Subsumption, Correctness and Relative Correctness

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.
Objective. Highlight their analogies, commonalities and overlaps; explore their potential for synergy and shared research goals.
Method. Introduce and analyze a minimal set of concepts that enable us to model these disparate research efforts, and explore how these models 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.

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. Four Research Directions

We consider four recent research directions in software engineering:

- In [16, 17] 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 [9, 26, 28, 18, 13, 12, 16, 29, 25], and tools are proposed to support its use [26].

- In [3] 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 [14, 7], and for quantifying program faultiness [15].

- In [27] 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 [20], 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 [6], 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.

1.2. Analogies and Overlaps

Even though these four 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 insights that they offer, as well as the research questions that they raise.

- *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. In section 3 we define detector sets for total correctness and partial correctness and use them to characterize absolute (partial and total) correctness and relative (partial and total) correctness.

- *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. In section 4 we give three definitions of differentiator sets, which depend on what we consider to be the outcome of an execution, under what condition we can compare two outcomes, and under what condition we consider that two comparable outcomes are identical. From these three definitions of differentiator sets, we derive three different definitions of mutant subsumption.

- *Subsumption as Relative Correctness.* By considering how relative correctness [21, 3] can be characterized by means of detector sets, and subsumption [16, 17] can be characterized by means of differentiator sets, we show in section 5 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.

- *Prevalence of Subsumption/ Relative Correctness.* Subsumption experiments generate mutants of a base program then seek to highlight subsumption relationships between the mutants; on the other hand, program repair experiments generate mutants as repair candidates then seek to highlight relative correctness relationships between them. Given that subsumption and relative correctness are equivalent, these two experiments are essentially identical, and ought to produce similar outcomes. Yet even a casual review of published literature in mutant subsumption and program repair reveals a paradox: whereas subsumption graphs are usually fairly dense (in terms of number of arcs for a given number of nodes), program repair methods struggle mightily to find solutions. This seems to suggest that subsumption algorithms may be prone to loss of precision, and/or that program repair algorithms may be prone to loss of recall. In section 6 we present a statistical model that estimates the prevalence of subsumption/ relative correctness relations between mutants.

- *Mutant Set Minimization as an Optimization Problem.* Subsumption was introduced as a means to minimize the cardinality of a mutant set [16]. 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, it does not include a formal definition of the optimization constraint. In section 7 we propose a definition of mutant set effectiveness, and 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.

In section 2 we introduce some elementary mathematics that we use in the paper to carry out our discussions. In section 8 we summarize our results, critique them, discuss threats to their validity, and sketch directions of further research.

2. Relational Mathematics

2.1. Sets

Because we use sets to represent program spaces, 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* $I = \{(s, s')|s' = s\}$ 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 ${}_{T\setminus}R = \{(s, s')|s \in T \wedge (s, s') \in R\}$.

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 $R \sqsubseteq R'$. When relations are used as specifications, refinement simply means that R' represents a stronger specification than R; in terms of pre/post specifications, this is equivalent to having weaker preconditions and stronger postconditions [5, 8, 24, 10].

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. 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 an 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 [19]. 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 [23], 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 1 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 [19, 8, 10]. 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:

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

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:

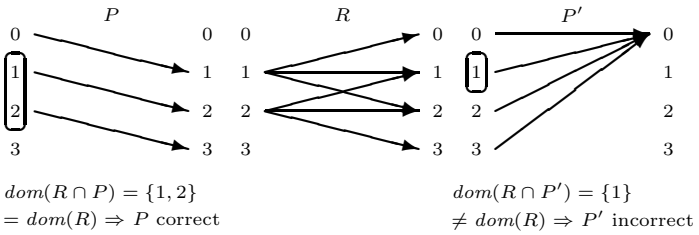

Figure 1: Total Correctness

$\forall s : s \in dom(R) \Rightarrow (\exists s' : (s, s') \in (R \cap P)$.
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 terminates,
- $(s, P(s)) \in R$ as: $P(s)$ satisfies the postcondition implied by R,

then we find that this is exactly the traditional definition of total correctness [19, 8, 5].

The following definition, due to [22], 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 [11, 5, 19, 8]. See Figure 2: Program Q is partially correct with respect to R because for any initial state of $dom(R)$ for which it terminates, 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.

3.3. Testing for Absolute Correctness

Whereas the distinction between total correctness and partial correctness has been at the center of studies of correctness and correctness verification, it has not been considered in studies of program testing; yet testing a program for total correctness is different from testing it for partial correctness, in the following sense. We consider a specification R on space S and a program P on S, and we let s be an element of $dom(R)$; we assume that execution of P on s diverges. The conclusion we draw from this observation depends on the standard of correctness we are considering:

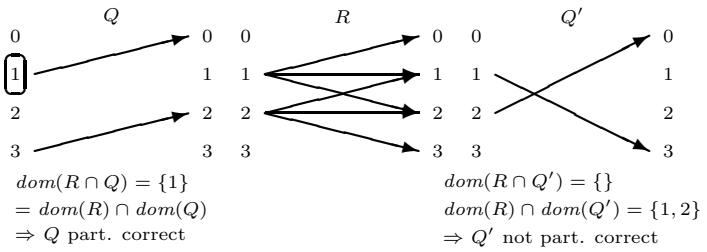

Figure 2: Partial Correctness

- Under total correctness, we conclude that program P fails the test and is proven incorrect with respect to R.
- Under partial correctness, in theory we conclude that program P has passed the test and is not proven to be incorrect. Though in practice most testers will draw no conclusion on P, but conclude instead that the test s is invalid, and choose another test.

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

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 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 1. *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.*

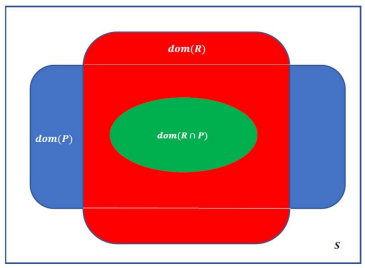	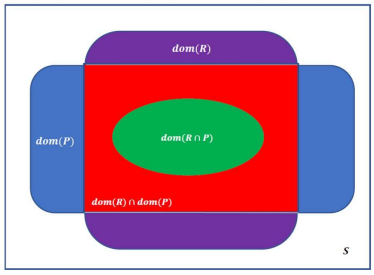
Total Correctness: $\Delta_T(R, P)$	Partial Correctness: $\Delta_P(R, P)$

Table 1: Detector Sets for Total and Partial Correctness

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 2. *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.

3.5. 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 definitionintroduces the concept of relative correctness. Since to be correct means to have an empty detector set (per Propositions 1 and 2), it is natutal to define relative correctness by means of inclusion of detector sets; whence the following definitions.

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:*

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

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:*

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

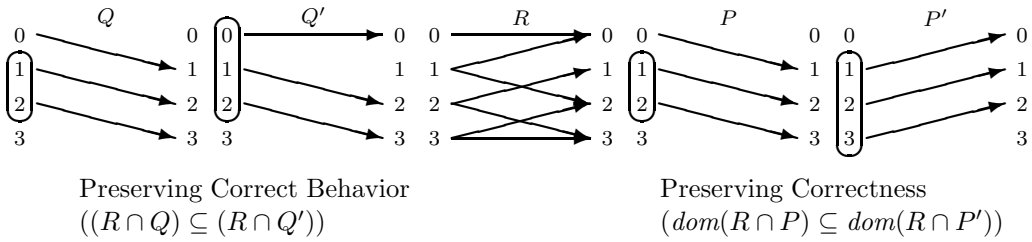

Figure 3: Relative Total Correctness

Figure 3 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 4 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).

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

$$\Delta_P(R, P) = dom(P) \cap \Delta_T(R, P).$$

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

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

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

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

We consider twelve candidate programs, listed in Table 2. Figure 5 shows how these candidate programs are ordered by relative total correctness; this ordering stems readily from the inclusion relations between the differentiator sets of the candidate programs with respect to R; the differentiator sets are given in Table 3, to allow interested readers to check Figure 5. The green oval shows those

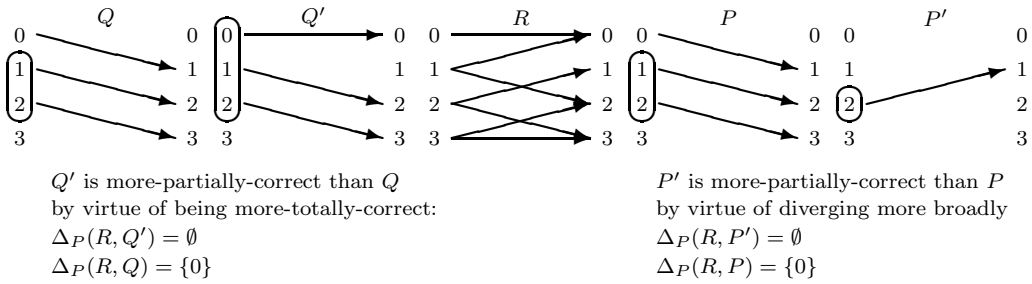

Figure 4: Relative Partial Correctness

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

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 $dom(R)$). This example is clearly artificial, but we choose it for its illustrative nature.

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 5.

Table 4 summarizes and organizes the definitions of correctness to help contrast them.

In [21, 4], total relative correctness is defined, not by comparing detector sets, as we do in Definition 4, but by comparing competence domains. The following Proposition provides that these definitions are equivalent, so as to assure us that prior results on relative correctness [14, 7] hold for the current definition.

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

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

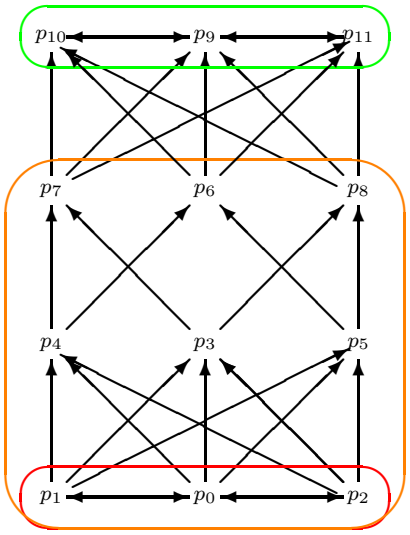

Figure 5: Ordering Candidate Programs by Relative Total (and Partial) Correctness with Respect to R

	Partial Correctness	Total Correctness
Absolute Correctness P is correct iff:	$\Delta_P(R, P) = \emptyset$	$\Delta_T(R, P) = \emptyset$
Relative Correctness P' is more-correct than P iff:	$\Delta_P(R, P') \subseteq \Delta_P(R, P)$	$\Delta_T(R, P') \subseteq \Delta_T(R, P)$

Table 4: Definitions of Correctness by Means of Detector Sets

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**

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:

- *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?

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

How we define differentiator sets depends on how we answer these questions. Though these questions may sound like mundane academic exercises in hair-splitting, 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 [27, 20]. 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 6. *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 6 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:

$dom(P \cap Q) = \{s | s^4 + 35s^2 + 24 = 10s^3 + 50s\}$.

Solving this equation in the fourth degree, we find:

$dom(P \cap Q) = \{s | 1 \leq s \leq 4\}$.

Taking the complement, we find:

$\delta_0(P, Q) = \overline{dom(P, Q)} = \{s | s < 1 \vee s > 4\}$.

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.

```
P:  {if (s<0) {while (s!=0) {s=s-1;}}
     else {s=pow(s,4)+35*s*s+24;}}
Q:  {if (s>5) {while (s!=5) {s=s+1;}}
     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:

$dom(P) = \{s | s \geq 0\}$.

$dom(Q) = \{s | s \leq 5\}$.

$P \cap Q = \{(s, s') | 0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s \wedge s' = 10s^3 + 50s\}$.

$dom(P \cap Q) = \{s | 0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s\}$.

By solving the equation ($s^4 + 35s^2 + 24 = 10s^3 + 50s$), we can simplify the formula of $dom(P \cap Q)$ as:

$dom(P \cap Q) = \{s | 1 \leq s \leq 4\}$.

Whence we find the following results for the strict differentiator set and the broad differentiator set of programs P and Q:

$\delta_1(P, Q) = \{0, 5\}$.

$\delta_2(P, Q) = \{s | s \leq 0 \vee s \geq 5\}$.

Interpretation:

- *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 (if the initial state is 0 or 5) or because one of

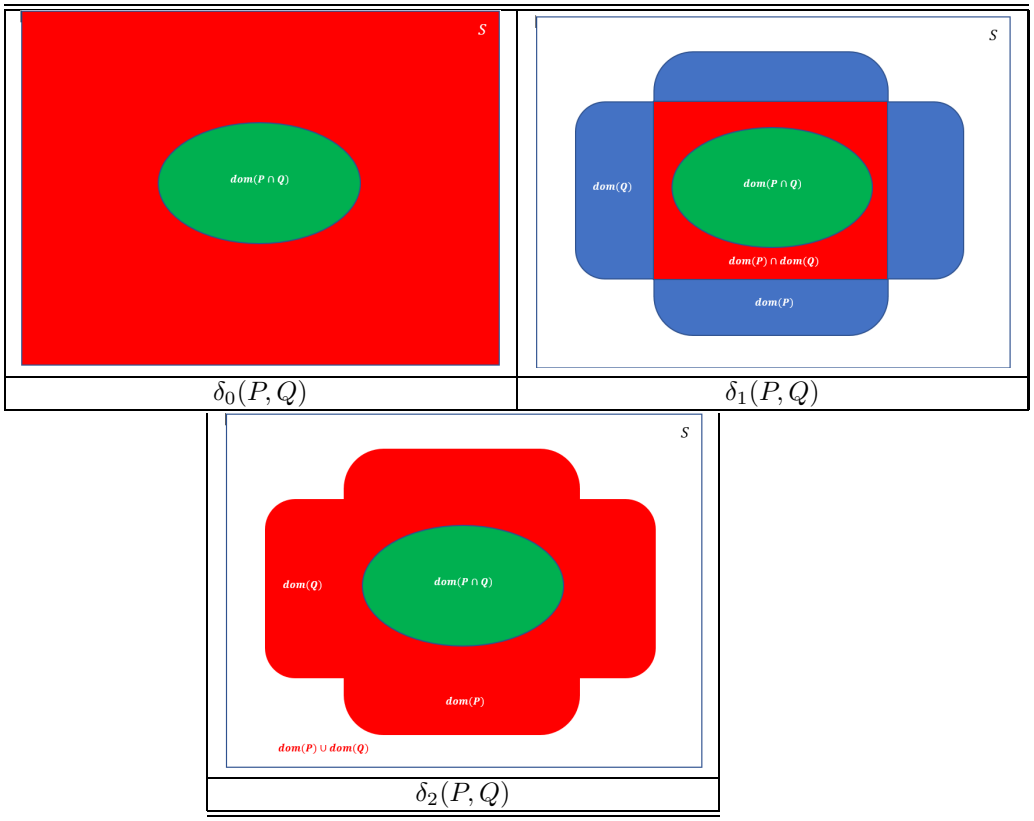

Figure 6: Three Definitions of Differentiator Sets (shown in red)

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 7 for illustration.

To illustrate strict and broad differentiator sets, we consider non-total versions of P and Q; we use artificial devices to make these programs non-total, to serve our illustrative purposes. We define programs P' and Q' as follows:

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 8 helps in this analysis.

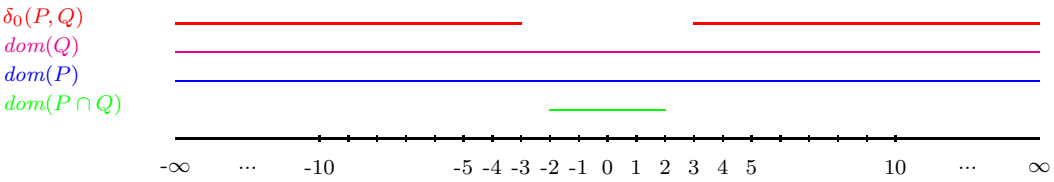

Figure 7: Basic differentiator Set for P and Q

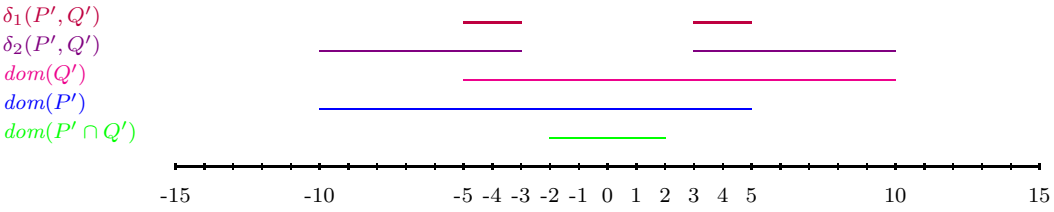

Figure 8: Strict and Broad differentiator Sets for P' and Q'

4.3. Differentiator Sets and Mutant Subsumption

In [16, 17] 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 4. *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 ([16, 17]) 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 7. *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 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* strict subsumption *if and only if:*

$$\emptyset \subset \delta_1(P, M') \subseteq \delta_1(P, M).$$

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* 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 [16, 17], 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. Given a program P and a set T, the program whose function is ${}_{T\backslash}P$ is:

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

5. 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 5. *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 $\emptyset \subset \Delta_T(P, M')$, which according to Proposition 1 is equivalent to: M' is not (absolutely) totally correct with respect to (the function of) P.

P2 $\Delta_T(P, M') \subseteq \Delta_T(P, M)$, 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 [16, 17, 9, 26, 28, 18, 12, 16, 29], represent essentially the same relation as the graphs of relative correctness seen in [2, 3, 15, 14]; 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 program, which represents the *maximally stubborn* mutants [30].

Proposition 6. *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.*

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:

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

According to Proposition 2, 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;}}
```

where `{abort();}` is a special program which fails whenever it is called, hence its function is empty. 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 3.5, though we changes 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 3.5, 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 (to say the least, since it converges for only three initial states) 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 9 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 5, except for the different names (`mi` vs `pi`). In Figure 9 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: {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

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

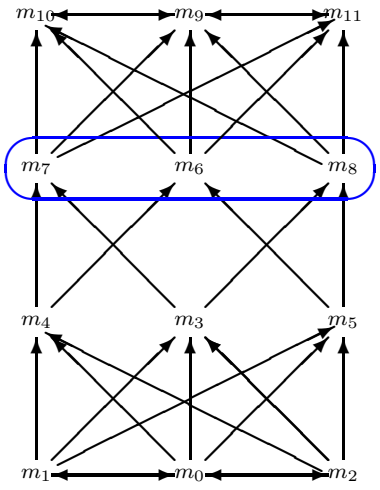

Figure 9: Ordering Mutants by Subsumption with respect to P

6. Statistical Analysis

In [6], Gazzola et al present a comprehensive survey of program repair; this survey highlights the predominance of search space size as the most critical concern in program repair. At its core, program repair is the act of making a program more-correct than it is [14, 7]; when the program has only one fault (which is what many program repair experiments assume), then making the program absolutely correct is indistinguisahble from making it relatively correct (i.e. more-correct than it is). Most program repair methods rely on common mutation operators to generate repair candidates, essentially the same kind of mutation operators that are used in mutation experiments; on the other hand, according to Proposition 6, the search for candidate repairs is based on the same criterion (relative correctness) as the determination of subsumption relations. This raises the following question:

- *If mutants are generated by the same operators, and pairs of mutants are compared using the same criterion (relative correctness $\Leftrightarrow$ subsumption), why is it so difficult to find program repairs (i.e. to reveal relative correctness relationships) yet so easy to reveal subsumption relations, as most subsumption graphs published in the literature are very dense (in terms of number of arcs over number of nodes)? Is this perhaps the result of loss of recall in program repair, or loss of precision in mtant subsumption?*

We are not going to answer this question in this paper, as that requires an empirical study well beyond the scope of this paper; but we will discuss the mathematics that make it possible to conduct such an empirical study.

Relative correctness is determined by checking an inclusion relationship between detector sets, and subsumption is determined by checking an inclusion relationship between differentiator sets. Hence both criteria can be modeled statistically by considering the following question: If we choose K non-empty subsets of a set of size T, what is the probability that any two subsets be in an inclusion relationship? Once we estimate this probability, we can answer two related questions, such as:

- What is the expected number of inclusion relationships between these K subsets? This would be the expected number of arcs in a subsumption graph of K nodes.
- What is the expected number of maximal subsets among the K subsets? This would be the size of the minimal set of mutants, as determined by subsumption.

These two questions are addressed in the next two subsections. It is important to note that by modeling subsumption and relative correctness with set inclusion, we are assuming that the differentiator sets / detector sets of two mutants with respect to a specification are statistically independent. Our statistical analysis is sound only to the extent that this asumption is valid.

6.1. Graph Density

By abuse of notation, we use the same symbol to denote a set and its cardinality. Given a set T and K non-empty subsets thereof, we ponder the question: what is the probability that any two subsets among K are in an inclusion relationship? 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:

$$prob(|D| = n) = \frac{\left(\begin{array}{c} T \\ n \end{array} \right)}{2^T - 1}.$$

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}

$\sum_{n=1}^{T} \frac{2^n - 1}{2^T - 1} \times \frac{\left(\begin{array}{c} T \\ n \end{array} \right)}{2^T - 1}.$

$=$ {simplification}

$\sum_{n=1}^{T} \frac{2^n - 1}{(2^T - 1)^2} \times \left(\begin{array}{c} T \\ n \end{array} \right).$

$=$ {factorization}

$\frac{1}{(2^T - 1)^2} \times \sum_{n=1}^{T} 2^n \left(\begin{array}{c} T \\ n \end{array} \right)$

$-\frac{1}{(2^T - 1)^2} \times \sum_{n=1}^{T} \left(\begin{array}{c} T \\ n \end{array} \right).$

$=$ {highlighting the binomial formula}

$\frac{1}{(2^T - 1)^2} \times \sum_{n=1}^{T} 2^n \times 1^{T-n} \left(\begin{array}{c} T \\ n \end{array} \right)$

$-\frac{1}{(2^T - 1)^2} \times \sum_{n=1}^{T} \left(\begin{array}{c} T \\ n \end{array} \right).$

$=$ {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:

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

6.2. Number of Maximal Nodes

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:

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

7. Mutant Set Minimization

Mutant subsumption has been introduced as a way to minimize the cardinality of a mutant sets, 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 (we want to minimize the cardinality of the mutant set to reduce the cost of mutation testing), 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? Once we have defined the effectiveness of a mutant set, we must then ponder the question: does elimination of a subsumed mutant preserve the effectiveness of a mutant set? We discuss these questions in this section.

The effectiveness of an artifact must be defined with respect to the purpose of the artifact, and must be assessed as function of its fitness for that purpose; hence the first question we must address prior to defining the effectiveness of a mutant set 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). This leaves two questions to address:

- How do we define/ quantify the quality of a test suite?
- How do we aggregate the quality of all the test suites vetted by a mutant set μ into a synthetic quality metric of the whole set?

These two questions are the subject of the next two sections.

7.1. Test Suite Effectiveness

We consider a program P on state space S and we consider a subset T of S. If we view T as a test suite, then its effectiveness must be analyzed with respect to two important attributes:

- *First, whether we are testing P for total correctness or partial correctness.*
- *Second, the specification with respect to which correctness is expected.*

To derive a quality metric of a test suite T, we first consider how to characterize an ideal test suite T, then we define a function that reflects how close we are to the ideal case. An ideal test suite can be characterized by two logically equivalent conditions: Test suite T is considered ideal to test program P for (partial or total) correctness with respect to specification R if and only if:

- If program P passes all the tests in T, we can be sure that P is correct with respect to R.
- If program P is not correct with respect to R then testing P on T will expose its incorrectness (i.e. P will fail at least one test in T).

This condition can be formulated as:

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

where $\Delta(R, P)$ is a stand-in for $\Delta_P(R, P)$ or $\Delta_T(R, P)$, 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 10. *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 $\sigma_{R,P}(T)$ and defined by:*

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

Measures of test suite effectiveness are usually referred to as *coverage metrics*, and usually refer to syntactic attributes (such as statement coverage, path coverage, condition coverage, etc); because the metric we introduce above refers to the semantics of the program, not to its syntactic representation, we refer to it as *semantic covereage*. A more meaningful distinction between traditional syntactic measures of coverage and ours is that whereas measures of syntactic coverage take numeric values (ranked by numeric inequality), our semantic coverage takes its values in subsets of S, ranked by set inclusion. In [1] Brinksma et al. introduce a semantic measure of test coverage that focuses, as we do, on the function of the program (rather than its syntax), and assigns different weights to different execution failures.

Definition 10 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 $\sigma^{PC}_{R,P}(T)$ and defined by:

$$\sigma^{PC}_{R,P}(T) = T \cup \overline{\Delta_P(R,P)},$$

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

$$\sigma^{TC}_{R,P}(T) = T \cup \overline{\Delta_T(R,P)},$$

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).

- $\sigma_{R,P}(T) = \emptyset$. 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 a failure 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.

- $\sigma_{R,P}(T) = S$. 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 1 and 2, the semantic coverage of any test suite T with respect to P and R is S.

See Figure 10; 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

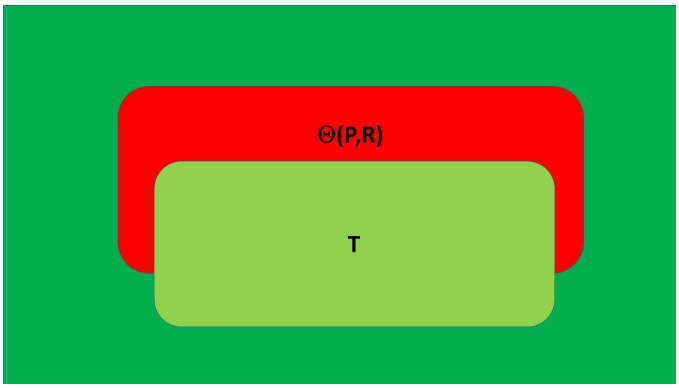

Figure 10: Semantic Coverage of Test T for Program P with respect to R (shades of green)

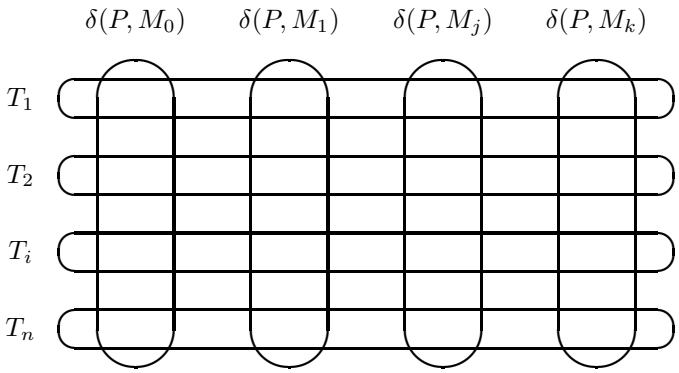

Figure 11: Test Suites $\{T_i\}$ Vetted by a Mutant Set$\mu = \{M_j\}$

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.

7.2. Mutant Set Effectiveness

Now that we know how to quantify the quality of a test suite, we seek to quantify the effectiveness of a mutant set; we argue that a set of mutants is as good as the test suites that it vets; hence we resolve to quantify the effectiveness of a mutant set by means of 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 (i.e. test suites that kill all the mutants of μ such that no subset thereof does):

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

Figure 11 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:

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

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

  Any test suite that is vetted by μ will have a semantic coverage at least equal to (i.e. a superset of) $\eta_A(\mu)$.

- *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.

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

  Any test suite that is vetted by μ will have a semantic coverage at most equal to (i.e. a subset of) $\eta_A(\mu)$.

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.

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

The following Proposition stems readily from the definition of semantic coverage, and gives a simpler expression to the assured effectiveness and the potential effectiveness of a set of mutants.

Proposition 7. *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:

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

This proposition stems readily from the fact that $\Delta(R, P)$ is independent of T, hence can be factored out of the effectiveness formulas. The following Proposition provides that removing a subsumed mutant in a mutant set preserves its potential effectiveness.

Proposition 8. *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 7, 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**

By virtue of this Proposition, we can infer that removing a subsumed mutant from a mutant set preserves the potential effectiveness of a mutant set. 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 9. *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, $\sigma_{R,P}(T_0) = T_0 \cup \overline{\Delta(R, P)}$. 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:

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

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 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
$\Delta_P(R,P) =$ $dom(R) \cap dom(P) \cap \overline{dom(R \cap P)}$	$\Delta_T(R,P) =$ $dom(R) \cap \overline{dom(R \cap P)}$

  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: $\Delta(R,P) = \emptyset$.
  - *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 iff: $\Delta(R,P') \subseteq \Delta(R,P')$.

Correctness Type	Property	Condition
Absolute Correctness	P is correct wrt R	$\Delta(R,P) = \emptyset$
Relative Correctness	P' is more-correct than P with respect to R	$\Delta(R,P) \subseteq \Delta(R,P')$

- *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 mutant 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,
- *Estimating Graph Density.* Once we model mutant subsumption and relative correctness as an inclusion relationship between two sets, it becomes possible to apply statistical methods to estimate the probability of occurrence of subsumption/ relative correctness relationships. We do so by estimating the probability that two random subsets of a finite set are in an inclusion relationship, and we use this probability to estimate the number of arcs in a subsumption graph, and the number of maximal nodes in such a graph.
- *Mutant Set Minimization as an Optimization Problem.* We argue that mutant set minimization is at its core an optimization problem and we resolve to identify the measure of mutant set effectiveness that published algorithms preserve as they remove subsumed mutants. We define the effectiveness of a mutant set as an aggregate of the effectiveness of the test suites that it vets, and we define the effectiveness of a test suite by considering the extent to which fewer elements of the detector set fall outside the test suite.

  Because test suites vetted by a mutant set may have widely varying semantic coverages, we aggregate the semantic coverages of vetted test suites

by computing their lower bound (assured effectiveness) and their upper bound (potential effectiveness). We prove that removing a subsumed mutant from a mutant set preserves (does not reduce) its potential effectiveness.

8.2. Assessment and Critique

Detector sets are an interesting concept because they link testing concerns with program correctness concerns: whereas they are introduced to represent test data that exposes the incorrectness of a program, they prove to be useful to characterize absolute correctness and relative correctness.

Differentiator sets are useful in the study of mutant subsumption, as they enable us to characterize mutant subsumption in uniform ways, across different interpretations of execution outcomes. In particular, they enable us to reason about subsumption while considering the possibility of program divergence; indeed, divergence is a *fact of life* in mutation testing, as several mutation operators are prone to generate divergent mutants by altering the guards that programmers use to prevent divergence. By defining several versions of differentiator sets, we generalize the concept of mutant subsumption without adding much complexity.

Highlighting the analogy between (strict) subsumption and relative (partial) correctness is interesting, becuse it opens the possibility of synergy between two research directions, one focused primarily on program testing, the other focused primarily on program correctness verification. This analogy stems from the identity between the detector set of partial correctness of a program with respect to a specification and the strict differentiator set of a program and its mutant. The potential synergy that stems from this analogy remains to be explored and exploited.

The statistical model that we have introduced to estimate the probability of occurrence of subsumption / relative correctness relationships between mutants of a base program provides a quantitative analysis of mutant subsumption and relative correctness. It is valid to the extent that the detector sets/ differentiator sets of different mutants of the same base program can be considered statistically independent.

8.3. Threats to Validity

When we apply the statistical model of section 6 to published subsumption graphs, we find that its estimates are lower than the actuals; one possible explanation is that our assumption of statistical independence between the differentiator sets of different mutants is unfounded.

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. While we did conduct some empirical experiments to validate the proposed metrics (semantic coverage, assured effectiveness, potential effectiveness), these are insufficient to draw any statistically significant conclusions.

8.4. Research Prospects

Our agenda of future research includes the following venues:

- Explore potential synergies between research in mutant subsumption and research in relative correctness, to share insights and advance the agenda of each branch.
- Improve the precision of the statistical analysis of mutant subsumption, by analyzing the possible correlations between differentiator sets of mutants of the same base program.
- Conduct empirical validations of the measures of semantic coverage (of a test suite) and assured effectiveness/ potential effectiveness of a mutant set, using benchmark programs and common mutant generators.

Acknowledgement

This research is supported in part by the National Science Foundation under grant numbre DGE1565478.

References

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

[2] 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.

[3] 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.

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

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

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

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

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

[25] 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.

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

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

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

[29] 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.

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