ' have the same potential effectiveness.*

Proof. According to Proposition 14, it suffices to prove that

$$\bigcup_{T \in \theta(\mu)} T = \bigcup_{T \in \theta(\mu')} T.$$

Since μ' is a superset of μ, $\theta(\mu')$ is a subset of $\theta(\mu)$ (since the test suites of $\theta(\mu')$ have more test to kill). Therefore

$$\bigcup_{T \in \theta(\mu')} T \subseteq \bigcup_{T \in \theta(\mu)} T.$$

To prove the reverse inclusion, we consider an element t of $\bigcup_{T \in \theta(\mu)} T$; there exists a test suite T in $\theta(\mu)$ that contains t. This test suite has a non-empty intersection with the detector sets of all elements of μ, including with $\delta(P,M)$; since $\delta(P,M) \subseteq \delta(P,M')$, T has a non-empty intersection with $\delta(P,M')$, hence with the detector sets of all the elements of μ'. We infer that t is an element of $\bigcup_{T \in \theta(\mu')} T$. **qed**

We have not proven that removing a subsumed mutant preserves the assured effectiveness of a mutant set, nor have we found a counter-example; this matter is under investigation.

Given that subsumption favors mutants whose detector sets are smallest without being empty, we would expect that it lends a special importance to mutants whose detector sets are singletons; this is confirmed by the following Proposition.

Proposition 16. *Let μ be a set of mutants of a program P on space S and let R be a specification on S. Let μ_0 be the set of mutants in μ whose detector set is a singleton and let T_0 be the union of all the detector sets in μ_0. Then the semantic coverage of T_0 with respect to P and R is a subset of the assured effectiveness of μ with respect to R.*

Proof. The semantic coverage of T_0 is, by definition, $\Gamma_{R,P}(T_0) = T_0 \cup \overline{\Delta(R,P)}$. Any test suite that kills all the mutants in μ kills all the mutants in μ_0, hence is a superset of T_0, since for all M in μ_0, $T_0 \cap \delta(P,M) \neq \emptyset$ is equivalent to $T_0 \supseteq \delta(P,M)$. Since T_0 is a subset of any test suite that kills all the mutants in μ, it is necessarily a subset of their intersection. Whence we infer:

$$\Gamma_{R,P}(T_0) \subseteq \bigcap_{T \in \theta(\mu)} T \cup \overline{\Delta(R,P)} = \eta_A(\mu).$$

qed

We can see a confirmation of this Proposition in the example below, where T_0 is $\delta(P,M0) \cup \delta(P,M1) = \{0,1\}$.

7.3. Summary Illustration

We let space S be defined by $S = \{0,1,2,3,4,5\}$ and we consider a (fictitious) program P and specification R such that the detector set of P with respect to R is:

$\Delta(R,P) = \{0,1,2,3\}$.

Since the detector set of this program is not empty, this program is incorrect; testing it on any element of this set exposes its incorrectness. Also, we consider the set of mutants $\mu = \{M0, M1, M2, M3\}$ whose differentiator sets with respect to P are, respectively:

$\delta(P,M0) = \{0\}$.
$\delta(P,M1) = \{1\}$.
$\delta(P,M2) = \{2,4\}$.
$\delta(P,M3) = \{3,5\}$.

The following test suites kill all the mutants in μ, hence are elements of $\theta(\mu)$:

$T1 = \{0,1,2,3\}$.
$T2 = \{0,1,2,5\}$.
$T3 = \{0,1,4,3\}$.
$T4 = \{0,1,4,5\}$.

The assured effectiveness of mutant set μ is:

$$\begin{aligned}\eta_A(\mu) &= T1 \cap T2 \cap T3 \cap T4 \cup \overline{\{0,1,2,3\}}\\ &= \{0,1\} \cup \overline{\{0,1,2,3\}}\\ &= \{0,1,4,5\}.\end{aligned}$$

The potential effectiveness of mutant set μ is:

$$\begin{aligned}\eta_P(\mu) &= T1 \cup T2 \cup T3 \cup T4 \cup \overline{\{0,1,2,3\}}\\ &= \{0,1,2,3,4,5\} \cup \overline{\{0,1,2,3\}}\\ &= S.\end{aligned}$$

Interpretation: Though program P fails for four tests ($\{0,1,2,3\}$) a test suite that is vetted by mutant set μ is assured to reveal only two of these failures: $\{0,1\}$. But mutant set μ has the *potential* to reveal all the failures of program P, if only we are lucky to pick the right test suite among those that are vetted by μ. If we select test suite $T1$, then we reveal all the failures of program P; in fact the semantic coverage of $T1$ is all of S. But $T4$ is also vetted by μ, yet it reveals only two failures of P with respect to R: $\{0,1\}$.

7.4. Mutant Set Minimization as an Optimization Problem

We conclude this section by formulating the minimization of mutant sets as an optimization problem, including an objective function and a constraint under which the objective function is minimized.

Mutant Set Minimization: Given a mutant set μ, find a mutant set $\mu*$ that

- minimizes $|\mu*|$,
- under the constraints:
  - $\mu* \subseteq \mu$.
  - $\eta_P(\mu*) = \eta_P(\mu)$.

8. Conclusion

8.1. Summary

The main contributions of this paper can be summarized as follows:

- *Detector Sets and Correctness.* The detector set of a program with respect to a specification is the set of inputs that disprove the correctness of the program with respect to the specification. Since there are two definitions of correctness (partial, total), we have two definitions of detector sets:

Detector Set for Partial Correctness	Detector Set for Total Correctness
$\Delta^{PAR}(R,P) = dom(R) \cap dom(P) \cap \overline{dom(R \cap P)}$	$\Delta^{TOT}(R,P) = dom(R) \cap \overline{dom(R \cap P)}$

  We use detector sets to characterize absolute correctness and relative correctness:

– *Absolute Correctness*: A program P is absolutely correct (in the sense of partial correctness or total correctness) with respect to R if and only if the correspnding detector set is empty:
  P correct with respect to R if and only if: $\Delta(R,P)=\emptyset$.

– *Relative Correctness.* A program P' is more-correct-than a program P (in the sense of partial or total correctness) with respect to R if and only if the detector set of P' is a subset of the detector set of P:
  P' more-correct than P if and only if: $\Delta(R,P')\subseteq\Delta(R,P')$.

Correctness Type	Partial Correctness	Total Correctness
Absolute Correctness : P is correct wrt R	$\Delta^{PAR}(R,P)$ $=\emptyset$	$\Delta^{TOT}(R,P)$ $=\emptyset$
Relative Correctness : P' is more-correct than P with respect to R	$\Delta^{PAR}(R,P')$ $\subseteq$ $\Delta^{PAR}(R,P)$	$\Delta^{TOT}(R,P')$ $\subseteq$ $\Delta^{TOT}(R,P)$

• *Differentiator Sets and Subsumption.* The differentiator set of two program P and Q is the set of inputs that expose their different behavior. Taking into account the possibility that one or both programs may diverge, we consider three possible definitions of differentiator sets:

Basic Differentiator Set	Strict Differentiator Set	Broad Differentiator set
$\delta_0(P,Q)=\overline{dom(P\cap Q)}$	$\delta_1(P,Q)=dom(P)\cap dom(Q)\cap\overline{dom(P\cap Q)}$	$\delta_2(P,Q)=(dom(P)\cup dom(Q))\cap\overline{dom(P\cap Q)}$

We use detector sets to generalize the concept of mutant subsumption by taking into account the possibility that the base program or its mutants diverge for some input: Mutant M' subsumes mutant M with respect to base program P if and only if:
  $\emptyset\subset\delta(P,M')\subseteq\delta(P,M)$,
where $\delta(P,M)$ stands in for the basic, strict or broad differentiator set of P and M, depending on the interpretation we choose.

Interpretation	M' subsumes M with respect to P iff:
Basic Subsumption	$\emptyset\subset\delta_0(P,M')\subseteq\delta_0(P,M)$
Strict Subsumption	$\emptyset\subset\delta_1(P,M')\subseteq\delta_1(P,M)$
Broad Subsumption	$\emptyset\subset\delta_2(P,M')\subseteq\delta_2(P,M)$

• *Subsumption as Relative Correctness.* Given that the detector set of partial correctness has the same definition as the strict differentiator set, we find that mutant M' subsumes mutant M with respect to program P in the sense of strict subsumption if and only if:

  - Mutant M' is not partially correct with respect to (the function of) P.
  - Mutant M' is more-partially-correct than mutant M with respect to (the function of) P,

- *Semantic Coverage.* Given a program P, a specification R, a test suite T, and a standard of correctness (partial or total correctness), the semantic coverage of T to test program P against specification R for the selected (partial or total) standard of correctness is denoted by $\Gamma_{R,P}(T)$ and defined as:
$$\Gamma_{R,P}(T) = T \cup \overline{\Delta(R, P)},$$
where $\Delta(R, P)$ is the detector set of (partial or total) correctness of program P with respect to specification R. Semantic coverage depends on four parameters, and is monotonic with respect to each one of them: We find that it increases when T increases (with respect to set inclusion), when R becomes less refined (easier to satisfy), when P grows more-correct with respect to R, and when we transition from total correctness to partial correctness.

- *Mutant Set Effectiveness.* We postulate that the purpose of a mutant set is to vet test suites, and we quantify the effectiveness of a mutant set by the semantic coverage of the test suites that the mutant set vets. Given that different vetted test suites may have different levels of semantic coverage, we aggregate these by means of two metrics: *assured effectiveness* and *potential effectiveness*, which represent, a lower bound and an upper bound, respectively, of the semantic coverage of vetted test suites.

- *Mutant Set Minimization as an Optimization Problem.* We model the problem of mutant set minimization as an optimization problem, where the objective function is the cardinality of the mutant set and the constraint under which the minimization is attempted is the presevation of the potential effectiveness. We prove that removal of a subsumed mutant from a mutant set does preserve the set's potential effectiveness.

8.2. Assessment and Critique: Practical Implications

In this section we briefly discuss the practical implications of the main results of this paper.

- *Detector Sets as a Basis for Modeling Correctness.* Detector sets can be used to model absolute total correctness, absolute partial correctness, relative total correctness and relative partial correctness; they do so in simple, uniform formulas. Relative correctness can, in turn, be used to define (and reason about) faults, elementary faults, fault density, fault depth, and fault removal, with applications in program repair [33]. In [33] Khaireddine et al. argue that to repair a program does not neccessarily mean to make it absolutely correct; it only means to make it more-correct than it is. They use the definition of relative correctness to derive an oracle for relative correctness, and show empirically that when they replace the test for absolute correctness by a test for relative correctness in existing program repair tools, they achieve better precision, better recall, and higher performance than the original version of the tools.

- *Detector Sets as a Basis for Defining Semantic Coverage.* One of the most important decisions we make in software testing is the choice of test suites, and one of the most important factors in choosing test suites is the criterion that we use to compare test suites. In this paper we argue that test suite effectiveness can be quantified by equating it with the ability to detect faults, or by equating it with the ability to expose failures, and we observe that most existing metrics for test suite effectiveness are based on the former criterion. Also, we propose a measure of test suite effectiveness that equates effetiveness with the ability to expose failures, and argue that exposing failure is the very purpose of a test suite (per Axiom 1).

- *Differentiator Sets as a Basis for Modeling Subsumption.* The original definition of mutant subsumption is based on comparing the outcomes of the execution of the base program and its mutants on selected inputs. This definition assumes implicitly that program outcomes are always defined, and can always be compared; but in practice neither of these assumptions is legitimate. When a program execution diverges, it is not clear whether we consider that the execution has no outcome or that divergence is itself an outcome; also, when we execute two programs and one of them or both of them diverge, it is unclear whether we consider the outcomes to be comparable, and under what condition we consider them to be identical or distinct. This is all the more important in mutation testing that several mutation operators are prone to cause mutants to diverge even when the original program converges.

  In this paper we discuss three definitions of differentiator sets, which differ by the way in which they define execution outcomes, the condition under which they consider outcomes to be comparables, and the condition under which they consider comparable outcomes to be identical or distinct. These three definitions yield, in turn, three definitions of subsumption: *basic sumsumption*, *strict subsumption*, and *broad subsumption* [3].

- *Relative Correctness and Subsumption.* Highlighting the analogy between (strict) subsumption and relative (partial) correctness is interesting, because it enables researchers to share results and insights across two areas of research that have proceeded concurrently but independently. Given that subsumption was introduced to analyze program mutations and relative correctness was introduced to analyze program faults, this analogy can be viewed as part of the ongoing discussion of the relationship between mutations and faults [5, 4, 48, 29].

- *Subsumption Graphs.* Modeling subsumption as an inclusion relationship between (differentiator) sets offers an opportunity to reason about the shape of a subsumption graph without actually drawing it: We can apply statistical methods to estimate the number of arcs in the graph, and the number of maximal nodes in the graph. Given K mutants M_1, M_2, ... M_K of some program P, we consider their differentiator sets $\delta(P, M_1)$, $\delta(P, M_2)$, ... $\delta(P, M_K)$, which are all subsets of test suite T, and we ponder the questions: what is the probability that any two non-empty differentiator sets are in an inclusion relation? what is the estimated (mean) number of such relationships? what is the estimated (mean) number of differentiator sets that are maximal by inclusion? The answers to these questions are useful in practice, as they enable us to predict the shape of the subsumption graph, and in fact whether it is worthwhile to build the graph at all (if the number of maximal mutants is not much smaller than the total number of mutants, it may not be worthwhile).

8.3. Threats to Validity

In theory, it is possible to estimate the probability that two random subsets of a set T are in a subset relation; applying this probability to differentiator sets of mutants, we can estimate the number of arcs in a subsumption graph and the number of maximal mutants in such a graph. When we apply this statistical analysis to published subsumption graphs [34, 35, 22, 50, 53, 37, 30, 28, 34, 55, 49], we find that our estimate is much lower than reality. This seems to shed doubt on our modeling assumption to the effect that differentiator sets of mutants are statistically independent. If the differentiator sets of the mutants of a given program are not statistically independent, then elucidating their statistical relationships is certainly a very worthwhile research goal, and an intriguing question.

Also, the formulas of mutant set effectiveness are based on a hierarchy of modeling decisions pertaining to detector sets, semantic coverage, assured effectiveness, and potential effectiveness, each of which adds a layer of threats to validity. These ought to be validated with analytical and empirical investigations.

Acknowledgements

The authors are very grateful to the anonymous reviewers for their careful, thoughtful evaluation and their insightful feedback. This work is partially support by the US National Science Foundation throught grant number DGE2043104.

References

[1] Aaltonen, K., Ihantola, P., Seppala, O., 10 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In: Companion to the 25th Annual ACM SIGPLAN Conference on OOPSLA. Reno, NV, pp. 153–160.

[2] AlBlwi, S., Ayad, A., Mili, A., April 2024. Mutation coverage is not strongly correlated with mutation coverage. In: Proceedings, IEEE Conference on Automated Software Testing. Lisbon, Portugal.

[3] AlBlwi, S., Marsit, I., Khaireddine, B., Ayad, A., Loh, J., Mili, A., July 2022. Generalized mutant subsumption. In: Proceedings, ICSOFT 2022. Lisbon, Portugal.

[4] Andrews, J., Briand, L., Labiche, Y., 01 2005. Is mutation an appropriate tool for testing experiments? pp. 402–411.

[5] Andrews, J., Briand, L., Labiche, Y., Siami Namin, A., 09 2006. Using mutation analysis for assessing and comparing testing coverage criteria. Software Engineering, IEEE Transactions on 32, 608–624.

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

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

[8] Ayad, A., AlBlwi, S., Mili, A., 2024. Detecting faults vs. revealing failures: Exploring the missing link. In: Proceedings, QRS 2024: 24th International Conference on Software Quality, Reliability, and Security. Cambridge, UK.

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

[10] Brinksma, E., Stoelinga, M., Briones, L. B., 2006. A semantic version for test coverage. Tech. rep., University of Twente.

[11] Chen, T. Y., Kuo, F. C., Lu, H., Poon, P. L., Towey, D., Tse, T., Zhou, Z. Q., 2018. Metamorphic testing: Challenges and opportunities. ACM Computing Surveys 51 (1), 1–27.

[12] Desharnais, J., Diallo, N., Ghardallou, W., Frias, M. F., Jaoua, A., Mili, A., September 2015. Relational mathematics for relative correctness. In: RAMICS, 2015. Vol. 9348 of LNCS. Springer Verlag, Braga, Portugal, pp. 191–208.

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

[14] Diallo, N., Ghardallou, W., Mili, A., June 2015. Program derivation by correctness enhancements. In: Refinement 2015. Oslo, Norway.

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

[16] Gazzola, L., Micucci, D., Mariani, L., January 2019. Automatic software repair: A survey. IEEE Trans. on Soft. Eng. 45 (1).

[17] Ghardallou, W., Diallo, N., Mili, A., Frias, M., April 2016. Debugging without testing. In: Proceedings, International Conference on Software Testing. Chicago, IL.

[18] Gladisch, C., Nov 2008. Verification-based test case generation for full feasible branch coverage. In: Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on. pp. 159–168.

[19] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., 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.

[20] Gopinath, R., Jensen, C., Groce, A., 05 2014. Code coverage for suite evaluation by developers.

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

[22] Guimaraes, M. A., Fernandes, L., Riberio, M., d'Amorim, M., Gheyi, R., 2020. Optimizing mutation testing by discovering dynamic mutant subsumption relations. In: Proceedings, 13th International Conference on Software Testing, Validation and Verification.

[23] Hehner, E., 1993. A Practical Theory of Programming. Springer-Verlag.

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

[25] Hoare, C., Oct. 1969. An axiomatic basis for computer programming. Communications of the ACM 12 (10), 576–583.

[26] Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. In: Procedings, 36th International Conference on Software Engineering. ACM Press.

[27] Inozemtseva, L., Holmes, R., 05 2014. Coverage is not strongly correlated with test suite effectiveness.

[28] Jia, Y., Harman, M., September 2008. Constructing subtle faults using higher order mutation testing. In: Proceedings, Eighth IEEE International Working Conference on Software Code Analysis and Manipulation. Beijing, China, pp. 249–258.

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

[30] Kaufman, S., Featherman, R., Alvin, J., Kurtz, B., Ammann, P., Just, R., May 2022. Prioritizing mutants to guide mutation testing. In: Proceedings, ICSE 2022. Pittsburgh, PA.

[31] Khaireddine, B., Martinez, M., Mili, A., April 2019. Program repair at arbitrary fault depth. In: Proceedings, ICST 2019. Xi'An, China.

[32] Khaireddine, B., Mili, A., May 2021. Quantifying faultiness: What does it mean to have n faults? In: Proceedings, FormaliSE 2021, ICSE 2021 colocated conference.

[33] Khaireddine, B., Zakharchenko, A., Martinez, M., Mili, A., March 2023. Toward a theory of program repair. Acta Informatica 60, 209–255.

[34] Kurtz, B., Amman, P., Delamaro, M., Offutt, J., Deng, L., 2014. Mutant subsumption graphs. In: Proceedings, 7th International Conference on Software Testing, Validation and Verification Workshops.

[35] Kurtz, B., Ammann, P., Offutt, J., 2015. Static analysis of mutant subsumption. In: Proceedings, IEEE 8th International Conference on Software Testing, Verification and Validation Workshops.

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

[37] Li, X., Wang, Y., Lin, H., 2017. Coverage based dynamic mutant subsumption graph. In: Proceedings, International Conference on Mathematics, Modeling and Simulation Technologies and Applications.

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

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

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

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

[42] Mili, A., 2021. Differentiators and detectors. Information Processing Letters 169.

[43] Mili, A., Frias, M., Jaoua, A., 2014. On faults and faulty programs. In: Hoefner, P., Jipsen, P., Kahl, W., Mueller, M. E. (Eds.), Proceedings, RAMICS 2014. Vol. 8428 of LNCS. pp. 191–207.

[44] Mili, A., Tchier, F., 2015. Software Testing: Operations and Concepts. John Wiley and Sons.

[45] Mills, H., Dyer, M., Linger, R., 1987. Cleanroom software engineering. IEEE Software 4 (5), 19–25.

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

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

[48] Namin, A. S., Kakarla, S., 2011. The use of mutation in testing experiments and its sensitivity to external threats. In: Proceedings, ISSTA.

[49] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., Harman, M., 2019. Mutation testing advances: An analysis and survey. In: Advances in Computers.

[50] Parsai, A., Demeyer, S., September 4-5 2017. Dynamic mutant subsumption analysis using littledarwin. In: Proceedings, A-TEST 2017. Paderborn, Germany.

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

[52] Someoliayi, K. E., Jalali, S., Mahdieh, M., 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). IEEE, pp. 584–588.

[53] Souza, B., December 2020. Identifying mutation subsumption relations. In: Proceedings, IEEE / ACM International Conference on Automated Software Engineering. pp. 1388–1390.

[54] Tengeri, D., Vidacs, L., Beszedes, A., Jasz, J., Balogh, G., Vancsics, B., Gyimothy, T., 04 2016. Relating code coverage, mutation score and test suite reducibility to defect density. In: Proceedings, 2016 IEEE 9th International Conference on Software Testing, Verification and Validation Workshops. pp. 174–179.

[55] Tenorio, M. C., Lopes, R. V. V., Fechina, J., Marinho, T., Costa, E., 2019. Subsumption in mutation testing: An automated model based on genetic algorithm. In: Proceedings, 16th International Conference on Information Technology –New Generations. Springer Verlag.

[56] Wei, y., Meyer, B., Oriol, M., 01 2010. Is branch coverage a good measure of testing effectiveness? Vol. 7007. pp. 194–212.

[57] Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings, ICSE.