World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:18, No:1, 2024

Minimizing Mutant Sets by Equivalence and Subsumption

Samia Alblwi, Amani Ayad

***Abstract*—**Mutation testing is the art of generating syntactic variations of a base program and checking whether a candidate test suite can identify all the mutants that are not semantically equivalent to the base; this technique can be used to assess the quality of test suite. One of the main obstacles to the widespread use of mutation testing is cost, as even small programs (a few dozen lines of code) can give rise to a large number of mutants (up to hundreds); this has created an incentive to seek to reduce the number of mutants while preserving their collective effectiveness. Two criteria have been used to reduce the size of mutant sets: equivalence, which aims to partition the set of mutants into equivalence classes modulo semantic equivalence, and selecting one representative per class; and, subsumption, which aims to define a partial ordering among mutants that ranks mutants by effectiveness and seeks to select maximal elements in this ordering. In this paper, we analyze these two policies using analytical and empirical criteria.

***Keywords*—**Mutation testing, mutant sets, mutant equivalence, mutant subsumption, mutant set minimization.

I. Introduction: Minimizing Mutant Sets

MUTATION testing is a reliable way to assess the effectiveness of test suites, but it is also an expensive proposition. As a consequence, it is sensible to try to reduce the size of mutant sets, without loss of effectiveness. To the best of our knowledge, two broad families of criteria are used for the purpose of minimizing mutant sets:

- *Subsumption* [1]-[7]: A mutant M is said to subsume a mutant M' if and only if any test that kills M also kills M', and there exists a test that kills M. The subsumption criterion provides that if *M* subsumes *M'* then *M'* can be removed from the set of mutants.
- *Equivalence* [8]: In [8], Marsit et al. consider the equivalence relation of *semantic equivalence* between mutants and resolve to derive a minimal set of mutants as a set that includes one element from each equivalence class.

In this paper, we consider these two policies of mutant set minimization and compare them analytically and empirically. While subsumption is defined as an ordering relation between individual mutants, we argue that it is best viewed as an ordering relation between equivalence classes of mutants (modulo semantic equivalence). Indeed, if M subsumes M' and 10 mutants are equivalent to M and 10 are equivalent to M', we are still looking at a single instance of the ordering relation, not 100 instances. The implication of this remark is that subsumption ought not be applied as an alternative to equivalence, but rather alongside equivalence. We must first identify equivalence classes of mutants modulo semantic equivalence, then identify which equivalence classes are maximal by subsumption, and select a representative from each maximal equivalence class. This raises the question: if we have reduced a set of mutants to one representative per equivalence class, how much more reduction do we achieve by applying the criterion of subsumption? As a corollary of this question, it is also legitimate to ask: is the extra reduction in the set of mutants commensurate with the effort and risk of subsumption? Another aspect to consider is that, in [8], Marsit et al. derive the minimal set of mutants without analyzing all the mutants; rather, they estimate the amount of redundancy in the base program, and use a regression model to estimate the number of equivalence classes in the set of mutants. Knowing this number, we can shorten the search of equivalence classes significantly, as shown in [8].

Both criteria of mutant set minimization fail to explicitly specify a constraint under which the minimization is attempted. Indeed, all optimization problems aim to maximize or minimize an objective function under some constraints: The Knapsack Problem aims to maximize some benefit function under the constraint that the capacity of the knapsack is bounded; the Linear Programming problem aims to maximize some linear objective function under some affine constraints on the system parameters; and, the Maximum Flow problem aims to maximize the flow through a flow network subject to the topology of the network and the constraint that each are has a limited capacity. etc. Of course, we consider that the minimization of mutant sets assumes implicitly that discarded mutants do not reduce the effectiveness of the mutant set; but, in the absence of an explicit definition of what the effectiveness of a mutant set is and how to quantify it, it is difficult to make the case that the minimization algorithms are sound.

Another question that is raised by the use of subsumption as a criterion for mutant set minimization is the fact that the definition of subsumption is based on the outcome of programs and mutants being different. In order to give a precise meaning to this definition, we must agree on what is the outcome of a program, and when do we say that two outcomes are identical or distinct. This is less clear-cut than it may appear:

- What is a program's outcome? If a program or a mutant fails to terminate, due to an infinite loop, or a division by zero, or an array reference out of bound, do we consider these to be legitimate outcomes? Or do we define the outcome of a program only when the program's execution

Samia Alblwi is with New Jersey Institute of Technology, Newark NJ, USA (e-mail: sma225@njit.edu).

Amani Ayad is with Kean University, Union NJ, USA (e-mail: amanayad@kean.edu).

terminates normally?

- Also, even when a program does terminate normally, it is not always clear what we consider to be its outcome: Is it its final state or the output that the program delivers as a projection of the final state? For example, if a program permutes two variables x and y using an auxiliary variable z, what is the outcome of the program? Is it the final values of x, y and z, or just the final values of x and y?
- When do we say that two outcomes are identical? If two programs terminate normally for some common input, then (assuming we agree what variables represent the program`s outcome) we can tell whether they have the same outcome. But what about if one of them converges and the other fails to converge? Do we assume that they have distinct outcomes, or that their outcomes cannot be compared? What about the case when two programs fail to converge, do we consider that they have the same outcome (failure to converge) or that their outcomes are incomparable?

This matter will be discussed in Section IV, and subsumption will be (re)defined accordingly. In Section II, we introduce some elements of relational mathematics which we use in Section III to discuss differentiator sets and detector sets. In Section IV, we use the concept of differentiator sets to generalize the definition of mutant subsumption, and in Section V, we conduct an experiment in which we compare the results of minimizing a set of mutants by equivalence and by subsumption. In Section VI, we summarize our results, critique them, and sketch directions of future research.

II. Mathematics for Program Analysis

In this paper, we use relations and functions [3] to capture program specifications and program semantics. For the sake of simplicity, and without loss of generality, we consider homogenous relations on sets represented by program-like declarations. Modeling the program behavior by homogenous relations encompasses the case where we want to model it by a relation from inputs to outputs: It suffices to add an input stream and an output stream as state variables. We, generally, denote sets (referred to as spaces) by S, and elements of S (referred to as states) by lower case s, specifications (binary relations on S) by R and programs (functions on S) by P, Q. We denote the domain of a relation R (or a function P) by dom(R) (dom(P)). Because we model programs and specifications by homogenous relations/functions, we usually talk about initial states and final states; we may talk about inputs to refer to the initial value of the input stream and outputs to refer to the final value of the output stream. A specification R includes all the (initial state, final state) pairs that the specifier considers correct: hence the domain of a specification R (dom(R)) includes all the initial states for which candidate programs must make provisions. A program P includes all the initial state/final state pairs (s, s') such that if P starts execution in initial states, it terminates normally (i.e., after a finite number of steps, without raising an exception) in state s'. From this definition, it stems that the domain of program P (dom(P)) is the set of initial states s such that execution of P on s terminates after a finite number of steps and does not raise an exception (such as an overflow. underflow, division by zero, array reference out of bounds, etc.). Whenever a program P fails to terminate or raises an exception for initial state s, we say that it diverges on s; else we say that it terminates normally (or that it converges) on s.

III. Differentiators and Detectors

A. Differentiator Sets

Given a base program P and a mutant M, a differentiator set of M with respect to P is the set of initial states for which execution of P and M yield different outcomes. This concept is inspired from [9] and the definition we adopt in this paper is due to [10]. Following [10], we consider three different definitions of a differentiator set, depending on what we consider to be the outcome of an execution, under what condition we consider that two outcomes are comparable, and if they are, under what condition we consider them to be identical. Even though in this paper we use differentiator sets only in reference to a base program (say, P) and its mutant (say, M), our definitions talk about two arbitrary programs P and Q, regardless of what syntactic relation holds between them.

- *Basic Interpretation:* We assume that programs P and Q converge (terminate normally) for all initial states in S, and their outcome is their final state (or the final value of their output stream). Their differentiator set (which we denote by $\delta 0(P, Q)$) is the set of initial states for which their outcomes 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 this 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 or 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 this 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 inequality holds by set theory: $dom(P \cap Q) \subseteq dom(P) \cap dom(Q)$.

Definition1: 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.

- Under the basic interpretation, the differentiator set of two programs P and Q is denoted by $\delta 0(P, Q)$ and defined as:

$$\delta 0\ (P, Q) = dom\ (P \cap Q).$$

- Under the strict interpretation, the differentiator set of two programs P and Q is denoted by δ1(P, Q) and defined as:

$$\delta 1\ (P, Q) = dom(P) \cap dom(Q) \cap dom\ (P \cap Q).$$

- Under the broad interpretation, the differentiator set of two programs P and Q is denoted by δ2(P, Q) and defined as:

$$\delta 2(P, Q) = (dom(P) \cup dom(Q)) \cap dom\ (P \cap Q).$$

For the basic interpretation of program outcomes and outcome comparison, imagine that dom(P) and dom(Q) are both equal to (all of) S. Whenever we want to refer to a differentiator set of programs P and Q without specifying the interpretation, we use the notation $\delta(P, Q)$. 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:

$$P = \{(s, s') | s' = s^4 + 35s^2 + 24\}.$$

$$Q = \{(s, s') | s' = 10s^3 + 50^s\}.$$

Their intersection is:

$$P \cap Q = \{(s, s') | s4 + 35s2 + 24 = 10s3 + 50s \wedge s' = s4 + 35s2 + 24\}.$$

The domain of their intersection is:

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

Solving this equation in the fourth degree, we find:

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

Taking the complement, we find:

$$\delta 0\ (P, Q) = dom(P, Q) = \{s | s < 1 \vee s > 4\}.$$

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:

$$P = \{(s, s') | s \geq 0 \wedge s' = s4^4 + 35s^2 + 24\}.$$

$$Q = \{(s, s') | s \leq 5 \wedge s' = 10s^3 + 50^s\}.$$

From these definitions, we compute the following parameters:

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

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

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

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

By solving the equation ($s^4 + 35s^2 + 24 = 10s^3 + 50^s$), we can simplify the formula of dom (P∩ 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) ∩ dom(Q)), and programs P and Q return the same results for initial states in the interval [1...4] (dom (P ∩ Q)).
- *Broad Differentiator Set:* Any initial state outside the interval [1..4] exposes the difference between P and Q, either because they are both defined but give different results (if the initial state is 0 or 5) or because one of them terminates normally while the other diverges (for s greater than 5, P terminates normally but Q does not; for s negative, Q terminates normally but P does not).

B. Detector Sets

An ideal test suite is one that we can rely on to prove correctness: If program P runs successfully on test suite T, we want to be able to infer that P is correct; equivalently, we want that if P is incorrect, then testing P on test suite T ought to expose a failure of P. This leads us to the concept of detector set, i.e., the set of all the initial states on which program P violates its specification. This set is important because it enables us to characterize ideal test suites: ideal test suites are supersets of the program's detector set. This concept also enables us to compare (at least theoretically) test suites; a better

test suite is one that has a larger intersection with the detector set. But before we define detector sets, we must consider that there are two definitions of correctness, and these yield two distinct interpretations of what it means for a program to fall short of the standard of correctness; therefore, there are two possible definitions of detector sets, depending on what standard of correctness we adopt. We consider two definitions of program correctness: total correctness [11], [12] and partial correctness [13].

Definition2: According to [14], given a program P on space S and a specification (relation) R on S, P is said to be totally correct with respect to R if and only if:

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

According to [15], given a program P on space S and a specification(relation) R on S, P is said to be partially correct with respect to R if and only if:

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

These definitions are equivalent, to the traditional definitions of total and partial correctness [11]-[13]. The domain of (R ∩ P) is the set of initial states on which P satisfies R; we refer to it as the competence domain of P with respect to R. Since total correctness is a stronger property than (logically implies) partial correctness, we expect the set of tests that disprove the former to be superset of the set of tests that disprove the latter. We adopt the definitions of detector sets given in [15]; hence, we content ourselves in this paper with introducing these definitions and briefly commenting on them.

Definition3: According to [15], given a program P on space S and a specification R on S, the total detector set of P with respect to R is the set denoted by $\theta_T\ (P, R)$ and defined as the set of initial states on which execution of P produces an outcome that disproves the total correctness of P with respect to R (either the execution fails to converges or it does converge but produces a final state s′ such that (s, s′) ∉R). Given a program P on space S and a specification R on S, the partial detector set of P with respect to R is the set denoted by $\theta_P(P, R)$ and defined as the set of initial states on which execution of P produces an outcome that disproves the partial correctness of P with respect to R (the execution converges but produces a final state s′ such that (s, s′) ∉R).

When we want to refer to a detector set and do not wish to specify to which one we refer, we use the notation (P,R). The following proposition, according to [15], gives explicit expressions of the detector sets.

Proposition1: Given a program P on space S and a specification R on S, the total detector set and the partial detector set of P with respect to R are given by the following formulas:

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

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

A test suite T disproves the total (respectively, partial) correctness of P with respect to R if and only if (respectively):

$$T \cap \emptyset_T\ (R, P) \neq \emptyset.$$

$$T \cap \emptyset_P\ (R, P) \neq \emptyset.$$

If a test suite T disproves a correctness property, then so does any superset thereof.

Proposition2: A program P is totally (resp. partially) correct with respect to a specification R if and only if its total (resp. partial) detector set is empty, see Fig. 1.

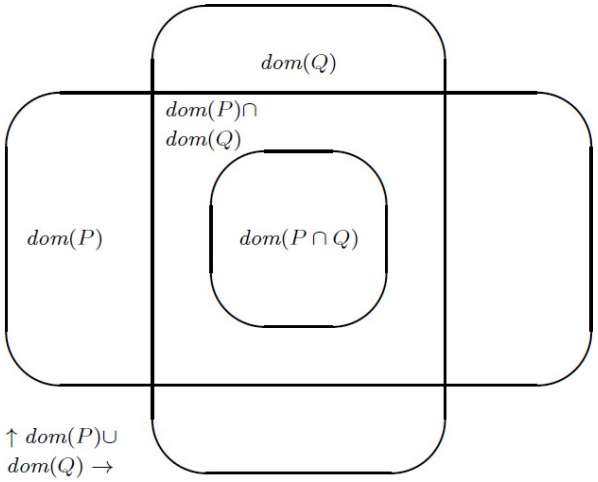

Fig. 1 Test data to expose behavior difference

IV. Revisiting Subsumption

In this section, we use differentiator sets to characterize the subsumption relation.

A. Subsumption of Convergent Programs

We consider a program P on space S and two mutants M and M′ of P, and we assume that P, M and M′ converge for all s in S. According to [10], [11], mutant subsumption is defined as follows:

Definition4*:* Given a program P on space S and two mutants M and M′ of P, we say that M subsumes M′ with respect to P if and only if:

- There exists an initial state s in S such that P and M compute different outcomes.
- For all initial states s in S, if M computes a different outcome from P on s, then so does M′.

The following proposition gives a simple characterization of mutant subsumption, using differentiator sets.

Proposition3: Given a program P on space S and two mutants M and M′, M subsumes M′ with respect to P if and only if:

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

Proof. The statement $P(s) \neq M(s)$ is equivalent to $s \notin dom(P \cap M)$; this, in turn, is equivalent to $s \in dom(P \cap M)$, which is equivalent to $s \in \delta_0(P, M)$. The existence of such an s means that δ_0 (P, M) is not the empty set.

The second condition of the definition of subsumption can be

written as:

$$\forall s : P(s) \neq M(s) \Rightarrow P(s) \neq M'(s).$$

This is equivalent to:

$$\forall s : s \in dom(P \cap M) \Rightarrow s \in dom(P \cap M').$$

By the definition of basic differentiator set, we rewrite this as:

$$\forall s : s \in \delta_0(P, M) \Rightarrow s \in \delta_0(P, M').$$

By set theory, this can be rewritten as:

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

B. Considering Divergence

Failure to converge is a condition that arises often, not least in mutation testing; for example, if we have a loop that visits all the cells of an array between indices 0 and N − 1 using the condition while (i < N), and we change the condition of the loop from < to ≤ (a common mutation operator), the mutant we generate will raise an exception (array reference out of bound) and fail to terminate normally. Hence, it is sensible to (re)define subsumption in a way that makes provisions for the possibility that the program or its mutants may diverge for some initial states. To this effect, we use the more general definitions of differentiator sets, namely $\delta_1(P, M)$ and $\delta_2(P, M)$.

Definition5. Given a program P on space S and two mutants M and M′ of P, we say that M subsumes M′ with respect to P if and only if:

$$\emptyset \subseteq \delta(P, M) \subseteq \delta(P, M').$$

We use δ (,) as a surrogate for any of the differentiator sets we have introduced in Section III: $\delta_0(,)$, $\delta_1(,)$, $\delta_2(,)$. A user may select a function among these depending on their interpretation of program outcomes. Note that even though we define subsumption as if it were a relation between individual mutants, we really refer to classes of equivalence of mutants modulo semantic equivalence. Since we are taking a semantic approach, we do not distinguish between mutants that compute the same function on S, even if they are syntactically distinct.

V. Minimizing a Mutant Set for jTerminal

We consider the Java benchmark program of jTerminal, an open-source software product routinely used in mutation testing experiments [5]. We apply the mutant generation tool LittleDarwin in conjunction with a test generation and deployment class that includes 35 test cases [5]. All our analyses of mutant equivalence, mutant redundancy, mutant survival, etc. are based on the outcomes of programs and mutants on this test suite (and carefully selected subsets thereof). For differentiator sets, we adopt the broad definition $\delta_2(P, M)$; hence, we consider that failure to converge is a legitimate execution outcome, and that failure to converge is comparable to a normal outcome, and is distinct therefrom. In other words, when the program fails to converge on some input x, we consider that execution of the program on x does have an outcome, and that this outcome is distinct from the outcome of a program that converges, regardless of the output generated by the convergent execution. Execution of LittleDarwin on jTermanal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants. For the sake of documentation, we list them below:

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

The remaining 46 mutants are semantically equivalent to the pre-restriction of jTerminal to the selected test suite. In this section, we generate a minimal mutant set out of the 48 mutants using respectively the criterion of equivalence and the criterion of subsumption.

A. Minimal Mutant Set by Equivalence

The procedure for generating a minimal mutant set outlined in [8] provides the following steps for executing:

- Parse the source code of jTerminal to compute its redundancy metrics: State redundancy (SRI, SRF); functional redundancy (FR); Non Injectivity (NI).
- Use the redundancy metrics to estimate the REM (Rate of Equivalent Mutants) of jTerminal: $REM = f\ (SRI, SRF, FR, NI)$.
- Use the REM of jTerminal to estimate the number of equivalence classes of the set of mutants modulo semantic equivalence: $K = NEC(N, REM)$, where N is the number of (killed) mutants.
- Using K and N, estimate the expected number of mutants that we must inspect (among N) before we encounter K distinct mutants: $H = NOI(N, K)$.
- Inspect the mutants one by one, comparing them against previously inspected mutants, until we find K distinct mutants, or we inspect H mutants in total.
- We adopt the resulting set of distinct mutants as a minimal set of mutants that preserves (approximately) the same functionality as the original set of N mutants.

Because in this case the number of mutants is not very large, and because we want to obviate the uncertainties that stem from estimating the redundancy metrics, then REM, then K, then H, we resolve to inspect all 48 mutants and compare them to each other to find K distinct mutants. We find that out of the 48 mutants under consideration, the following 30 are distinct from each other. We find μE =

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

We compute the detector set of each of these 30 mutants, which we use to derive minimal test suites that kill all the mutants. To this effect, we record the detector sets on a two-dimensional array where the mutants are represented in columns and the test data are represented in rows. We iterate through the following two steps until the array is empty.

- We select the test data that kill the most mutants.
- We remove the row that corresponds to the selected data, as well as the columns of all the mutants that the data kills.

Because there are several instances where more than one rows have the same maximal number of mutants, we may (and typically do) generate several minimal test suites. We list 10 minimal test suites generated according to this procedure; the numbers refer to the line of code where the data are generated in the original test class. For our purposes, these numbers uniquely identify the test data. Interestingly, all these sets have exactly 11 elements. We find:

$TE1 = \{t90, t118, t133, t168, t185, t189, t191, t209, t215, t239, t280\}$.
$TE2 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.
$TE3 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t284\}$.
$TE4 = \{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.
$TE5 = \{t90, t118, t133, t168, t185, t189, t191, t203, t209, t239, t280\}$.
$TE6 = \{t90, t118, t133, t168, t185, t189, t191, t209, t215, t239, t280\}$.
$TE7 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t284\}$.
$TE8 = \{t90, t118, t133, t168, t185, t189, t191, t203, t207, t239, t284\}$.
$TE9 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t280\}$.
$TE10 = \{t90, t118, t133, t168, t185, t189, t191, t209, t215, t241, t284\}$.

By construction, this test suite kills the 30 mutants of the minimal mutant set. Because the mutants outside the minimal mutant set are semantically equivalent to mutants of the set, the test suites above also kill the 48 killable mutants.

We notice that $TE1$ and $TE6$ are identical different selections made when two or more test data kill the same member of mutants may ultimately yield the same minimal test suite.

B. Minimal Mutant Set by Subsumption

To apply the subsumption criterion, we consider a representative from each of the 30 equivalence classes of the 48 killable mutants and test them pairwise by comparing their broad differentiator sets (δ_2 (P, M)). Then, we isolate the maximal mutants, i.e., those that are not subsumed by any other mutants.

We find the following minimal set of mutants:

$\mu S = m1, m19, m23, m24, m25, m27, m44, m45, m48, m51, m60$.

We compute the broad detector sets of these mutants, which are (by construction) much smaller than those of the mutants selected by equivalence; and, we apply the same procedure as above to derive minimal test suites that kill all these mutants. We find:

$TS1 = \{t90, t114, t118, t133, t168, t185, t189, t191, t207, t239, t284\}$.
$TS2 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.
$TS3 = \{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.
$TS4 = \{t90, t118, t133, t168, t185, t189, t191, t209, t215, t241, t280\}$.
$TS5 = \{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.
$TS6 = \{t90, t118, t133, t168, t185, t189, t191, t203, t209, t241, t284\}$.
$TS7 = \{t90, t118, t133, t168, t185, t189, t191, t203, t207, t239, t280\}$.
$TS8 = \{t90, t114, t118, t133, t168, t185, t189, t191, t207, t241, t280\}$.
$TS9 = \{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.
$TS10 = \{t90, t118, t133, t168, t185, t189, t191, t203, t207, t241, t284\}$.

A cursory inspection of the minimal test suites generated from the minimal mutant set derived by equivalence and the minimal mutant set derived by subsumption reveal that some of the test suites are identical. For example, TE2 is identical to TS2; and TE4 is identical to TS3. It is possible, even likely, that the set of all the minimal test suites that kill all the mutants of μE is the same as the set of all the minimal test suites that kill all the mutants of μS. To be certain, we need to generate all the minimal test suites for each mutant set; this is currently under investigation.

VI. Concluding Remarks

In this paper, we have considered two policies for minimizing a set of mutants and tried to analyze and compare them using analytical and empirical arguments. Some of the premises of our comparative study include the following:

- Subsumption is not a relation between individual mutants; rather it is a relation between equivalence classes of mutants, modulo semantic equivalence.
- As a consequence of this premise, the subsumption policy is not orthogonal to the equivalence policy; rather it must be mindful/cognizant of the equivalence relation and must identify equivalence classes prior to identifying subsumption relations between classes.
- If we quantify the effectiveness of a mutant set by the minimal test suites that it vets, then the empirical study of Section V is a resounding endorsement of subsumption, since it appears to vet the same test suites with one third the size (nine mutants vs. 30) and the test suites it vets kills all

(48) of the killable mutants of the base program.

- The results of Section V, to the extent that they are valid, may also be interpreted to mean that if the set μE vets the same minimal test suites as μS, then it may be sufficient to generate μE.

Among the contributions of this paper, we mention:

- A reformulation of subsumption using differentiator sets, and a generalization of subsumption to take into consideration the possibility that the base program or the mutants fail to converge.
- The observation that maximal mutants by subsumption feature minimal detector sets and are in fact what Yao et al. [16] refer to as stubborn mutants.
- A generalization of subsumption to apply, not to equivalence classes of mutants, but to sets thereof. This comes about naturally by generalizing the concept of detector sets to sets of (equivalence classes of) mutants.

Future research prospects include completing the experiment of Section V by computing all the minimal test suites of μE and μS and comparing them. Also, we envision to apply the generalized definition of subsumption that ranks sets of mutants rather than individual (equivalence classes of) mutants: it would be interesting to see whether this generalized formula enables us to reduce further the minimal set of mutants while preserving its effectiveness.

REFERENCES

[1] M. A. Guimaraes, L. Fernandes, M. Ribeiro, M. D'Amorim, and R. Gheyi, "Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations," 2020. doi: 10.1109/ICST46399.2020.00029.

[2] Y. Jia and M. Harman, "Constructing subtle faults using Higher Order mutation testing," 2008. doi: 10.1109/SCAM.2008.36.

[3] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," 2014. doi: 10.1109/ICSTW.2014.20.

[4] X. LI, Y. WANG, and H. LIN, "Coverage-Based Dynamic Mutant Subsumption Graph," *DEStech Transactions on Computer Science and Engineering*, no. mmsta, 2018, doi: 10.12783/dtcse/mmsta2017/19661.

[5] A. Parsai and S. Demeyer, "Dynamic mutant subsumption analysis using little darwin," 2017. doi: 10.1145/3121245.3121249.

[6] B. Souza, "Identifying Mutation Subsumption Relations," 2020. doi: 10.1145/3324884.3418921.

[7] M. C. Tenório, R. V. V. Lopes, J. Fechine, T. Marinho, and E. Costa, "Subsumption in mutation testing: An automated model based on genetic algorithm," in *Advances in Intelligent Systems and Computing*, 2019, vol. 800 Part F1. doi: 10.1007/978-3-030-14070-0_24.

[8] I. Marsit *et al.*, "The ratio of equivalent mutants: A key to analyzing mutation equivalence," *Journal of Systems and Software*, vol. 181, 2021, doi: 10.1016/j.jss.2021.111039.

[9] D. Shin, S. Yoo, and D. H. Bae, "A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, 2018, doi: 10.1109/TSE.2017.2732347.

[10] A. Mili, "Differentiators and detectors," *Information Processing Letters*, vol. 169, 2021, doi: 10.1016/j.ipl.2021.106111.

[11] D. Gries, *The Science of Programming*. 1981. doi: 10.1007/978-1-4612-59831.

[12] A. Blikle, "Zohar Manna. Mathematical theory of computation. McGraw-Hill Book Company, New York etc. 1974, x + 448 pp.," *Journal of Symbolic Logic*, vol. 44, no. 1, 1979, doi: 10.2307/2273714.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun ACM*, vol. 12, no. 10, 1969, doi: 10.1145/363235.363259.

[14] R. G. H. V. R. B. J. D. G. by Harlan D. Mills, *Principles of Computer Programming: A Mathematical Approach*. Allyn & Bacon, 1986.

[15] M. Ali and T. Fairouz, *Software Testing Concepts and Operations*, vol. XXXIII, no. 2. 2015.

[16] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings International Conference on Software Engineering*, 2014, no. 1. doi: 10.1145/2568225.2568265.