Measuring Effectiveness of Mutant Sets

Rahul Gopinath*, Amin Alipour†, Iftekhar Ahmed‡, Carlos Jensen§, and Alex Groce¶
Department of EECS, Oregon State University
Email: *gopinath@eecs.orst.edu, †alipour@eecs.orst.edu, ‡ahmed@eecs.orst.edu, §cjensen@eecs.orst.edu, ¶agroce@gmail.com

***Abstract*—Redundancy in mutants, where multiple mutants end up producing the same semantic variant of a program, is a major problem in mutation analysis; a measure of effectiveness that accounts for redundancy is an essential tool for evaluating mutation tools, new operators, and reduction techniques. Previous research suggests using the size of the disjoint mutant set as an effectiveness measure.**

We start from a simple premise: test suites need to be judged on both the number of unique variations in specifications they detect (as a variation measure), and also on how good they are at detecting hard-to-find faults (as a measure of thoroughness). Hence, any set of mutants should to be judged on how it supports they allow these measurements.

We show that the *disjoint mutant set* has two major inadequacies — the *single variant assumption* and the *large test suite assumption* — when used as a measure of effectiveness in variation, which stem from its reliance on minimal test suites. We show that when used to emulate hard to find bugs (as a measure of thoroughness), *disjoint mutant set* discards useful mutants.

We propose two alternative measures: one is oriented toward measuring variation and not vulnerable to either the *single variant assumption*, or the *large test suite assumption*; the other is oriented towards measuring thoroughness, and provides a benchmark of these measures using diverse tools.

I. Introduction

Software engineering relies on mutation analysis [1], [2] as the primary means of ascertaining quality of a test suite. Mutation analysis involves exhaustive generation of a syntactically defined set of faults (called mutations), and evaluation of the ability of a test suite to detect the resulting variants of the program. The ratio of faults detected to faults generated is taken as the mutation score of a test suite, and is taken to correlate with the ratio of detectable variants to total number of variants. Daran et al. [3], Andrews et al. [4], [5], Do et al. [6], and more recently Just et al. [7] suggest that mutation analysis is capable of generating faults that resemble real bugs, the ease of detection for mutations is similar to that for real faults, and the effectiveness of a test suite in detecting real faults is reflected in its mutation score. The terms used in this paper are given in Box 1.

A key concern in mutation analysis is whether the generated mutants are sufficient to generate all possible variants of the program. Given that not all mutants produce unique variants, another key concern [8] is to avoid generation of redundant mutants, using only a single representative mutant for each variant. Researchers have identified various techniques [9]–[24] for reducing redundancy in generated mutants: these include selective mutation, static subsumption, mutation clustering to identify similar mutants, static analysis of generated mutants, and other approaches. Recently [25] it was found that there is limited utility in mutation reduction strategies compared to random sampling, and it was more worthwhile to investigate additional effective mutation operators, rather than strategies for removal of mutants, however intelligent.

Whether it is for comparison between two mutation reduction strategies, or evaluation of effectiveness of a new mutation operator, one requires a measure of effectiveness of a set of mutation operators. Mutation analysis is a means of assessing the quality of a test suite, and any measure of effectiveness should consider how well a given set of mutants achieves this objective. A test suite should be judged on two main criteria: it should be able to detect and prevent as many unique variants as possible (measure of variation), and secondly, it may also be judged on how good it is in detecting subtle bugs (as a measure of thoroughness). Hence, we have corresponding requirements against which a set of mutants may be judged. A given set of mutants may be judged by the ratio of the unique variants it contains to the size of the full set of possible variants. It may also be judged by how hard to detect the variants induced by its mutants are.

Considering the first requirement, given a set of mutants, and a reduction strategy, one may judge the effectiveness of the strategy by the fraction of original unique variants that the reduction strategy was able to preserve in the reduced set. That is, for an ideal reduction strategy, each mutant in the reduced set should correspond to a unique variant, and each variant in the original set of mutants should have a unique mutant associated with it in the reduced set. Theoretically [26]–[28] this can be done by running *all possible* test sets — T^U against each mutant, identifying its unique signature, and removing those mutants that are *subsumed*[1]. This is however, undecidable as a consequence of Rice's theorem [30]. Ammann et al. [27] suggests a compromise. Rely on the minimal test suite (denoted as T_d), a subset of the available test suite T, such that T_d kills all mutants in the complete set M, and identify the minimal subset of mutants (we denote it by M_d) that requires at least all the tests in T_d to completely kill (called the disjoint set by Kintis et al. [31], and minimal mutants by Ammann et al. [27]). We adopt the moniker *disjoint mutant set* because this was the original name, and also because we propose an alternative minimal set of mutants. There is however, a problem.

Mutation reduction techniques and newer mutation operators are not added with a particular test suite in mind. They take into account static characteristics to include or exclude certain mutants, and the set of mutants should not change with the

[1]Traditionally [26], [28], [29] subsumption is based on all execution paths. That is, $kill(m_a, T^U) \implies kill(m_b, T^U)$ when m_a subsumes m_b. However, Ammann et al. [27] use dynamic subsumption, which relies only on the *available test set*. That is, $kill(m_a, T) \implies kill(m_b, T)$ when m_a subsumes m_b.

Box 1 Terms used

Fault: A fault is an erroneous part of a program, the syntactic source of a semantically wrong behavior [32].
Mutation: A mutation is a fault that was introduced into the program by a mutation tool. We do not distinguish between real faults and mutations in this paper.
Mutant: A mutant is a program with a fault in it. Traditional mutation is typically *first-order*.
Program behavior: The behavior of a program is the set of runtime properties that can be used to differentiate one *variant* of a program from another [32]. We measure these using *spectra* [33], and for the purpose of this research, especially the spectra that can be checked by a test case — the output spectrum — which is the record of the output of the program for a given input.
Program specification A program specification describes the *behavior* that a program must conform to.
Variant: A program or a mutant that shows a deviation in runtime behavior from the original program P.
Unique variant: A fault produces a unique variant with respect to a set of faults when the variant it produces is not produced by any (other) fault in the set.
Redundant fault: A fault (or mutant) is redundant with respect to a set of faults when the variant it produces is not unique compared to the variants of the (other) faults in the set.

Box 2 Triangle

$$triangle(a,b,c) = \quad (a < b+c)\&(b < c+a)\&(c < a+b)$$

Mutants:

$$triangle_a(a,b,c) = \top\&(b < c+a)\&\top$$
$$triangle_b(a,b,c) = \top\&\top\&(c < a+b)$$
$$triangle_c(a,b,c) = (a < b+c)\&\top\&\top$$

Test cases:

$$t_1 : triangle(3,2,1) \rightarrow false$$
$$t_2 : triangle(1,3,2) \rightarrow false$$
$$t_3 : triangle(2,1,3) \rightarrow false$$

test suite used, especially if the mutants are used to judge test suite quality. Hence, the aim for a measure of effectiveness of mutants should be to identify unique variants that are produced from a set of mutants. The test suites are only incidental to this requirement.

Does the theoretical minimum[2] *disjoint mutant set* satisfy this requirement? Given that test suites are the best available tools to judge whether a mutant produces a unique variant or not, **is size of theoretical minimum using disjoint mutant set the best one can do?** Our aim here is to show that the disjoint mutant set mutants calculated is not the actual set of unique variants, and could underestimate that set. Further, we propose a better measure of effectiveness than size of minimal set.

The unique set of variants corresponding to a set of mutants is important as it is the real target of mutation reduction strategies relying on static analysis of the program and mutants. Test suite results are only incidental to the accurate determination of unique variants.

[2] Minimum set is the smallest sized set among minimal sets.

Expanding on the second requirement, it may be argued that identifying the actual hardest mutants to detect is important (not just the size of the set), and disjoint mutant set is a set of mutants that are hardest to detect. In other words, not all the unique variants should be considered with the same weight. Some of the variants are trivially detected by multiple test suites, and hence a measure of effectiveness ought to consider the value of trivial and non-trivial mutants too. **Does the theoretical minimum set using disjoint mutant set capture all the hardest mutants to detect?** and **Is the size of theoretical minimum set using disjoint mutant set the best measure of effectiveness in this respect?** Our aim here is to show that disjoint mutant set does not select all the hardest mutants, and that subsumption can be used to define hardness – and thus can be used to provide a more informative measure.

A. *Problems with size of disjoint mutant set as a measure of effectiveness*

The explicit assumptions made in theoretical disjoint mutant set are [27]: a comprehensive[3] test suite, and a fixed set of mutants. However, using the theoretical disjoint mutant set as the true set of unique variants involves a few more implied assumptions. Let us imagine that we have a large set M of non-redundant mutants (size $m > t$), each producing a different variant, and a *minimum* test suite T containing a set of test cases (size $t > 1$), which is adequate to kill every mutant in the set of mutants. Say we create a *super test case* t' by joining together all other test cases, and add it to T, creating T'. Plainly, both T and T' are adequate for M. Now, according to Ammann's definition, the size of disjoint mutant set is the same as the size of the corresponding minimal test suite size, which is t if we are using T, but 1 if we are using T'. However, we do know that the actual number of variants is m. Even if we assume that $t = m$ initially, the actual number of variants is not 1.

This suggests that if one is looking for the actual true set of unique variants, one has to assume that test cases are *small*, with no test case killing more than one variant. (If any test case kills more than one variant, the size of the minimal test suite will no longer correspond to the number of unique variants. This is the *single variant assumption*. The second assumption is that the number of test cases are at least equal to or greater than the number of unique variants — the *large test suite assumption*.

Even for *large* mutation adequate test suites, the size of the minimum test suite may be much smaller than the number of variants, because some test cases may detect more than a single variant. This makes size of corresponding disjoint mutant set a less than ideal measure for the number of variants.

The *large test suite assumption* manifests itself as two problems. We can only identify $|T_d|$ number of unique variants, and secondly we can only distinguish between $|T_d|$ such sets of mutants (varying between 1 to $|T_d|$ in number of unique variants).

[3] The word *comprehensive* is not defined by Ammann [27], but the rest of the paper suggests that it means mutation adequate test suite.

As a practical example, consider a set of tests for *triangle* (Box 2) $\{t_1, t_2, t_3\}$, with corresponding mutant kills $\{m_a, m_b\}, \{m_b, m_c\}, \{m_c, m_a\}$. Plainly, all the mutants produce different variants, and the three test cases are different from each other in terms of the variants they detect, with none subsumed by others. However, a minimal test suite based on mutation scores discards one of t_1, t_2, t_3, as not all three are required to maintain the mutation score. The size of the disjoint mutant set is same as the cardinality of the corresponding minimal test suite. This means that the theoretical minimum disjoint mutant set may discard mutants which represent actual unique variants.

There is a simple fix to this problem. Any two mutants can be considered to be representing two different variants if the test cases that kill them are different. We denote a set of mutants such that no pair have similar kill pattern as unique mutants M_δ. Such a set has a maximum size limit of 2^T, the maximum resolving power of the test suite, and is not vulnerable to either of the *large test suite assumption*[4] or the *single variant assumption*. The size of unique mutant set can be used as a measure to compare two mutant sets of the same size. To compare two tools, we use the ratio $R_\delta = \frac{|M_\delta|}{|M_k|}$, which provides the odds that a given mutant is unique.

The second question is more involved. In Box 2, we know that disjoint mutant set discards at least one of the mutants, even though all of them are equivalent in terms of the number of tests they detect, and the pattern of detection. The same is true in the first example too, where adding a super test case results in ignoring most of the useful mutants. Hence the question is, do we have an alternative? What exactly is a trivial mutant? How do we measure effectiveness? Is the size of the non-trivial set the best one can do as a measure of thoroughness?

We develop a theory of hyper-geometric representation of variants such that variants enclosed by similar volumes have similar effectiveness irrespective of the number of variants included. We also show that the variants at the surface of this volume (denoted by M_s) can be computed by a small modification to the computation of disjoint mutant set M_d. We show that the alternative does not result in discarding important mutants in the given example.

This suggests a simple measure of effectiveness (with consideration for triviality). We compute the effectiveness as the ratio of volume of the sphere enclosed to the maximal volume.

M_s is an alternative to M_d, and like M_d, selects the possible non-trivial variants in a set of mutants. However, unlike M_d, we do not advocate its size as the effectiveness measure.

For empirical analysis, we use large well tested real-world projects from Github, and diverse tools. As in previous work [27], we compare the measurements between multiple tools, and identify the fraction of unique mutant set expected from the mutants produced by each tool. For mutation reduction, we investigated the reduction in effectiveness due to sampling. We find that even though the size of unique mutant set decreases along with decrease in sample size, the volume ratio remains similar. This suggests that the random samples produce mutants that are as hard to detect as the full set of mutants. Our data as well as the subject programs are available for replication [34].

[4] It is still vulnerable to a limit of $2^{|T|}$, where $|T|$ is the size of *all* tests, is usually larger than the number of mutants. Given that $2^{|T|}$ grows much faster than $|T|$, with about 10 test cases sufficient for uniquely identifying $1,024$ variants, we do not consider this a practical problem.

Contributions:

- We identify the need for two different kinds of effectiveness measures for mutant sets. The first to measure the fraction of unique variants present (variation effectiveness), and the second to measure how good the given mutants are in emulating subtle bugs (measure for thoroughness).
- We show that the size of disjoint mutant set can not be used as a variation effectiveness measure due to two unstated assumptions — *the single variant assumption*, and the *large test suite assumption*.
- We provide an alternative for size of *disjoint mutant set* ($|M_d|$) — the size of *unique mutant set* ($|M_\delta|$), which is not vulnerable to *the single variant assumption* or *large test suite assumption*.
- We also show that disjoint mutant set is not the best set of most hard to detect variants, as it may miss variants that are equally hard.
- We develop a theory of geometric representation of variants and use it to provide an alternative measure for *disjoint mutant set*— the *surface mutant set*, and also provide the semantic interpretation through volume ratio ψ.
- We provide an empirical benchmark for the different measures using three different mutation tools, and a diverse set of real world projects.

The organization of this paper is as follows. Section II discusses our theoretical model. Section III details the methodology used for selecting tools and subjects for benchmark. Section IV analyzes the results, followed by a detailed discussion in Section V. Related works are given in Section VI. As in the previous publication by Ammann et al. [27], we do not provide a *threats to validity* section since the focus is on theory. The conclusion is given in Section VII.

II. Geometric Model

Our approach builds on the disjoint mutant set [31] formalized by Amman et al. [27] but extends it to provide fine-grained criteria for evaluating performance of different mutation techniques. We note that our formulation is very similar to the recently proposed theoretical model for mutation testing methods [35].

A. Terminology

Given a program P, and its test suite T, a mutation tool may generate a set of mutants M by injecting a set of mutations. The mutations are a subset of all faults that are possible for a given program, which is generated by a mutation

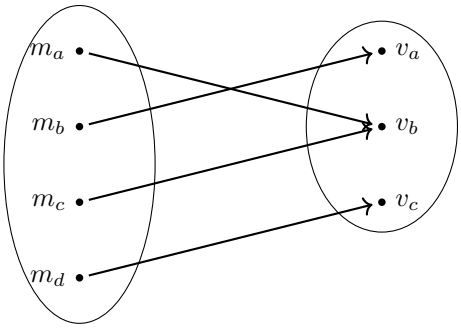

Fig. 1: Relationship between Mutants and Variants $M \twoheadrightarrow V$

tool. We use lowercase for elements, while sets are represented by uppercase — $m \in M$ is a single mutant, and $t \in T$ is a single test case. The set T^U represents *all possible* test cases for P. That is, $T \subseteq T^U$. The powerset of T is represented by 2^T, and contains $2^{|T|}$ elements. The mutants in M killed by a test suite T are given by $kill(T, M)$. Similarly, the tests in T that kill M are given by $cover(T, M)$.

$$kill : T \times M \to M$$
$$cover : T \times M \to T$$

A test suite T is *mutation adequate* for M if $kill(T, M) = M$. Two tests t_1 and t_2 are *indistinguishable* or *non unique* if they kill exactly the same set of mutants. That is, $kill(\{t_1\}, M) = kill(\{t_2\}, M)$. They are called *distinguished* or *unique* otherwise.

The same applies to mutants. Two mutants m_1 and m_2 are *distinguished* or *unique* if $cover(T, \{m_1\}) \neq cover(T, \{m_2\})$, and *indistinguishable* or *non unique* otherwise. A set of mutants is called unique mutant set if no pairs within the set are *indistinguishable*.

A mutant m_1 *dynamically subsumes* another m_2 if the tests that kill m_1 are guaranteed to kill m_2 (and m_1 can be killed). That is, $cover(T, \{m_1\}) \subseteq cover(T, \{m_2\})$

A set of tests are *minimal* if removal of any test case causes the mutation score to drop, and a disjoint mutant set corresponds to a minimal test set such that no mutant in the set dynamically subsumes another.

A *runtime variant* (or simply *variant*) is a program or a mutant that shows a deviation in runtime behavior from the original program P. Not all mutants express a detectable deviation and not all deviations are unique — equivalent, and redundant mutants exist. A *variant* may be detected by multiple test cases. We call a variant v_a that is detected only by a subset of test cases that detect another variant v_b, a *harder* variant to detect than v_b (and conversely *easier*). In terms of subsumption, the variant v_a can be said to *subsume* variant v_b. That is, $v_a \underset{subsume}{\geq} v_b$. We consider only dynamic subsumption by available test cases here. A variant v_a is *included* in the volume of a set of variants V if and only if at least one variant $v_b \in V$ subsumes v_a.

B. Approach

Imagine that we have an exhaustive set of variants V^U for any program P. Some variants may be easier to detect, and hence detected by multiple test cases. We know that mutants and variants have a *surjective* relationship. As shown in Figure 1, while multiple mutants can produce the same variant, multiple variants can not be associated with a single mutant, and there is at least one unique mutant for each variant generated (and at least one *fault* for any possible variant).

Consider an n dimensional volume with the number of dimensions given by $|T|$. A unit distance along any dimension represents the variant escaping the corresponding test case, while 0 in each dimension represents a detection of the variant by the test case corresponding to that dimension. With this formulation, the origin point is the variant that is detected by all test cases, and a volume of 2^n is the maximal volume possible. The inclusion of a variant now has a geometric representation. It is within the volume bounded by the variants that are already present. Effectively, this means that the inclusion needs to be checked only with the surface variants of the enclosed volume.

Thus suggests a way to evaluate the effectiveness of a set of mutants given a test suite, which is given by the ratio between the possible variants that could be *included* by the detected variant surface corresponding to the mutant set and the maximal volume of variants $2^{|T|}$. Let us denote the volume ratio by ψ, and the mutants in the variant surface by M_s[5].

Volume ratio ψ is a measure of thoroughness of a group of mutants. The more harder to detect, the larger the ψ of the mutant set. ψ is *independent* of the size of the project, or the size of test suite — all a volume ratio of 0.5 means is that of the possible variants, only 50% of the variants were included by the mutant set.

For example, consider Figure 2 where mutant m_1 is detected by test cases $\{t_1, t_2, t_3, t_4\}$, m_2 by $\{t_1, t_3, t_5\}$, and m_3 by $\{t_5\}$. This is a two dimensional representation of the five dimensional matrix. The central polygon represents the origin point $(0, 0, 0, 0, 0)$, representing a mutant that can be killed by all test cases. We say that m_3 subsumes m_2 (harder) but not m_1 (different), and the ratio of detected variants to total is $\frac{1}{2}$. The number of surface mutants here is $M_s = \{m_1, m_3\}$.

Unfortunately, measuring the ψ score becomes difficult as the number of test cases and mutants increases. That is, for computing ψ, the numerator is given by

$$\left|\bigcup_{i=1}^{|M|} A_i\right| = \sum_{J \subseteq \{1,2,..|M|\} \setminus \emptyset} (-1)^{|J|-1} \left|\bigcap_{j \in J} A_j\right|$$

where the number of terms is $2^{|M|} - 1$ — increasing exponentially. Computation of such a value is infeasible.

For example, if we have a set of test cases $\{a, b, c\}$, and a set of distinguishable mutants $\{A, B, C\}$ each killed by some combination of given tests, the total volume of mutants covered by our set of mutants $\{A, B, C\}$ would be

$$\psi = \frac{2^{|A|} + 2^{|B|} + 2^{|C|} - 2^{|A \cap B|} - 2^{|A \cap C|} - 2^{|B \cap C|} + 2^{|A \cap B \cap C|}}{2^{|\{a,b,c\}|}}$$

[5] Note that not all points in the volume may have corresponding mutants, such as due to test cases that check an exact subset of specification from another test case. We ignore these in favor of a simpler model.

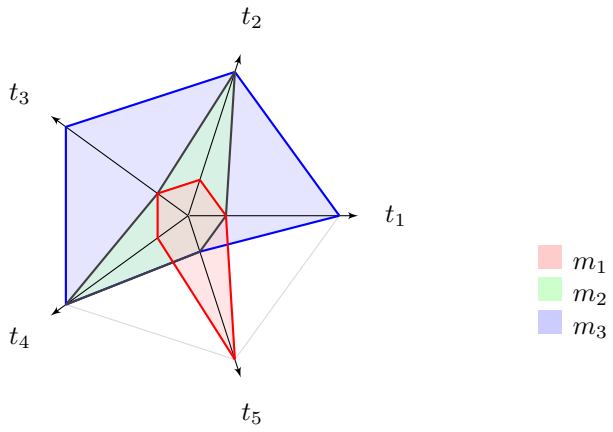

Fig. 2: Inclusion relationship between mutants. The outer points represents mutants not killed by respective test cases. The mutant m_2 (killed by test cases t_1, t_3, t_5) is subsumed by the mutant m_3 (killed by test case t_5 alone). However, the mutant m_1 (killed by test cases t_1, t_2, t_3, t_4) is different from both. The ψ score of m_3 (and the included m_2) is $\frac{2^4}{2^5} = 0.5$. Note that this is different from disjoint mutant set from M_d. With disjoint mutant set, mutants m_3 and m_1 is sufficient to cover the entire set of test suite. With our formulation, the above grouping will not cover a mutant killed by t_1, t_2, t_3

That is, we can expect $2^{|\{A,B,C\}|} - 1$ terms on the numerator. We can avoid this problem by using statistical approximation of the ratio. That is, randomly generate N possible combinations of $|T|$ test kills. Let N' be the number of these combinations that were *included* in the mutant surface M_s. The value $\frac{N'}{N}$ provides a reasonable approximation to ψ, refined to the requisite accuracy.

Note that the mutants corresponding to variants in the enclosing surface M_s are the same as the disjoint mutant set corresponding to the *entire* test suite T as per the definition given by Ammann et al. [27].

While the volume ratio can provide information about the ratio of included variants, due to the massive redundancy of mutations generated, the ratio may be very close to 1. Hence, another useful measure is the *surface correction* (s). Geometrically, it represents how close the given surface is to an n-sphere. A perfect spherical surface will have $s = 1$. For our purposes, it represents the mean number of test cases killing each surface mutant. The smaller the s, the more number of test cases that kill the surface mutants. A small s is an indication that the surface mutants are easy to detect. Hence, they may be improved further to fail in a smaller number of test cases, perhaps by engineering them using higher order mutants. The s can vary between $\frac{|M|}{|M|\times 1} = 1$ (each mutant fails only for a single test case) and $\frac{|M|}{|M|\times(|T|-1)} = \frac{1}{|T|-1}$ (each mutant fails for almost all test cases, but none are subsumed by others). Note that we can distinguish $(|T| - 1) \times |M|$ cases with s.

Each measure we have listed evaluates a different aspect of a given mutation set. The *surface mutant set* provides an alternative to the *disjoint mutant set*— the set of mutants that are shown to be hardest to detect out of the given set of mutants, and its effectiveness is given by *volume ratio*. The *surface correction* indiciates whether the selected mutants are trivial to detect.

C. Analysis

In this section, we compare the runtime complexity of three measures: disjoint mutant set M_d, surface mutant set M_s and unique mutant set M_δ.

Finding the true *minimum* test suite for a set of mutants is NP-complete[6]. The best possible approximation algorithm is Chvatal's [37], using a greedy algorithm where each iteration tries to choose a set that covers the largest number of mutants. This is given in Algorithm 1, and achieves an approximation ratio of $H(|M|)$[7] The complexity of greedy *set-cover* approximation is $O(\sum_{m\in M_t} |m|)$ [38] where M_t is the family of subsets of T. That is, M_t is the set of tests killing each mutant $m \in M$. The bound can be simplified to $O(|T| \times |M|)$, where $|T|$ is the number of test cases and $|M|$ is the number of mutants.

Algorithm 1 Finding the minimal test suite

```
function MinTest(Mutants, Tests)
    T ← Tests
    M ← kill(T, Mutants)
    T_min ← ∅
    while M ≠ ∅ do
        t ← random(max_t |kill({t}, M)|)
        M ← M \ kill({t}, M)
        T_min ← T_min ∪ {t}
    end while
    return T_min
end function
```

Adding a new mutant to an existing set of non-subsumed mutants is to simply iterate through the existing set of non-subsumed mutants to see if the new mutant is dynamically subsumed, and if not add it to the set. The algorithm to remove subsumed mutants is given in Algorithm 2.

The only difference between computing *disjoint mutant set* and *surface mutant set* is in which test suite gets passed to this algorithm. If the test suite that gets passed is the *minimal* test set (an approximation of the *minimum* test set), computed in Algorithm 1, then the minimal mutant set returned is the disjoint mutant set M_d. On the other hand, if what gets passed is the full test suite, then the surface mutant set M_s are computed.

The complexity of Algorithm 2, ignoring the growth of tests, is $O(m^2)$ where $m = |M|$. If we assume that the size of minimal test set is proportional to the size of the test set as shown in Figure 3 (consider also that in the worst case, a minimal (or minimum) test set is same as the full set), computing M_d has at least $O(m^2)$ complexity (ignoring the computation of minimal test suite). This is the same as

[6]This is the *Set Covering Problem* [27] which is NP-Complete [36].

[7]$H(n)$ is the n-th *harmonic number*. It is given by

$$H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq lnn + 1$$

Algorithm 2 Removing subsumed mutants

```
function RmSubsumed(Tests, Mutants)
    M ← kill(T, Mutants)
    T ← Tests
    M_min ← M
    while M ≠ ∅ do
        m ← random(M)
        M ← M \ {m}
        N ← M_min \ {m}
        while N ≠ ∅ do
            n ← random(N)
            N ← N \ {n}
            if cover({m}, T) ⊆ cover({n}, T) then
                M_min ← M_min \ n
                M ← M \ n
            end if
        end while
    end while
    return M_min
end function
```

computing M_s because it uses the same algorithm and has the same worst case test suite. Finally, the algorithm to compute *unique mutant set* is given in Algorithm 3

Algorithm 3 Finding the unique mutant set

```
function UniqueMutants(Tests, Mutants)
    M ← sort(Mutants)            ▷ Compare by cover(T,{m})
    T ← Tests
    M_ν ← ∅
    while M ≠ ∅ do
        m ← pop(M)
        M_ν ← M_ν ∪ {m}          ▷ Compare by cover(T,{m})
    end while
    return M_ν
end function
```

Of the three reduced mutation sets, the unique mutant set M_δ is the easiest to compute because the maximum computational complexity is for sorting — $O(m \times log(m))$[8].

III. Methodology

Our assertion that the unique mutant set is a better effectiveness measure than the disjoint mutant set can only be shown through reasoning. Hence we do not attempt to validate it here.

In previous work [27], Ammann et al. compared the mutation score for various selective mutation score strategies using the disjoint mutant set. However, we know [25] that mutation reduction strategies are severely limited in how much effectiveness they can gain, in theory and practice. In order to judge the quality of a reduction strategy one should compare the effectiveness of the strategy to the effectiveness expected from random sampling. Our previous research showed that popular mutation reduction strategies do not fare well in such a comparison [39].

Hence, for our empirical analysis, we have two main concerns. The first one is to benchmark how the different

[8]The complexity could even be reduced to linear because we are not sorting arbitrary integers. There is a fixed limit to the size of each element — the total number of test cases in the test suite.

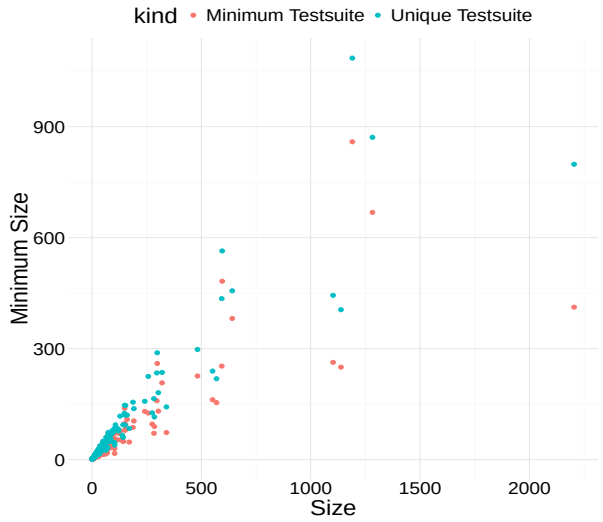

Fig. 3: Growth of mean minimal test suite size, and mean unique test suite size corresponding to growth of test suite size

measures perform for mutants from different tools. The second is to evaluate the effectiveness of random sampling. Computing how many of the mutants in the original set that belongs to any of M_d, M_s, M_δ, end up in the reduced random sample is not very useful. Because we are using random sampling, we are assured that the average ratio of sizes of these different sets to the size of full set M_k will remain the same due to the central limit theorem.

What we investigate instead is the second measure. That is, provided M_d is a measure of the effectiveness accounting for the ease of detection of mutants, our proposed replacement is M_s, and its actual impact is computed by the volume ratio ψ. Hence our question is how the effectiveness will change when we sample. Is the surface mutant set from the sample easier or harder to detect than the surface mutant set from the full set?

To benchmark new measures, it is important to capture the variations present in the real world use of these measures. The major avenues of variation are: variation due to mutant distribution in individual projects, variation due to the language used, and variation due to the mutation generation tools used (especially the phase during which the mutants were produced).

Unfortunately, the language choice is not orthogonal to other sources of variation. That is, language choice determines the projects, and the tool being used, which makes it difficult to compare different tools, and reflect variation introduced due to projects. Hence, we avoided variation due to language, and standardized on Java projects. Keeping the goal of real world projects that best represent real world software, we chose 25 large Java projects from Github [40] and Apache Software Foundation [41], that had large test suites. These projects, the size of their test suite (number of tests), and mutation scores are given in Table I.

We performed our evaluation with three tools: PIT 1.0,

TABLE I: Subject Programs, the size of test suite, and mutation scores

Project	TestSuite	Judy	Major	PIT
annotation-cli	126	42.420	43.275	59.375
asterisk-java	214	13.540	21.541	20.644
beanutils	1185	50.712	42.689	56.778
beanutils2	680	59.472	52.488	61.854
clazz	205	24.458	39.452	30.198
cli	373	71.174	76.606	86.137
collections	4407	76.988	58.631	34.687
commons-codec	605	92.725	73.524	82.661
commons-io	964	88.385	70.650	77.342
config-magic	111	55.191	29.800	60.690
csv	173	53.010	68.077	79.683
dbutils	239	44.234	65.205	47.340
events	206	77.136	70.030	59.949
faunus	172	2.552	58.653	49.066
java-api-wrapper	125	14.952	84.912	76.035
java-classmate	219	66.167	77.230	90.257
jopt-simple	566	84.500	79.315	94.499
mgwt	103	40.722	6.612	8.853
mirror	303	58.734	74.725	75.472
mp3agic	206	72.465	51.697	54.507
ognl	113	13.962	6.463	56.324
pipes	138	65.994	62.637	67.662
primitives	2276	93.350	71.326	35.705
validator	382	50.266	59.061	68.208
webbit	146	73.949	67.172	52.407

TABLE II: Subject programs, their size in LOC and mutants produced by each tool

Project	LOC	Judy	Major	PIT
annotation-cli	870	777	512	981
asterisk-java	29477	12658	5812	15476
beanutils	11640	6529	4382	9665
beanutils2	2251	990	615	2069
clazz	5681	2784	2022	5165
cli	2667	2308	1411	2677
collections	25400	1006	10301	24141
commons-codec	6603	44	7362	9953
commons-io	9472	164	6486	9799
config-magic	1251	527	650	1181
csv	1384	1154	991	1798
dbutils	2596	1159	677	1922
events	1256	2353	615	1155
faunus	9000	3723	3771	9668
java-api-wrapper	1760	929	611	1711
java-classmate	2402	1423	952	2543
jopt-simple	1617	497	695	1790
mgwt	16250	1394	6654	12030
mirror	2590	1316	449	1876
mp3agic	4842	1272	4822	7182
ognl	13139	8243	5616	21227
pipes	3513	590	1171	3001
primitives	11965	14	4916	11312
validator	5807	3320	3655	5846
webbit	5018	144	1327	3707

Judy 2.1.x, and Major 1.1.5. For each tool, we used the settings for the maximum number of operators to mutate.

Unlike other structural coverage measures such as statement, branch or path coverage, there is very little agreement on what constitutes an acceptable set of mutants in mutation analysis. This means that we can expect a wide variation in the number of mutants produced. The mutants produced by each tool on each program is given in Table II. Unfortunately, this also means that the mutation scores do not necessarily agree as we see in Table I. One of the culprits is the presence of equivalent mutants — mutants that do not produce a measurable semantic variation to the original program. To avoid skewing the results due to the presence of equivalent mutants, we removed the mutants that were not killed by any of the test cases we had. Note that some of the projects produced very small number of mutants, ignoring classes that could not be mutated due to various reasons. For the sake of completeness, and to show that these does not impact our conclusion, all the projects tool combinations that produced any number of mutants are included in our study.

First, we computed the different minimal sets M_d, M_s, M_δ along with the measures ψ and s for each project and tool combination. Next, we took 100 samples from each project, and computed the same measures for each sample.

IV. Empirical Comparison

We have two goals in our empirical comparison. The first one is to tabulate the different measures M_d, M_s, M_δ for each set of mutants M, along with the secondary measures ψ and s. The second goal is to compare the effectiveness of random sampling of a limited number of mutants to that of the full set of mutants.

A. Measures for the full set of mutants

The different measures for the full set of mutants are given in Table VI. As can be seen, except in a few cases, the volume being covered is indeed close to the full volume. However, we see that the surface correction s given in Table VI suggests that there is still quite a bit of improvement possible. The number of mutants necessary by variant surfaces M_s, along with disjoint mutant set M_d the size of unique mutant set M_δ, and the total number of mutants detected M_k are also given in Table VI. As expected, the surface mutant set M_s is larger than the disjoint mutant set M_d, and the unique mutant set M_δ is the largest out of the total detected M_k.

B. Measures for 100 mutants sampled

The mean measures for 100 mutants sampled randomly 100 times from each tool, for each project is given in Table VII. The columns with labels ψ and s are measures from the full set, and are given for comparison. ψ^μ, s^μ, and M_δ^μ are measures computed from the sample.

C. Comparing complete and sampled mutants

As we indicated previously, we know the behavior of M_δ under random sampling. Once we label some of mutants in M_k as belonging to M_δ, taking random samples from M_k can be expected to preserve the ratio $\frac{M_\delta}{M_k}$ on average. Hence, if we are using the size of unique mutant set as an effectiveness measure, the effectiveness will decrease in inverse proportion to the size of the sample. That is, a sample with $\frac{1}{10}$ mutants will have only $\frac{1}{10}$ of the original unique variants.

One measure that is actually of interest is the volume ratio ψ We evaluate the change in volume ratio using the linear regression:

$$\psi_{original} = \beta_\psi \times \psi_{sample}$$

Similarly, we evaluate the change in surface correction using the linear regression:

$$s_{original} = \beta_s \times s_{sample}$$

PIT

Project	ψ	s	M_s	M_d	M_δ	M_k
annotation-cli	1.00	0.46	31	26	109	583
asterisk-java	1.00	0.77	211	171	449	3195
beanutils	1.00	0.53	548	398	1568	5488
beanutils2	1.00	0.65	149	145	464	1280
clazz	1.00	0.59	64	49	150	1560
cli	1.00	0.52	207	136	787	2306
collections	1.00	0.68	898	797	2080	8374
commons-codec	1.00	0.63	488	351	1392	8228
commons-io	1.00	0.71	692	570	1598	7579
config-magic	1.00	0.58	60	49	203	717
csv	1.00	0.66	139	99	399	1433
dbutils	1.00	0.62	124	104	206	910
events	1.00	0.47	25	30	58	693
faunus	1.00	0.71	179	122	550	4744
java-api-wrapper	1.00	0.64	137	90	307	1301
java-classmate	1.00	0.80	217	184	550	2296
jopt-simple	1.00	0.45	177	131	579	1692
mgwt	1.00	0.82	85	74	168	1065
mirror	1.00	0.70	205	173	531	1416
mp3agic	1.00	0.71	160	116	733	3915
ognl	1.00	0.42	379	81	1998	11956
pipes	1.00	0.66	130	95	337	2031
primitives	1.00	0.64	685	445	1372	4039
validator	1.00	0.62	273	204	757	3988
webbit	1.00	0.64	114	90	315	1943

Major

Project	ψ	s	M_s	M_d	M_δ	M_k
annotation-cli	1.00	0.49	20	20	64	222
asterisk-java	1.00	0.75	170	142	295	1252
beanutils	1.00	0.56	428	344	892	1871
beanutils2	1.00	0.59	105	105	191	323
clazz	1.00	0.55	73	59	196	798
cli	1.00	0.56	174	130	471	1081
collections	1.00	0.59	995	910	2098	6040
commons-codec	1.00	0.60	364	267	1047	5413
commons-io	1.00	0.69	552	477	1138	4583
config-magic	1.00	0.55	50	45	93	194
csv	1.00	0.74	104	91	245	675
dbutils	1.00	0.53	69	60	155	442
events	0.91	0.37	22	10	28	431
faunus	1.00	0.75	126	103	378	2212
java-api-wrapper	1.00	0.55	72	42	270	519
java-classmate	1.00	0.72	136	108	281	736
jopt-simple	1.00	0.44	118	95	266	552
mgwt	1.00	0.84	77	70	119	440
mirror	1.00	0.72	124	112	198	336
mp3agic	1.00	0.70	149	108	614	2493
ognl	1.00	0.74	39	27	71	363
pipes	1.00	0.63	110	81	207	734
primitives	1.00	0.58	723	662	1565	3507
validator	1.00	0.60	218	168	478	2159
webbit	1.00	0.64	69	59	191	892

Judy

Project	ψ	s	M_s	M_d	M_δ	M_k
annotation-cli	1.00	0.39	29	20	74	330
asterisk-java	1.00	0.79	148	121	324	1714
beanutils	1.00	0.56	478	348	1364	3311
beanutils2	1.00	0.46	80	67	218	589
clazz	1.00	0.67	24	18	105	681
cli	1.00	0.56	157	106	553	1643
collections	1.00	0.61	148	130	246	775
commons-codec	0.50	0.29	6	4	11	41
commons-io	1.00	0.55	30	33	53	145
config-magic	1.00	0.48	42	33	107	291
csv	1.00	0.78	67	64	161	612
dbutils	1.00	0.58	78	73	129	513
events	0.98	0.34	25	21	35	1815
faunus	0.98	0.88	8	7	10	95
java-api-wrapper	1.00	0.85	12	10	21	139
java-classmate	1.00	0.80	116	98	276	942
jopt-simple	1.00	0.41	90	67	248	420
mgwt	1.00	0.81	58	55	120	568
mirror	1.00	0.72	148	127	359	773
mp3agic	1.00	0.51	88	54	333	922
ognl	1.00	0.72	20	14	67	1151
pipes	1.00	0.59	41	29	98	390
primitives	0.24	0.12	10	9	14	14
validator	1.00	0.68	140	102	341	1669
webbit	0.86	0.32	15	9	46	107

TABLE VI: Measures of all tools for full set of mutants

Projects	ψ	ψ^μ	ψ^σ	s	s^μ	s^σ	M_s^μ
annotation-cli	1.00	0.99	0.01	0.46	0.44	0.01	20.54
asterisk-java	1.00	1.00	0.00	0.77	0.66	0.00	55.71
beanutils	1.00	1.00	0.00	0.53	0.43	0.00	71.94
beanutils2	1.00	1.00	0.00	0.65	0.42	0.00	50.10
clazz	1.00	1.00	0.00	0.59	0.59	0.00	28.99
cli	1.00	1.00	0.00	0.52	0.45	0.00	52.69
collections	1.00	1.00	0.00	0.68	0.57	0.00	81.85
commons-codec	1.00	1.00	0.00	0.63	0.48	0.00	56.25
commons-io	1.00	1.00	0.00	0.71	0.57	0.00	78.39
config-magic	1.00	1.00	0.00	0.58	0.50	0.00	32.87
csv	1.00	1.00	0.00	0.66	0.57	0.00	43.65
dbutils	1.00	1.00	0.00	0.62	0.60	0.00	44.47
events	1.00	0.93	0.06	0.47	0.40	0.06	14.51
faunus	1.00	1.00	0.00	0.71	0.68	0.00	52.00
java-api-wrapper	1.00	1.00	0.00	0.64	0.64	0.00	44.09
java-classmate	1.00	1.00	0.00	0.80	0.68	0.00	55.21
jopt-simple	1.00	1.00	0.00	0.45	0.31	0.00	49.07
mgwt	1.00	1.00	0.00	0.82	0.71	0.00	35.84
mirror	1.00	1.00	0.00	0.70	0.55	0.00	55.77
mp3agic	1.00	1.00	0.00	0.71	0.56	0.00	40.62
ognl	1.00	1.00	0.00	0.42	0.57	0.00	34.58
pipes	1.00	1.00	0.00	0.66	0.68	0.00	49.61
primitives	1.00	1.00	0.00	0.64	0.62	0.00	76.64
validator	1.00	1.00	0.00	0.62	0.58	0.00	56.51
webbit	1.00	1.00	0.00	0.64	0.48	0.00	35.66

Projects	ψ	ψ^μ	ψ^σ	s	s^μ	s^σ	M_s^μ
annotation-cli	1.00	0.99	0.01	0.49	0.44	0.01	19.09
asterisk-java	1.00	1.00	0.00	0.75	0.67	0.00	49.69
beanutils	1.00	1.00	0.00	0.56	0.46	0.00	71.40
beanutils2	1.00	1.00	0.00	0.59	0.52	0.00	49.28
clazz	1.00	1.00	0.00	0.55	0.42	0.00	32.56
cli	1.00	1.00	0.00	0.56	0.47	0.00	55.17
collections	1.00	1.00	0.00	0.59	0.47	0.00	77.10
commons-codec	1.00	1.00	0.00	0.60	0.41	0.00	50.44
commons-io	1.00	1.00	0.00	0.69	0.55	0.00	68.67
config-magic	1.00	1.00	0.00	0.55	0.56	0.00	33.43
csv	1.00	1.00	0.00	0.74	0.60	0.00	42.15
dbutils	1.00	1.00	0.00	0.53	0.48	0.00	37.01
events	0.91	0.82	0.09	0.37	0.55	0.09	8.77
faunus	1.00	1.00	0.00	0.75	0.62	0.00	47.35
java-api-wrapper	1.00	1.00	0.00	0.55	0.50	0.00	32.98
java-classmate	1.00	1.00	0.00	0.72	0.70	0.00	52.05
jopt-simple	1.00	1.00	0.00	0.44	0.35	0.00	47.89
mgwt	1.00	1.00	0.00	0.84	0.76	0.00	37.38
mirror	1.00	1.00	0.00	0.72	0.65	0.00	61.07
mp3agic	1.00	1.00	0.00	0.70	0.55	0.00	39.96
ognl	1.00	1.00	0.00	0.74	0.73	0.00	25.29
pipes	1.00	1.00	0.00	0.63	0.68	0.00	46.80
primitives	1.00	1.00	0.00	0.58	0.43	0.00	81.71
validator	1.00	1.00	0.00	0.60	0.58	0.00	54.32
webbit	1.00	1.00	0.00	0.64	0.44	0.00	30.40

Projects	ψ	s	ψ^μ	s^μ	ψ^σ	s^σ	M_s^μ
annotation-cli	1.00	0.39	0.99	0.43	0.01	0.01	19.38
asterisk-java	1.00	0.79	1.00	0.75	0.00	0.00	50.84
beanutils	1.00	0.56	1.00	0.42	0.00	0.00	67.48
beanutils2	1.00	0.46	1.00	0.36	0.00	0.00	39.94
clazz	1.00	0.67	0.97	0.50	0.02	0.02	16.13
cli	1.00	0.56	1.00	0.44	0.00	0.00	47.54
collections	1.00	0.61	1.00	0.61	0.00	0.00	62.19
commons-codec	0.50	0.29	0.50	0.34	0.00	0.00	4.53
commons-io	1.00	0.55	1.00	0.52	0.00	0.00	29.46
config-magic	1.00	0.48	1.00	0.43	0.00	0.00	30.09
csv	1.00	0.78	1.00	0.65	0.00	0.00	31.25
dbutils	1.00	0.58	1.00	0.57	0.00	0.00	40.87
events	0.98	0.34	0.82	0.62	0.07	0.07	6.03
faunus	0.98	0.88	0.98	0.84	0.00	0.00	8.40
java-api-wrapper	1.00	0.85	1.00	0.79	0.00	0.00	12.37
java-classmate	1.00	0.80	1.00	0.69	0.00	0.00	47.90
jopt-simple	1.00	0.41	1.00	0.31	0.00	0.00	47.64
mgwt	1.00	0.81	1.00	0.66	0.00	0.00	29.65
mirror	1.00	0.72	1.00	0.56	0.00	0.00	50.50
mp3agic	1.00	0.51	1.00	0.42	0.01	0.01	32.19
ognl	1.00	0.72	0.99	0.66	0.01	0.01	15.14
pipes	1.00	0.59	1.00	0.59	0.00	0.00	27.50
primitives	0.24	0.12	0.24	0.11	0.00	0.00	11.44
validator	1.00	0.68	1.00	0.60	0.00	0.00	45.86
webbit	0.86	0.32	0.85	0.38	0.04	0.04	11.15

TABLE VII: Mean measures of all tools for 100-sample set of mutants

V. Discussion

Comparing the size of M_δ produced by each tool (Figure 4), we find that PIT produced on average 0.224 unique variants per mutant produced (standard deviation = 0.084). In comparison, Major produced on average 0.334 unique variants per mutant produced (standard deviation = 0.072) and Judy produced 0.306 unique variants per mutant produced (standard deviation = 0.195). We note that comparing the size of disjoint mutant set, PIT produced on average 0.065 disjoint mutants per mutant produced (standard deviation = 0.03). In comparison, Major produced on average 0.129 disjoint mutants per mutant produced (standard deviation = 0.033) and Judy produced 0.116 disjoint mutants per mutant produced (standard deviation = 0.121).

These results suggest an advantage for mutants produced from Major, closely followed by Judy as suggested in Figure 4. Given that $|M_\delta|$ is the size of variants in any given set of mutants, and R_δ is an effectiveness measure, it is worthwhile to estimate this effectiveness quickly for any set of mutants corresponding to a program generated by any tool. However, computing $|M_\delta|$ requires the full test run for the entire set of mutants. Is there a way to reliably estimate the size of $|M_\delta|$? Our results suggest that the ratio R_δ is highly correlated to the ratio of unique mutant set from sampling. That is,

$$R_\delta = \beta_1 \times R_\delta^{sample} + \beta_2 \times Tool$$

The regression $R^2 = 0.832$, and $p < 0.001$. Since the tool is significant, considering each tool separately:

$$R_\delta = \beta_1 \times R_\delta^{sample}$$

PIT $R^2 = 0.924$, and $\beta_1 = 0.307$, Major $R^2 = 0.901$, and $\beta_1 = 0.483$, Judy $R^2 = 0.666$, and $\beta_1 = 0.513$ (p < 0.0001). That is, we have moderate to strong correlation between R_δ and R_δ^{sample}, which may be used to predict R_δ from R_δ^{sample}. However, further research is needed in this area to determine

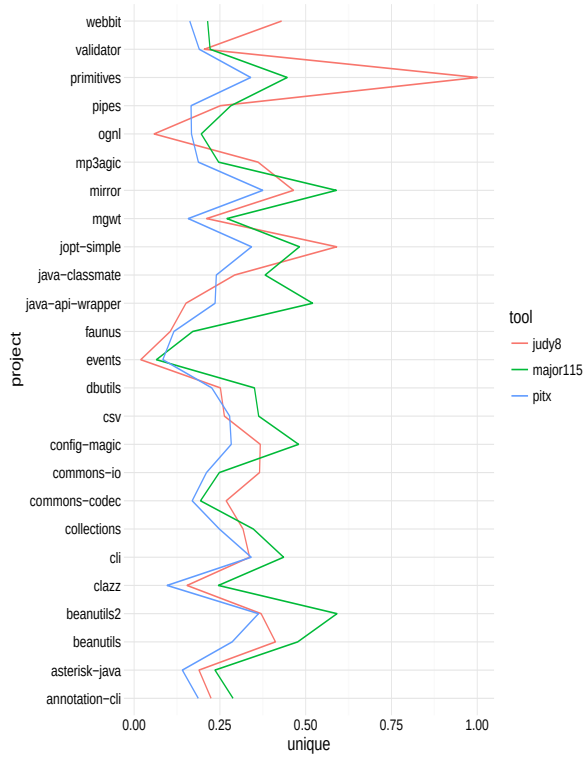

Fig. 4: Ratio of size of unique mutant set to full set of detected mutants (larger is better).

the effects of sample size, project characteristics, and mutation tool used.

Table VI suggests that for most of our projects ψ is very close to 1. That is, the volume enclosed by mutants is very close to the limit possible. This may be because most test cases are very targeted unit test cases that kill mutants within a given function. That is, for most mutants which are judged by a test suite, there is a small number of tests that detect each mutant. We also find that when we sample just 100 mutants (Table VII), the volume ratio remains close to the original volume ratio. That is, the coefficient β_ψ for PIT is 0.997, for Major is 0.996, and for Judy is 0.99, with $R^2 > 0.99$ and $p < 0.0001$. Further, the standard deviation of ψ suggests that there is very little deviation in any of the samples. A similar observation can be made for the surface correction s. The coefficient β_s for s_{sample} for PIT is 0.872, for Major is 0.875, and for Judy is 0.9, with $R^2 > 0.97$ and $p < 0.0001$. That is, the surface correction for samples is close to the surface correction detected for the full set.

Previous research [27] showed that using size of disjoint mutant set as a measure for effectiveness accounting for ease of detection, mutation reduction strategies fare poorly when it comes to maintaining effectiveness. Our results here show that even though the size of surface mutant set varies, the volume ratio ψ of samples remains close to the original volume ratio ψ of the full set of mutants, and the same case is true for the finer surface correction s. This result suggests that even if the size of surface mutant set or disjoint mutant set varies, the effectiveness accounting for ease of detection of the mutant set does not vary as much for random sampling. Does this effect extend to more intelligent mutation reduction strategies? Our previous research [25], [39] suggests that intelligent mutation strategies often fare poorly when compared against random sampling, and there is a need for further research in this area.

VI. Related Work

The idea of mutation analysis was first proposed by Lipton [1], and its main concepts were formalized by DeMillo et al. in the "Hints" [42] paper. The first implementation of mutation analysis was provided in the PhD thesis of Budd [43] in 1980. Previous research in mutation analysis [44]–[46] suggests that it subsumes different coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage [44]–[46]. There is also some evidence that the faults produced by mutation analysis are similar to real faults in terms of error trace produced [3] and the ease of detection [4], [5]. Recent research by Just et al. [7] using 357 real bugs suggests that the mutation score increases with test effectiveness for 75% of the cases, which was better than the 46% reported for structural coverage.

The validity of mutation analysis rests upon two fundamental assumptions: "The competent programmer hypothesis" – which states that programmers tend to make simple mistakes, and "The coupling effect" – which states that test cases capable of detecting faults in isolation continue to be effective even when faults appear in combination with other faults [42]. Evidence of the coupling effect comes from theoretical analysis by Wah [47], [48], empirical studies by Offutt [49], [50] and Langdon et al. [51]. The competent programmer hypothesis was quantified in our previous work [52].

One theoretical (and practical) difficulty in mutation analysis is identifying equivalent mutants — mutants that are syntactically different, but semantically indistinguishable from the original program, leading to incorrect mutation scores, because in general, identifying equivalent mutants is undecidable. The work on identifying equivalent mutants is generally divided into categories of prevention and detection [17], with prevention focusing on reducing the incidence of equivalent mutants [53] and detection focusing on identifying the equivalent mutants by examining their static and dynamic properties. These include efforts to identify them using compiler equivalence [17], [54], [55] dynamic analysis of constraint violations [56], [57], and coverage [58].

A similar problem is that of redundant mutants [8], resulting in a misleading mutation score. A number of studies measured the redundancy among mutants. Ammann et al. [27] compared the behavior of each mutant under all tests and found numerous redundant mutants. More recently, Papadakis et al. [17] used the compiled representation of programs to identify equivalent mutants. They found that on average 7% of mutants are equivalent while 20% are redundant.

Another fruitful area of research has been reducing the cost of mutation analysis, broadly categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [59]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis. The *do faster* approaches include mutant schema generation, code patching, and other methods to make the mutation analysis faster as

a whole. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling.

Various studies have tried to tackle the problem of approximating the full mutation score without running a full mutation analysis. The idea of using only a subset of mutants (*do fewer*) was conceived first by Budd [44] and Acree [60] who showed that using just 10% of the mutants was sufficient to achieve 99% accuracy of prediction for the final mutation score. This idea was further investigated by Mathur [9], Wong et al. [11], [61], and Offutt et al. [10] using the Mothra [62] mutation operators for FORTRAN. Lu Zhang et al. [63] compared operator-based mutant selection techniques to random mutant sampling, and found that random sampling performs as well as the operator selection methods. Recently, it was found [64] that $9,604$ mutants were sufficient for obtaining 1% accuracy for 99% of the projects, irrespective of the independence of mutants, or their total population.

Researchers have evaluated different mutation tools in the past [65], based on fault model (operators used), order (syntactic complexity of mutations), and selectivity (eliminating most frequent operators), mutation strength (weak, firm, and strong), and the sophistication of the tool in evaluating mutants. Our evaluation differs from their research in focusing on the semantic impact of mutants produced by different tools. Another closely related publication is MuRanker [66] where the authors use various distance functions to rank different mutants. While that study is similar to ours, the focus in that paper is on differentiating mutants by measures other than test failures (such as coverage), and in ranking mutants, not in coming up with a measure for a set of mutants. We note that our formulation of variants as situated in a hypergeometric volume parallels the recent theoretical approach of Shin et al. [35]. Finally, our study extends the previous research by Ammann et al. [27], in providing a related but more fine-grained measure.

VII. Conclusion

Research on newer mutation operators, and on eliminating redundant mutants requires some measure of effectiveness for a given set of mutants. As the primary aim of mutation analysis is to evaluate the quality of test suites, the measure of effectiveness should be based on how best to achieve this.

For a given test suite, there are two main concerns. Does it detect a reasonable fraction of the deviations in the progam specification? Secondly, does it detect and prevent subtle bugs?

To enable these measurements, a set of mutants should be able to provide as large a set of unique variants as possible, and also provide variants that are hard to detect.

Size of disjoint mutant set have been proposed [27] recently as a measure of effectiveness of mutation reduction techniques. However, a problem with this measure is that it is too tightly coupled with the test suite.

Most reduction strategies and mutation operators use static properties of the program in question to generate mutants. The aim of these strategies is to identify variants that are actually unique, not just to maintain the quality of a particular test suite. The test suite is only incidental to the measurement.

We showed that if we are to use disjoint mutant set as the set of all unique variants in a set of mutants, we need two more assumptions: the *single variant assumption* (that each test case kills exactly one variant), and the *large test suite assumption* (that the number of test cases is larger than the number of unique variants) beyond the stated assumptions of a comprehensive test suite, and a fixed set of mutants. Given that test cases routinely have more than a single assertion [67], and the number of mutants is usually much larger than the test suite size, these two assumptions may not be justified for real world test suites. On the other hand, if we were to use disjoint mutant set as the set of hardest to find mutants, we show that disjoint mutant set can discard some of important mutants that are as hard to detect as any other mutants. Hence disjoint mutant set is not an adequate solution as a measure of effectiveness of a mutant set.

Our contribution in this paper is to recognize the two different effectiveness critieria required. The first is the number of unique variants identified from a given set of mutants, and the second is a measure of ease of detection of a given set of mutants.

We proposed *unique mutant set* as an alternative that does not require either the *single variant assumption*, or the *large test suite assumption*. For the second critieria, we proposed the surface mutant set which preserves useful hard to detect mutants, and a semantics using *volume ratio* and *surface correction* that provides a more concrete explanation of what ease of detection entails.

It may be asked, why try to mitigate problems with mutant analysis by relying again on mutation detection by test suites? Why not use other criteria such as coverage information or program metrics instead?

Detection of mutants by test suites is still the closest, most effective, and most direct approach we have for evaluating the effectiveness of test suites. Other measures (such as most kinds of coverage) are subsumed by mutation analysis, and measures such as static information from program metrics are not suitable as a theoretical bound on the effectiveness of mutants, because they are dependent on particular languages. Furthermore, we are unsure about the information they provide about the variants, primarily because such information is highly indirect compared to mutant kills.

We provide a benchmark of the different measures $|M_d|$, $|M_s|$, and $|M_\delta|$ using three different tools (PIT, Judy, and Major) on 25 real world projects. Our empirical analysis shows that random sampling works reasonably well for maintaining the effectiveness (accounting for ease of detection) of a given set of mutants.

References

[1] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[2] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[3] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1996, pp. 158–171.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

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

[6] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[8] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 720–725.

[9] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Computer Software and Applications Conference.*, 1991, pp. 604–605.

[10] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *International Conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 100–107.

[11] W. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185 – 196, 1995.

[12] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[13] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[14] R. H. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator," in *ACM Annual Southeast Regional Conference*, ser. ACM-SE 47. New York, NY, USA: ACM, 2009, pp. 71:1–71:4.

[15] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *International Conference on Software Testing, Verification and Validation*, Luxembourg, 2013.

[16] M. E. Delamaro, L. Deng, N. Li, V. Durelli, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for c," in *International Conference on Software Testing, Verification and Validation*, Cleveland, Ohio, USA, 2014.

[17] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *International Conference on Software Engineering*, 2015.

[18] M. Patrick, R. Alexander, M. Oriol, and J. A. Clark, "Probability-based semantic interpretation of mutants," in *Workshop on Mutation Analysis*, 2014.

[19] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *International Conference on Software Engineering*. ACM, 2008, pp. 351–360.

[20] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for c," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[21] G. Kaminski, P. Ammann, and J. Offutt, "Better predicate testing," in *International Workshop on Automation of Software Test*. ACM, 2011, pp. 57–63.

[22] ——, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[23] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis." in *SEKE*, 2009, pp. 422–425.

[24] A. Derezińska, "A quality estimation of mutation clustering in c# programs," in *New Results in Dependability and Computer Systems*. Springer, 2013, pp. 119–129.

[25] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016.

[26] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 249–258.

[27] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 21–30.

[28] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *International Conference on Software Testing, Verification and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 176–185.

[29] M. F. Lau and Y. T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 247–276, Jul. 2005.

[30] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[31] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 300–309.

[32] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. I, no. 9036264, p. 844âĂŞ857, 1990.

[33] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '98. New York, NY, USA: ACM, 1998, pp. 83–90.

[34] R. Gopinath, "Replication data for unique minimal sets of mutatnts with variant surfaces," http://eecs.osusol.org/rahul/icst16/.

[35] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *International Conference on Software Testing, Verification and Validation*, 2016.

[36] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 1972.

[37] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms third edition*. The MIT Press, 2009.

[39] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "Do mutation reduction strategies matter?" Oregon State University, Tech. Rep., Aug 2015, http://hdl.handle.net/1957/56917. [Online]. Available: http://hdl.handle.net/1957/56917

[40] GitHub Inc., "Software repository," http://www.github.com.

[41] Apache Software Foundation, "Apache commons," http://commons.apache.org/.

[42] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[43] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1980, pp. 220–233.

[44] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1980.

[45] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[46] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, Tech. Rep., 1996.

[47] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–45, 2000.

[48] ——, "An analysis of the coupling effect i: single test data," *Science of Computer Programming*, vol. 48, no. 2, pp. 119–161, 2003.

[49] A. J. Offutt, "The Coupling Effect : Fact or Fiction?" *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, Nov. 1989.

[50] ——, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[51] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of systems and Software*, vol. 83, no. 12, pp. 2416–2430, 2010.

[52] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *International Symposium on Software Reliability Engineering*, Nov 2014, pp. 189–200.

[53] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," *International Conference on Software Engineering*, pp. 919–930, 2014.

[54] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations." DTIC Document, Tech. Rep., 1979.

[55] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[56] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[57] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," in *Workshop on Formal Methods in the Development of Software, WS-FMDS*, 2012, pp. 1–8.

[58] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[59] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[60] A. T. Acree, Jr., "On mutation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[61] W. E. Wong, "On mutation and data flow," Ph.D. dissertation, Purdue University, West Lafayette, IN, USA, 1993, uMI Order No. GAX94-20921.

[62] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt, and K. King, "An extended overview of the mothra software testing environment," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 1988, pp. 142–151.

[63] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 435–444.

[64] R. Gopinath, A. Alipour, A. Iftekhar, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *International Symposium on Software Reliability Engineering*. IEEE, 2015.

[65] M. Delahaye and L. Du Bousquet, "A comparison of mutation analysis tools for java," in *International Conference on Quality Software*. IEEE, 2013, pp. 187–195.

[66] A. S. Namin, X. Xue, O. Rosas, and P. Sharma, "Muranker: a mutant ranking tool," *Software Testing, Verification and Reliability*, 2014.

[67] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 214–224.