The Journal of Systems & Software 181 (2021) 111039

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The ratio of equivalent mutants: A key to analyzing mutation equivalence☆

Imen Marsit [a], Amani Ayad [b], David Kim [c], Monsour Latif [c], JiMeng Loh [c], Mohamed Nazih Omri [a], Ali Mili [c,*]

[a] University of Sousse, Tunisia
[b] SUNY Farmingdale, NY, USA
[c] NJIT, Newark NJ, USA

ARTICLE INFO

Article history:
Received 27 October 2020
Received in revised form 13 June 2021
Accepted 28 June 2021
Available online 14 July 2021

Keywords:
Mutation testing
Mutation equivalence
Redundancy
Ratio of equivalent mutants

ABSTRACT

Mutation testing is the art of generating syntactic versions (called mutants) of a base program, and is widely used in software testing, most notably the assessment of test suites. Mutants are useful only to the extent that they are semantically distinct from the base program, but some may well be semantically equivalent to the base program, despite being syntactically distinct. Much research has been devoted to identifying, and weeding out, equivalent mutants, but determining whether two programs are semantically equivalent is a non-trivial, tedious, error-prone task. Yet in practice it is not necessary to identify equivalent mutants individually; for most intents and purposes, it suffices to estimate their number. In this paper, we are interested to estimate, for a given number of mutants generated from a program, the ratio of those that are equivalent to the base program; we refer to this as the *Ratio of Equivalent Mutants* (REM, for short). We argue, on the basis of analytical grounds, that the REM of a program may be estimated from a static analysis of the program, and that it can be used to analyze many mutation related properties of a program. The purpose/ aspiration of this paper is to draw attention to this potentially cost-effective approach to a longstanding stubborn problem.

1. Mutant equivalence

Mutation is the art of generating syntactic variations of a program P, and is meaningful only to the extent that the syntactic modifications applied to P yield semantic differences; but in practice a mutant M may be syntactically distinct from the base program P yet still compute the exact same function as P. The existence, and pervasiveness, of equivalent mutants is a source of bias and uncertainty in mutation based analysis:

- If we generate 100 mutants of program P and find that some test suite T kills 80 of them[1], what we can infer about T depends on the number of equivalent mutants among the 20 surviving mutants: if we know, somehow, that 20 of the mutants are equivalent to P, then T has killed all the mutants that can be killed; if, on the other hand, we know that only five mutants are equivalent, then T has missed 15 mutants.
- If test suite T kills 80 mutants of P, it is important to distinguish between the case when T has killed 80 distinct mutants, and the case when it has just killed 80 times the same mutant, or something in between. To this effect, we need to assess to what extent the 80 mutants are distinct from each other, or equivalent to each other.

The issue of mutant equivalence has been the focus of much research recently (Ammann et al., 2014; Shin et al., 2018; Budd and Angluin, 1982; Budd et al., 1980; Chao and Chiu, 2014; Boehme, 2018; Zhang et al., 2019; Offut and Pan, 1997; Yao et al., 2014; Aadamopoulos et al., 2004; Papadakis et al., 2014; Schuler and Zeller, 2010; Just et al., 2013; Nica and Wotawa, 2012; Andrews et al., 2005; Namin and Kakarla, 2011; Just et al., 2014b; Gruen et al., 2009; Kintis et al., 2018; Wang et al., 2017; Hierons et al., 1999; Carvalho et al., 2018; Ayad et al., 2019b,a; Gopinath et al., 2016; Menendez et al., 2020; Shin et al., 2018; Just et al., 2017; Zhang et al., 2016). It is beyond the scope of this paper to do a survey of mutation equivalence research (see Papadakis et al., 2019 for a recent survey). But if we may indulge in a broad generalization, we would say that most research on mutation equivalence falls into one of two broad categories:

☆ Editor: W. Eric Wong.

* Corresponding author.
E-mail addresses: imen.marsit@gmail.com (I. Marsit), ayada@farmingdale.edu (A. Ayad), dk432@njit.edu (D. Kim), au42@njit.edu (M. Latif), loh@njit.edu (J. Loh), mohamednazih.omri@eniso.u-sousse.tn (M.N. Omri), mili@njit.edu (A. Mili).

[1] We say that test suite T *kills* the mutant M of program P if and only if execution of M on T produces a different outcome from execution of P on T.

https://doi.org/10.1016/j.jss.2021.111039
0164-1212/

- Research that infers equivalence from a local analysis of the mutation site; such methods are prone to loss of recall, as two programs may be locally distinct but still globally equivalent, due to non-injectivity of program functions (Clark and Hierons, 2012; Clark et al., 2019; Androutsopoulos et al., 2014).
- Research that infers equivalence from comparing global behavioral attributes; such methods are prone to loss of precision, because programs may have similar behavioral attributes yet still be semantically distinct (Kushigian et al., 2019; Nica and Wotawa, 2012).

Determining whether two syntactically distinct programs are semantically equivalent is known to be undecidable (Budd and Angluin, 1982). Notwithstanding this theoretical result, attempting to determine whether two programs are semantically equivalent is a non-trivial, costly, and error-prone exercise; it is in fact of the same nature and scale as determining whether a program is correct with respect to a specification (and if we knew how to do that, we probably would not need to test the program). Attempting to decide whether M mutants are equivalent to a base program is $O(M)$ times more difficult/impractical; attempting to determine whether M mutants are distinct from each other is $O(M^2)$ times more difficult/impractical. Most importantly, we find that for most practical applications, it is not necessary to identify equivalent mutants individually; it suffices to estimate their number. Even when it is important to identify equivalent mutants individually, knowing their number can be helpful in practice: whenever a mutant is killed, the probability that the surviving mutants are equivalent to P increases as their number approaches the estimated number of equivalent mutants.

In this paper we argue that research on mutation equivalence ought to complement efforts to determine equivalence between programs with efforts to estimate the probability that two programs (e.g. a base program and a mutant, or two mutants) are semantically equivalent. We find that estimating this probability can be relatively easy and inexpensive, yet enables us to answer many questions pertaining to mutation equivalence (as we illustrate in Section 9.1).

This paper gives a snapshot of our ongoing research on mutant equivalence, and builds on previously published results: In Mili et al. (2014) we introduce entropy-based software metrics, which aim to reflect the semantic attributes of software artifacts, rather than their syntactic representation (McCabe, 1976; Halstead, 1977; Fenton, 1991; Fenton and Pfleeger, 1997; Abran, 2012); while our original intent was to explore relationships between our semantic metrics and the fault tolerance of a software artifact, we find empirically in Marsit et al. (2017) that actually these metrics are correlated with the proneness of an artifact to produce equivalent mutants. In Ayad et al. (2019a,c) we discover why our metrics are correlated with a program's proneness to generate equivalent mutants, even though we meant them to reflect fault tolerance: The same attributes that make a program fault tolerant also make it prone to generate equivalent mutants; given that fault tolerance is made possible by redundancy, we argue that if we could quantify the redundancy of a program, we can use it to estimate the ratio of equivalent mutants that a program is prone to produce in any mutation experiment (Ayad et al., 2019a,c). In Marsit et al. (2018) we argue that the ratio of equivalent mutants of a program depends not only on the program, but also on the mutation operators that are applied to it; and we discusses the impact of mutation operators on the ratio of equivalent mutants.

This paper makes the following contributions/advances with respect to previous publications:

- *Automated Calculation of Redundancy Metrics*. In Section 3 we discuss the calculation of redundancy metrics using an automated tool developed with compiler generation technology (Ayad, 2019).
- *Regression Model Based on Automatically Calculated Redundancy Metrics*. In Section 4 we discuss the derivation of a regression model that uses the redundancy metrics, as they are computed by the automated tool, to estimate the ratio of equivalent mutants of a software artifact.
- *Equivalence-based Mutation Score*. In Section 6 we revisit the formula of equivalence-based mutation score (Ayad et al., 2019c), and propose a new formula, which excludes from consideration mutants that are estimated to be equivalent to the base program, and quantifies test suite performance by means of distinct mutants killed, rather than potentially redundant/duplicate mutants.
- *Dependency on Mutation Policy*. In Section 8 we discuss how the REM of a program depends on the mutation operators that are deployed on the program. We consider two orthogonal approaches: the first is to select some mutation policies, defined by sets of mutation operators, and derive regression models for each; the second is to derive regression models for common mutation operators applied individually, then try to infer the REM of a set of operators from those of the individual operators.
- *Sample Use Case*. In Section 9.1, we present an illustrative sample use case of our results; the purpose of this section is to illustrate the potential of our quantitative approach as we envision it, once our results are empirically validated.
- *Threats to Validity*. This work is primarily analytical: It offers a set of results about how to assess the proneness of a program to generate equivalent mutants, how to quantify this property, and how to use this quantification to analyze mutation-related properties of a program. Most of these steps are carried out on the basis of analytical arguments, using information theoretic functions and statistical identities. While many of the analytical arguments we put forth are compelling, much empirical validation is needed before this material can be put into practice; in Section 9.2 we explore the challenges of this empirical work.
- *Related Work*. In Section 10.2 we discuss a number of related research efforts, and highlight their relationship to our work; interestingly, many of these efforts share the same premise as our work, namely that mutation testing is an expensive proposition in practice, and that one way to mitigate this expense is to attempt to predict the outcome of mutation testing without actually conducting the testing experiments.
- *Preliminary Empirical Validation*. In this paper we start testing some of our assumptions, with varying degrees of success, as we report throughout the paper. Many of these assumptions require a great deal of work to be fully/credibly validated.

We do not view our approach as a panacea for mutation equivalence but rather as a possible venue of research that offers a lightweight cost-effective alternative to prevailing semantics-based approaches to a stubborn problem.

2. Redundancy: the mutants' elixir of immortality

In Yao et al. (2014) Yao et al. ask the question: *what are the causes of mutant equivalence?* Mutant equivalence is determined by two factors, namely the mutant operators and the program being mutated. For the sake of argument, we consider a fixed mutation policy (defined by a set of mutant operators) and we

reformulate Yao's question as: *What attribute of a program makes it prone to generate equivalent mutants?* A program is prone to generate equivalent mutants if it continues to deliver the same function despite the presence of mutations. Given that mutations can be seen as instances of faults (Andrews et al., 2005; Just et al., 2014b; Namin and Kakarla, 2011), we can formulate this attribute as: *a program is prone to generate equivalent mutants if it continues to deliver the same function despite the presence of faults*. This attribute is well known: *fault tolerance*. Equally well-known is what makes programs fault tolerant: *redundancy* (Pullum, 2001).

Hence if only we could quantify the redundancy of a program, we may be able to use it to assess a program's predisposition to generate a larger number of equivalent mutants. We quantify the proneness of a program to generate equivalent mutants by the *Ratio of Equivalent Mutants* (*REM*, for short), which is the ratio of the expected number of equivalent mutants that are generated from a base program, over the total number of generated mutants.

In this section, we consider a number of quantitative measures of redundancy, which are due to Mili et al. (2014); for each measure, we briefly present its definition, how we compute it, then why we feel that it may be correlated to the REM of the program. Because the REM is a fraction (included between 0.0 and 1.0), we resolve to define the redundancy metrics as fractions as well, so as to streamline the search for functional relationships. All these metrics are defined by means of Shannon's entropy (Shannon, 1948); we assume the reader familiar with the main (simpler) concepts and properties of this theory (Csiszar and Koerner, 2011). For the sake of simplicity, all entropies will be computed under the assumption of uniform probabilities.

2.1. State redundancy

How we define it. It is very common for programmers to declare more state space than they really need — in fact it is very uncommon not to. When we declare an integer variable to store the day of the month, then we are using 32 bits (typical size of an integer variable) to represent the range of values 1..31, for which 5 bits are sufficient. Also, if we declare three variables, to represent, respectively, the birth year of a person, the age of the person, and the current year, then one of the three variables is redundant (not to mention that each variable can represent many more values than we are using it for). We want state redundancy to reflect the gap between the declared state and the actual (used) state. We let S be the random variable that represents the declared state of the program, and σ be the random variable that represents its actual state; we let the *state redundancy* of the program be the ratio between the conditional entropy of S given σ over the entropy of S. Since the entropy of σ decreases as execution of the program proceeds, we resolve to define two versions of state redundancy, one for the initial actual state (σ_I) and one for the final actual state (σ_F); because σ is a function of S, the conditional entropy $H(S|\sigma)$ can be written as $H(S) - H(\sigma)$ (Csiszar and Koerner, 2011).

$$SR_I = \frac{H(S) - H(\sigma_I)}{H(S)}.$$

$$SR_F = \frac{H(S) - H(\sigma_F)}{H(S)}.$$

How we compute it. We consider a simple illustrative example:

```
int gcd (int a, b)
   {assert(a>0 && b>0);
    while (a!=b)
       if (a>b) {a=a-b;}
       else {b=b-a;}
    return a;}
```

The state space S of this program is defined by two integer variables, whose entropy is 2×32 bits=64 bits. The initial actual state space σ_I is defined by two positive integer variables, whose entropy is 2×31 bits=62 bits. The final actual state space σ_F is defined by one positive integer variable (since $a = b$ at the end), whose entropy is 31 bits. Hence we find:

$$SR_I = \frac{64 - 62}{64} = 0.03125.$$

$$SR_F = \frac{64 - 31}{64} = 0.51562.$$

Of course, the state redundancy of the final state is greater than that of the initial state, because execution of the program creates dependencies between program variables, which did not exist initially.

Why do we think it is correlated with the REM. State redundancy reflects the volume of bits of information that are part of the declared state but not part of the actual state; the more such bits are lying around the declared state of the program, the more likely it is that a mutation of P alters those bits but does not alter the actual state.

2.2. Functional redundancy

How we define it. We model the software product as a function from an input space, say X, to an output space, say Y, and we model functional redundancy by the conditional entropy of X given Y; to normalize this metric, we divide it by the entropy of X. Since Y is a function of X, the conditional entropy of X given Y is merely the difference of entropies.

$$FR = \frac{H(X|Y)}{H(X)} = \frac{H(X) - H(Y)}{H(X)}.$$

Similar metrics have been introduced in the past by Clark and Hierons (2012), Clark et al. (2019), Morell and Voas (1993), Voas and Miller (1993) and Androutsopoulos et al. (2014) to reflect a program's tendency to mask errors.

How do we compute it. We consider the example introduced above, where X is defined by two positive integer variables and Y is defined by a positive integer variable. We find:

$$FR = \frac{62 - 31}{62} = 0.5.$$

Why do we think it is correlated to the REM. The smaller the range of values that the output takes, the harder it is for a mutant operator to affect a change in the program's output (as there are fewer outputs that are distinct from the original).

2.3. Non injectivity

How we define it. Whereas in the previous section we model the program as a function from an input space to an output space, for the purposes of this metric we model it as a function from initial states to final states. This function is injective if and only if distinct initial states are mapped onto distinct final states; to quantify the non-injectivity of a program, we define a metric that reflects to what extent the function that maps initial states onto final states is far from injective. One way to do so is to consider the conditional entropy of the initial actual state (σ_I) given the final actual state (σ_F). Given that the latter is a function of the former, this conditional entropy equals the difference between their respective entropies; for normalization, we divide this quantity by the entropy of the initial actual state.

$$NI = \frac{H(\sigma_I|\sigma_F)}{H(\sigma_I)} = \frac{H(\sigma_I) - H(\sigma_F)}{H(\sigma_I)}.$$

How we compute it. We have already discussed (albeit by means of illustrative examples) how to compute the entropy of the initial actual state and the final actual state. For an illustrative example, we consider an insertion sort algorithm for an integer array of size, say 100. The initial actual state of this program includes the initial array, as well as two index variables, say i and j (where i is used to scan the array and j is used to hold the index where $a[i]$ fits in the partially sorted array). Hence we find:

$$H(\sigma_I) = 100 \times 32 + 2 \times \log(100) = 3213.29 \; bits.$$

In the final actual state variable i is fixed ($i = 100$ if the array is scanned in increasing order), but j is unknown (hence its entropy is $log_2(100)$). To estimate the entropy of the final array, consider that when we sort an array of size N, we reduce its entropy by $\log(N!)$, since $N!$ different permutations map to a single sorted version. Using the approximation $\log(N!) = N \times \log(N)$, we find:

$$H(\sigma_F) = 100 \times (32 - \log(100)) + \log(100) = 2528.97 \; bits.$$

Whence:

$$NI = \frac{3213.29 - 2528.97}{3213.29} = 0.213.$$

Why do we think it is correlated to the REM. One of the main drivers of mutant equivalence is a program's ability to mask errors (caused by program mutations) by virtue of the non-injectivity of its state transformations. Factor *NI* quantifies this ability. Another way to interpret non-injectivity: the ratio of state information that a mutant can lose and still retrieve the correct final state (hence be equivalent to the base program).

2.4. Non-determinacy

How we define it. We readily acknowledge that this metric is controversial, hence we make provisions for our model to be used with this metric or without. Whereas the concept of equivalence between a program and a mutant is widely considered to be well-understood, we argue that there may be some ambiguity over what constitutes equivalence. We consider the following three programs:

```
swap1()              swap2()              swap3()
  {int x, y, z;       {int x, y, z;       {int x, y; {int z;
   z=x;                 z=y;                 x=x+y;
   x=y;                 y=x;                 y=x-y;
   y=z;}                x=z;}                x=x-y;}}
```

Whether these three programs are considered equivalent or not depends on whether we view variable `z` as part of the state (in the first two versions), or as an auxiliary variable (and the state is defined by variables x and y alone). Rather than make this a discussion about state spaces, we make it a discussion about the oracle that is used to test for equivalence; more specifically, we want to quantify the non-determinacy of the oracle that tests the equivalence relation between the final state of the program (σ_F^P) and the final state of the mutant (σ_F^M). We define the *non-determinacy* of an oracle $EQ(\sigma_F^P, \sigma_F^M)$ as the ratio of the conditional entropy of σ_F^P given σ_F^M over the entropy of σ_F^P:

$$ND = \frac{H(\sigma_F^P|\sigma_F^M)}{H(\sigma_F^P)}.$$

How we compute it. For illustration, we consider two possible oracles, and compute their non-determinacy. If we define the oracle of equivalence as:

$$EQ_1(\sigma_F^P, \sigma_F^M) \equiv (x_F^P = x_F^M) \wedge (y_F^P = y_F^M) \wedge (z_F^P = z_F^M),$$

then $H(\sigma_F^P|\sigma_F^M) = 0$, hence $ND_1 = 0$. Indeed, if $EQ_1(\sigma_F^P, \sigma_F^M)$ holds then $H(\sigma_F^P|\sigma_F^M) = 0$ since knowing σ_F^M leaves no uncertainty as to the value of σ_F^P. On the other hand, if we define the oracle of equivalence as:

$$EQ_2(\sigma_F^P, \sigma_F^M) \equiv (x_F^P = x_F^M) \wedge (y_F^P = y_F^M),$$

then $H(\sigma_F^P|\sigma_F^M) = 32 \; bits$, hence $ND_2 = \frac{32}{96} = 0.33$. In this case, knowing σ_F^M informs us about two variables of σ_F^P, x and y, but fails to inform us about z.

Why do we think it is correlated to the REM. Clearly, the looser the oracle of equivalence, the more mutants will be deemed equivalent to the base program. In fact *ND* can be interpreted as the ratio of state information by which a mutant can differ from the base program and still satisfy the oracle of equivalence. In the example above, the mutant can differ from the base program by one variable out of three and still pass the test of equivalence.

2.5. Redundancy vs. circumstances of equivalence

To conclude this section on modeling redundancy, and highlighting its relationship to mutant equivalence, we survey the circumstances that may lead a mutant to be equivalent to a base program, and show that each of these circumstances may be captured by one of our metrics. In this discussion we refer to the terminology of Laprie (1995, 2004) and Avizienis et al. (2004) pertaining to faults, errors, and failures. We envision five circumstances under which a mutant M may be equivalent to a base program P.

- *The mutation does not alter the program's state*. In other words, though the mutant is distinct from the original source text, it has no impact on the state of the program. For example, consider the case of two variables x and y which represent unique identifying keys of some database records and the mutation changes a condition `(x>y)` onto `(x>=y)`; since x and y are distinct the mutation has no impact on the program's state, i.e. it does not affect the program's execution. The fact that x and y are distinct is an attribute of the actual state of the program, not the declared state. Since state redundancy (SR_I) reflects the gap between the actual state and the declared state, we argue that it measures the likelihood of changing the declared state without affecting the actual state.
- *The mutation does alter the state, but it is not a fault*. Laprie et al. define a *fault* as the *adjudged or hypothesized cause of an error*; hence if a mutation alters the state but does not cause an error (i.e. the altered state is as correct as the original state) then it is not a fault. As an example, consider a program to compute the sum of a non-empty array $a[1..N]$ indexed by variable i into variable x. If the original program P initializes i to 0 and x to 0; and the mutant M initializes i to 1 and x to $a[1]$, then the mutation is not a fault because the state it generates is not an error (it satisfies the intended loop invariant $x = \sum_{k=1}^{i} a[k]$). Given that the state redundancy increases monotonically from the initial state to the final state, the state redundancy of any intermediate state increases with that of the final state, we argue that SR_F is an adequate measure for this situation.
- *The mutation is a fault, but the errors it causes are masked*. For example, the mutation changes the sign of a variable, but that variable is subsequently raised to an even power. We argue that non-injectivity (NI) reflects this exact property, since it measures to what extent different initial states are mapped to the same final state.
- *The mutation is a fault, the errors it causes are propagated (are not masked), but they do not cause failures*. This is prone to happen whenever the observable output of the program is a projection of the program's final state; hence the fact that

Table 1
Redundancy metrics as drivers of mutant equivalence.

Circumstance of mutant equivalence	Redundancy attribute	Metric
The Mutation does not alter the program's state.	Initial state redundancy	SR_I
The Mutation does alter the program's state, but it is not a fault (the altered state is correct).	Final state redundancy	SR_F
The mutation is a fault, but the errors it causes are masked.	Non Injectivity	*NI*
The mutation is a fault, the errors it causes are propagated, but they cause no failure.	Functional Redundancy	*FR*
The mutation is a fault, the errors it causes are propagated they do cause failures, but the failures fall within the tolerance of the oracle of equivalence.	Non Determinacy	*ND*

the final state of M is distinct from the final state of P does not preclude that the output of M is still the same as the output of P. As an extreme example, consider a program that computes the median of an array by sorting the array and returning the entry in the middle; all cells except the middle cell could be altered without affecting the output of M. We argue that *FR* reflects this exact property, as it focuses on input/output spaces of the program, rather than its internal state space.

- *The mutation is a fault, it causes errors, errors are propagated, they cause failure, but the failure falls in the same equivalence class as the correct output*. This arises whenever we define equivalence as identity of some output variables, but not all output variables. As an example, programs `swap1()` and `swap2()` in the previous section would be considered equivalent if we only check variables x and y. We argue that non-determinacy (*ND*) reflects this exact property.

Table 1 summarizes the discussions of this section. There is one aspect of mutation equivalence that redundancy metrics do not capture, because they are based on semantic (rather than syntactic) analysis: reachability, i.e. the possibility that a mutation operator is applied to a statement that (due to poor program design) is unreachable. Indeed, all our analysis revolves around the question of why would a function remain intact if some part of it is altered; hence our approach does not capture the case when a mutation is applied to dead code. The integration of this factor is beyond the scope of this paper, and is the subject of further research; one way to integrate it into our analysis is to include a parameter that quantifies the ubiquity of unreachable code, perhaps a software metric such as McCabe's *cyclomatic complexity* (McCabe, 1976).

3. Estimating redundancy metrics

In this section we discuss how we are automating the derivation of the redundancy metrics. Among the five redundancy metrics, four (SR_I, SR_F, *FR*, *NI*) pertain to the base program and one (*ND*) pertains to the oracle that we use to determine equivalence. We resolve to build the regression model using only the four program-specific metrics, then to factor the non-determinacy by means of the following formula:

$$REM = \rho(SR_I, SR_F, NI, FR) + ND \times (1 - \rho(SR_I, SR_F, NI, FR)),$$

where $\rho(SR_I, SR_F, NI, FR)$ is the regression model we derive by using SR_I, SR_F, *NI*, *FR* as independent variables and *REM* as the dependent variable, and setting *ND* to zero. Indeed, whereas SR_I, SR_F, *FR*, and *NI* characterize the program, *ND* is a characteristic of the oracle that is used to rule on equivalence; also, *ND* plays a totally different role from the other factors in determining *REM*. Consider that in the extreme case when $ND = 1$, which occurs when we choose the oracle $EQ(\sigma_F^P, \sigma_F^M) \equiv \textbf{true}$ then we obtain an *REM* of 1 regardless of all the other factors. Hence *REM* varies between $\rho(SR_I, SR_F, NI, FR)$ and 1 as *ND* varies between 0 and 1; see Fig. 1. If we were to integrate *ND* as an independent variable of the regression model, it would be a factor in the linear regression formula, and we would have no way to ensure that the *REM* takes value 1 when $ND = 1$.

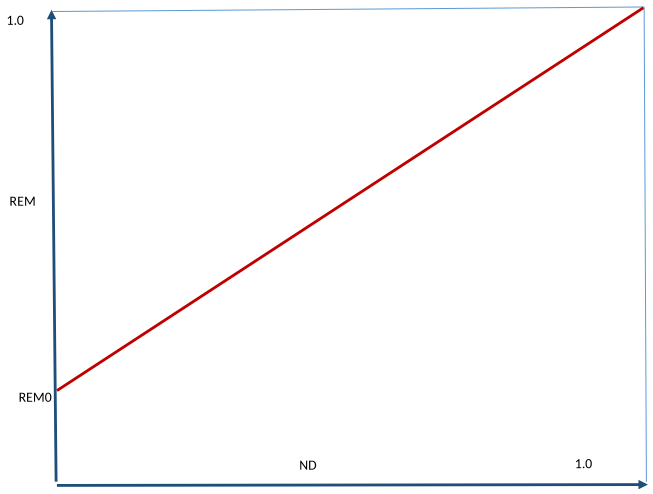


Fig. 1. The *REM*, a function of $REM0 = \rho(SR_I, SR_F, NI, FR)$ and *ND*.

It is possible to use this model by focusing exclusively on the program-specific redundancy metrics, which amounts to letting *ND* be zero. Then we get:

$$REM = \rho(SR_I, SR_F, NI, FR).$$

3.1. Program specific redundancy metrics

We deploy compiler generation technology (Parr, 1992) to automate the derivation of redundancy metrics for Java programs, which we use subsequently to derive an estimate of a program's REM, to analyze its mutation-related properties. Specifically, we apply syntax-directed translation to compute the program-specific redundancy metrics, namely SR_I, SR_F, *FR* and *NI*; we refer to this tool as *RC* (Redundancy Calculator). We have not yet automated the derivation of non-determinacy (*ND*), which is normally computed by analyzing the source code of the oracle of equivalence. In order to compute these metrics for Java methods, we need to evaluate the following entropies:

- $H(S)$: The entropy of the state space of the method.
- $H(\sigma_I)$: The entropy of the initial actual state of the method.
- $H(\sigma_F)$: The entropy of the final actual state of the method.
- $H(X)$: The entropy of the input space of the method.
- $H(Y)$: The entropy of the output space of the method.

We discuss below how these various quantities are evaluated.

3.2. Entropy of declared spaces

The semantic rules for $H(S)$, $H(X)$ and $H(Y)$ are fairly straightforward: each Java datatype is associated with a standard entropy under the assumption of equiprobability, which is just its number of bits. The declared spaces are S (the method's internal space), X (the method's input space, defined by explicit parameters or global variables accessed for reading), and Y (the method's output

space, defined by return statements or global variables accessed for writing). The entropy of each space is the sum of the entropies of its variables, and the entropy of each variable is dependent on the data type of the variable, as per the following table:

Data type	Entropy	Data type	Entropy
bool	1 bit	int	32 bits
byte	8 bits	float	32 bits
char	16 bits	long	64 bits
short	16 bits	double	64 bits

For the string data type, as well as for arrays whose size is not specified, we use default sizes (adjusted for the cell data type in the case of arrays), which can be overriden by the user.

3.3. Entropy of actual state spaces

The redundancy metrics require that we compute two entropies of actual state spaces: the entropy of the initial actual state and the entropy of the final actual state; we discuss these in turn, below.

3.3.1. Initial actual state

If we know nothing about the initial actual state of a method, then we assume that it can take any value within the range of the declared state space, hence $H(\sigma_I) = H(S)$. But if we know the precondition of the method, then we let the entropy of the initial actual state be defined as:

$$H(\sigma_I) = H(S) - \delta H(A),$$

where A is the assertion on S that defines the precondition of the method. In order to represent preconditions, we provide a special statement in (our slightly modified version of) Java, of the form `preassert(A)`, where A has the same syntax as in `assert(A)`, except that `preassert()` is used exclusively to record preconditions for the purpose of computing $H(\sigma_I)$. As for $\delta H(A)$ we define it recursively as follows:

- $\delta H(true) = 0$.
- $\delta H(false) = H(S)$.
- $\delta H(E1 == E2)$, where $E1$ and $E2$ are expressions, is the entropy of their (common) data type. Hence, for example, if the state space S of a method is defined by three integer variables, say x, y, z, and its precondition is written as `preassert(z==x+y)`, then we find:

  $$H(\sigma_I) = H(S) - \delta H(z = x + y) = 3 \times 32 - 32 = 64 \; bits.$$

- $\delta H(E1 > E2) = \delta H(E1 < E2) = \delta H(E1 \geq E2) = \delta H(E1 \leq E2) = 1$ bit, where $E1$ and $E2$ are two expressions of the same data type. Indeed, all these comparisons exclude half of the possible values of $E1$ and $E2$, hence reduce the entropy by 1 bit. As an illustration, if S is defined by a single variable x of type integer, and the precondition is `preassert(x>0)` then the entropy of the initial actual state is:

  $$H(\sigma_I) = H(S) - \delta H(x > 0) = 32 - 1 = 31 \; bits.$$

  Indeed, we know the sign bit of x, but we do not know its absolute value.
- $\delta H(E1 \neq E2) = 0$ bits. Indeed, the inequality excludes only one value for each expression ($E1$ and $E2$), and as an approximation we consider that this does not reduce the entropy. As an illustration, consider a state space S defined by an integer variable x, and a precondition written as `preassert(x!=0)`. Knowing that x can take any integer value except zero, we compute its entropy as: $H(\sigma_I) = \log(2^{32}-1)$; we are merely approximating this by $\log(2^{32}) = 32$ bits.
- $\delta H(A1 \wedge A2) = \delta H(A1) + \delta H(A2)$. Whereas this equation holds only if conditions $A1$ and $A2$ are logically independent, we adopt it for simplicity. As an illustration, consider a state space S defined by integer variables x, y, z and consider the precondition `preassert(x==y && x==z)`. Then, we find:

  $$\delta H(x = y \wedge x = z) = \delta H(x = y) + \delta H(x = z) = 64 \; bits.$$

  Whence, $H(\sigma_I) = 96 - 64 = 32$ bits, which reflects the fact that the entropy of the state is merely that of x.
- $\delta H(A1 \vee A2) = min(\delta H(A1), \delta H(A2))$. Indeed, if we do not know which of the two terms of the disjunction holds, we have to assume the term that reduces entropy the least; this yields the maximum entropy. As an illustration, consider a state S defined by three variables x, y, z, and consider the following precondition: `preassert(x==y || x==y && x==z)`. We have $\delta H(x = y) = 32 \; bits$, $\delta H(x = y \wedge x = z) = 64 \; bits$; hence $\delta H(x = y \vee x = y \wedge x = z) = 32 \; bits$, and

  $$H(\sigma_I) = 96 - 32 = 64 \; bits,$$

  which makes sense since all we know for sure is that $x = y$; we do not know whether $x = z$; and entropy measures uncertainty.

Note that despite our attempt to be systematic in analyzing preconditions, this method is not perfect: If we consider a space S defined by an integer variable x and the precondition `preassert(x>0&&x<17)` then our method finds that the entropy of this actual initial state is (32–2) bits, when it is in fact $log(16) = 4$ bits. To be able to analyze preconditions to this level of precision requires a depth of semantic analysis that is well beyond the scope of our tool.

3.3.2. Entropy of final actual state

While we can (barely) expect a user to provide us an explicit expression of the method's precondition, we cannot expect a user to provide a post condition; hence we resolve to compute the entropy of the final state by other means. Specifically, we resolve to compute the entropy of the final actual state by keeping track of the functional dependencies that the method creates between its variables as it executes; at the end of the program, we catalog all the variables whose initial value influences/determines the final state of the method; the entropy of the final actual state is the sum of the entropies of all these cataloged variables.

To this effect we introduce a square Boolean matrix (called D, for *dependencies*) which has as many rows and columns as the method has variables; at each location in the method's source code, this matrix contains **true** at $D[i, j]$ if and only if the value of variable i at the location depends on the initial value of variable j. Using matrix D, we derive vector V as the logical OR (disjunction) of all the rows of D; at each location in the method's source code, this vector contains **true** at $V[j]$ if and only if the state of the method at the selected location depends on the initial value of variable j. The entropy of the final actual state of the method is computed as the sum of the entropies of the variables whose corresponding entry in vector V_F is **true**, where vector V_F is the vector derived from the dependencies matrix D_F at the end of the method.

To compute the dependency matrix of a method, we proceed as follows:

- *Declaration*. Upon encountering the declarations of the variables that form the state space of the method, we create matrix D and initialize it to the identity Boolean matrix (**true** on the diagonal, **false** elsewhere, to signify that at the start, each variable depends exclusively on itself).
- *Initialization*. Whenever a variable is assigned a constant value, the row corresponding to that variable is assigned

Statement	Dependency Matrix
{int x, y, z;	D: x T F F; y F T F; z F F T
x=10;	D: x F F F; y F T F; z F F T
y=x+y;	D: x F F F; y F T F; z F F T
z=z+y;	D: x F F F; y F T F; z F T T
z=x+y;	D: x F F F; y F T F; z F T F
}	V: x y z; F T F

Entropy of Final Actual State: $H(\sigma_F) = H(y) = 32$ bits.

Fig. 2. Illustration of dependency matrix: successive assignments.

false everywhere, to signify that the value of this variable does not depend on any variable.

- *Assignment*. Whenever we encounter an assignment statement of the form {x=E}, where x is a variable and E is an expression, the row that corresponds to variable x in matrix D is replaced by the logical OR of the rows of all the variables that appear in expression E. If the assignment statement appears in an if-then statement, an if-then-else statement, or a while-do statement, then the variables that appear in the corresponding conditions (of the if-statement or the while-statement) are added to the list of variables in E.
- *If-Then-Else Statement*. Whenever we encounter a statement of the form

  `if (cond) {then − branch; } else {else − branch; }`

  we derive the dependency matrix of its then-branch, say $D1$, and the dependency matrix of its else-branch, say $D2$, then we select the matrix which yields the maximum entropy, and let that be the dependency matrix of the whole statement.
- *If-Then Statement*. We interpret the statement

  `if (cond) {then − branch; }`

  as a shorthand for

  `if (cond) {then − branch; } else {skip; }`

  and we apply the rule for If-Then-Else statements. Given that the `skip` leaves all variables intact, its entropy will be greater than that of the then-branch. The dependency matrix that corresponds to the `skip` would merely add the rows of all the variables that appear in the condition (`(cond)`) to all the rows of the matrix (since the decision to preserve all the program variables is dependent on these variables).
- *While Loop*. Whenever we encounter a statement of the form

  `while (cond) {loop − body; }`

  we derive the dependency matrix of its loop-body, and we let that be the dependency matrix of the whole statement (in other words, the dependency matrix of the loop-body jumps over the closing bracket). Remember that all assignments executed in the loop body involve (implicitly) the variables that appear in the loop condition.

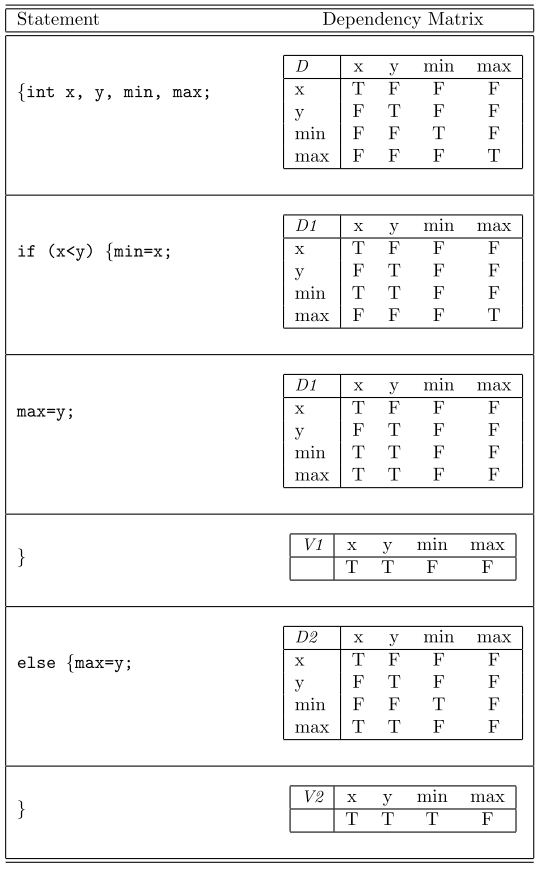
Figs. 2–4 show illustrative examples of, respectively, a sequence of assignment statements, an if-then-else statement, and an if-statement.

Note that despite our efforts to be systematic in analyzing functional dependencies between program variables, this method is not perfect: the stepwise analysis that we conduct as we scan the program starts from the initial declared state rather than the initial actual state; in other words, if we had a `preassert()` statement that cuts the entropy of the initial actual state in half, we would not know what to do with it, how to integrate it into our dependency analysis. In theory, we ought to start this process by considering the initial actual state defined by the precondition, and propagating it to the final state alongside functional dependencies. But doing so requires a depth of semantic analysis akin to symbolic execution, which is out of scope for our modest goal of estimating entropies.

4. Estimating the ratio of equivalent mutants

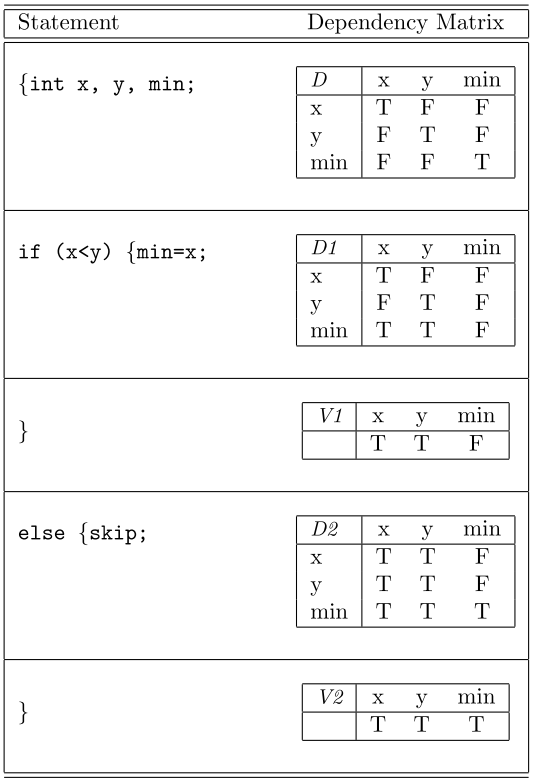
In order to test the validity of our conjecture that redundancy metrics enable us to predict the REM of a program, we conduct an experiment:

- We consider a set of functions taken from the *Apache Commons Mathematics Library* and the *Apache Commons Lang3 Library* (http://apache.org/), a benchmark of software components commonly used in software testing experiments.
- Each function comes with a test data file, which includes not only the test data proper, but also a test oracle that compares the output of the program under test with an expected value.
- To generate mutants, we use *PiTest*, (http://pitest.org/); in the context of this experiment, we deploy the default mutation operators of*PiTest*, which are: Increments Mutator; Void method Call Mutator; Return Vals Mutator; Math Mutator; Negate Conditionals Mutator; Invert Negs Mutator; Conditionals Boundary Mutator.
- To manage the executions, tests, and comparisons of the mutants, we use *Maven* (http://maven.apache.org/).



Entropy of Final Actual State: $H(\sigma_F) = max(H(x,y), H(x,y,min)) = 96$ bits.

Fig. 3. Illustration of dependency matrix: if-then-else statement.



Entropy of Final Actual State: $H(\sigma_F) = H(x,y,min) = 96$ bits.

Fig. 4. Illustration of dependency matrix: if-then statement.

- For each method, we run the redundancy metrics calculator, which generates estimates for the program-specific metrics (SR_I, SR_F, *FR*, *NI*).
- We estimate the REM of each function *P* as follows:
  - We execute *P* on all the data in the test suite and record its outputs.
  - We consider all the mutants that *PiTest* produces for *P*, and we execute every one of them on the test suite *T*.
  - We let REM be the ratio of mutants that produced the same output as *P* over the total number of generated mutants.

Our data sample includes 20 methods taken from three different classes of the benchmark (`FastMath`, `NumberUtils`, and `Fraction`). Their sizes range from 45 LOC to 219 LOC; and the size of the test suites that we use to estimate the actual ratio of equivalent mutants ranges between 49 and 1061. The total number of mutants that were generated for these methods ranges between 27 and 1009.

For lack of a more accurate method, we estimate the *REM* of a program *P* as the ratio of the number of surviving mutants over the total number of mutants in the mutation experiment; but a mutant may survive a mutation experiment either because it is equivalent or because the test suite is not sufficiently thorough to kill it. In order to minimize the likelihood of the latter scenario, we monitor the statement coverage of the mutants to ensure that they have been exercised thoroughly. We are mindful that perfect (100%) statement coverage is neither necessary (due to the possibility of unreachable code) nor sufficient (the same execution path of the mutant may yield the same outcome as *P* for some input data, and a different outcome from *P* for another). We are also mindful of the fact that the ratio of surviving mutants over the total number of mutants is an upper bound of the actual *REM*, and that we get more accurate estimates of the actual *REM* by enlarging and diversifying the test suite.

The outcome of this experiment is a data table that has five columns, which correspond to the four independent variables (SR_I, SR_F, *FR*, *NI*) and the dependent variable (*REM*). The raw data that we use for this model is available online (in pdf format) at http://web.njit.edu/~mili/model2.pdf. We use this data to fit a regression model for predicting REM from the redundancy metrics. We choose our final model to include the three redundancy metrics *SRI*, *SRF* and *FR*:

$$REM = -0.02715 - 1.2667SRI + 0.34349SRF + 0.08335FR.$$

The backward elimination procedure for selecting predictor variables found the three-variable model to have the highest adjusted R^2 and close to the lowest BIC value, suggesting it achieves a good balance between fit to the data and predictive performance. An all subset selection search had the same model as its best three-variable model. Since the data set is not large, we did not use cross-validation. The regression model is significant (ANOVA, p value = 0.03), and an examination of the residuals did not yield any concerns about model fit (see Fig. 5).

The R^2 value attained is 0.4, thus not a whole lot of the variation in REM is explained even by this model. This is most likely due to the small sample size. Nevertheless, this shows that the REM can be predicted fairly well with a very simple linear model. Performance can be improved with a larger sample size, allowing for consideration of non-linear models and better distinction between redundancy metrics; this is currently under

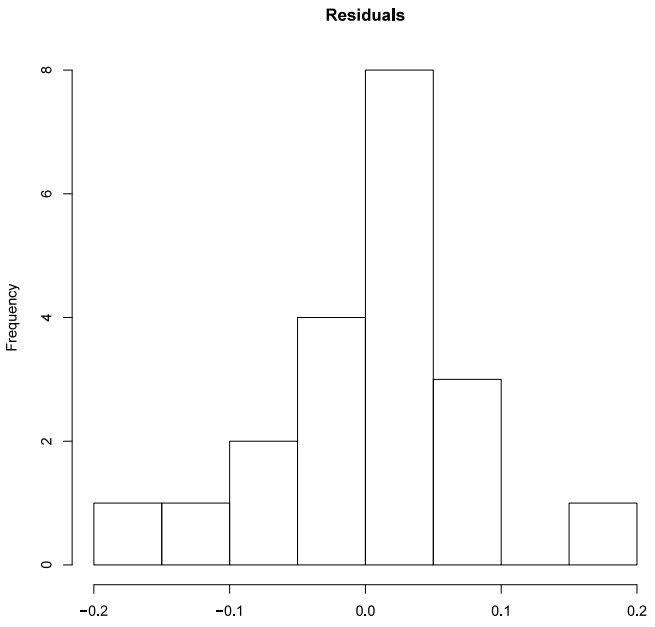


Fig. 5. Histogram of residuals: $REM = \rho(SR_I, SR_F, FR, NI)$.

investigation, alongside plans to include a factor that reflects the probability of mutant reachability.

Given a program P and a set of M mutants derived by the proposed mutation policy, we estimate the number of equivalent mutants as: $REM \times M$. Then the performance of a test suite T must be judged against the estimated number of non-equivalent mutants,

$$N = (1 - REM) \times M$$

rather than against M.

5. Quantifying mutation redundancy

We consider a program P whose ratio of equivalent mutants is REM and we consider a test suite T and let N be the number of mutants that T kills. We cannot tell how good set T is unless we know how many *distinct* mutants the set of killed mutants contains. What really measures the effectiveness of T is not N, but rather the number of equivalence classes of the set of killed mutants; the question that we must address then is, *how do we estimate the number of equivalence classes in a set of N mutants of P*?

To the extent that the mutants differ only slightly from the base program P (most common mutant generators apply minor syntactic differences) it is fair to assume that they have the same amount of redundancy as P, hence they also have the same REM. There are exceptions to this general assumption: if the mutation alters the flow of control of the program, and if the paths that are involved in this change have widely varying levels of redundancy, then clearly this assumption is not justified.

This situation notwithstanding, it is fair to assume that mutation operators do not generally alter the amount of redundancy in a program. Indeed, if we consider the set of programs made up of the original program P and its mutants, then all these programs are within slight syntactic differences from each other; the original program P is neither more likely nor less likely than any mutant to be equivalent to the other elements of the set. To test this hypothesis in practice, we consider the `fraction getFraction(double)` method of the benchmark class `Fraction` and apply *PiTest* to it with its default mutation operators, yielding

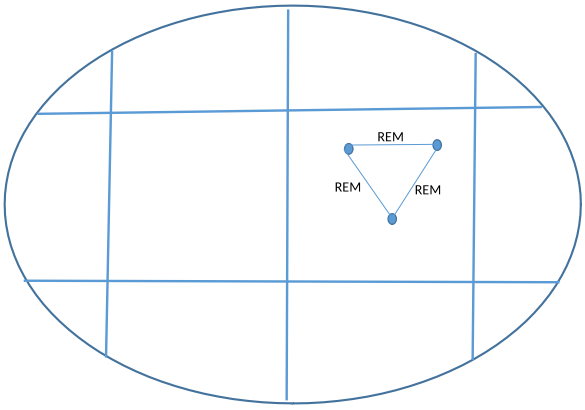


Fig. 6. Number of equivalence classes: $NEC(N, REM)$.

396 mutants; when we apply the RC tool to 40 randomly selected mutants among these (1 out of ten), we find that it returns the exact same value for each of the four program-specific metrics (SR_I, SR_F, FR, NI). To the extent that the REM of a program is determined by its redundancy metrics, programs that have the same redundancy values may be considered to have the same REM.

Hence we argue that REM can be used as an approximation of the probability that any two mutants are equivalent to each other; see Fig. 6. Then the question that we must address is: *Given a set of N elements, where any two have a probability REM of being equivalent, what is the expected number of equivalence classes*? We denote this number by $NEC(N, REM)$, and we write it as:

$$NEC(N, REM) = \sum_{k=1}^{N} k \times p(N, REM, k),$$

where $p(N, REM, k)$ is the probability that the number of equivalence classes is k. This probability can be estimated by the following inductive formulas:

- $p(N, REM, 1) = REM^{N-1}$.
  This is the probability that all N elements are equivalent.
- $p(N, REM, N) = (1 - REM)^{\frac{N\times(N-1)}{2}}$.
  This is the probability that no two elements are equivalent.
- $p(N, REM, k) = (1 - (1 - REM)^k) \times p(N - 1, REM, k) + (1 - REM)^{k-1} \times p(N - 1, REM, k - 1)$.
  This inductive formula considers two cases when we add an element to a set of size $(N - 1)$: the case when the new element falls in one of the existing equivalence classes; and the case when the new element defines its own equivalence class.

Now that we have an explicit formula for the number of equivalence classes in a set of N mutants, we argue that it is this number $NEC(N, REM)$, rather than N, that truly measures the effectiveness of T. With this in mind, it is worthwhile to consider two important properties of $NEC(N, REM)$:

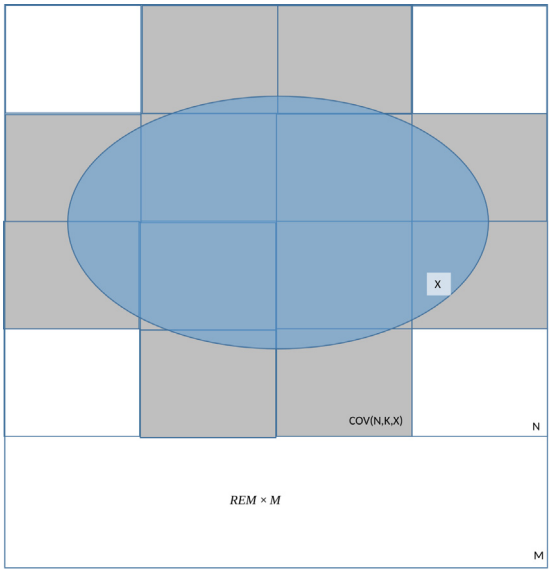
- This function depends a great deal more on REM than it depends on N; hence we cannot get a sense for the value of $NEC(N, REM)$ until we have determined REM.
- For typical values of REM (in the neighborhood of 0.05 to 0.25 for the sample programs we have encountered in common benchmarks (Just et al., 2014a; Benchmark, 2007) and in other empirical studies (Papadakis et al., 2015)), the number of distinct mutants is much smaller than the total number of generated mutants; in particular, we could kill

Table 2
Variation of $NEC(N, REM)$ as a function of N, REM.

N \ REM	0.05	0.10	0.15
30	18	13	11
300	54	33	24
3000	98	55	38

Table 3
Evolution of $p(N, REM, k)$ for $REM = 0.05$.

k	$p(3000, 0.05, k)$	k	$p(3000, 0.05, k)$
13	0	99	0.126499
14	4.94066e−324	110	0.000200261
87	8.35248e−05	111	6.43533e−05
88	0.000296773	240	8.67925e−320
97	0.114522	241	0
98	**0.126799**	3000	0



$MS = \frac{X}{M}, EMS = \frac{COV(N,K,X)}{K}$, where $N = M \times (1 - REM), K = NEC(N, REM). EMS = \frac{12}{16} = 0.75$

Fig. 7. Essential mutation score: $EMS(N, X)$.

thousands of mutants, only to realize that actually we have killed only a few dozen *distinct* mutants.

As a consequence of these observations, it is clear that when a test set T kills a large number N of mutants of some program P, we should not rush into premature celebration, until we have computed $NEC(N, REM)$; this, in turn, requires that we determine the REM of the program P. For the sake of illustration, we show in Table 2 the value of $NEC(N, REM)$ for some sample values of N and REM.

The results shown in this table are surprising, though they are borne out by empirical observations of other researchers: even with an REM as low as 0.05, the number of distinct mutants in a set of 3000 mutants is only 98. For an REM of 0.15, this number is even smaller at 38, nearly 1 percent of the total number of mutants. To validate and come to terms with this result, we consider a set of N elements divided into k equivalence classes, and we consider what happens if we add an element to this set. In order for the $(N + 1)^{st}$ element to increase the number of equivalence classes (to $k + 1$), it must be non-equivalent to all the existing equivalence classes. The probability of this event is $(1 - REM)^k$. Unless REM is very near zero, this quantity

Table 4
Evolution of $p(N, REM, k)$ for $REM = 0.10$.

k	$p(3000, 0.10, k)$	k	$p(3000, 0.10, k)$
6	0	56	0.158642
7	4.94066e−324	63	0.000289045
47	0.00014182	64	5.79254e−05
54	0.169991	157	8.04833e−321
55	**0.182252**	158	0

Table 5
Evolution of $p(N, REM, k)$ for $REM = 0.15$.

k	$p(3000, 0.15, k)$	k	$p(3000, 0.15, k)$
4	0	39	0.207674
5	4.94066e−324	45	0.000310997
31	1.02128e−05	46	4.20583e−05
32	0.000164307	122	1.10127e−320
37	0.173966	123	0
38	**0.22365**	3000	0

eventually becomes zero for sufficiently large k. This is borne out by observation of $p(N, REM, k)$ in the examples shown above for $N = 3000$; Tables 3, 4, 5 show the evolution of $p(3000, REM, k)$ for $k = 1..3000$, and $REM = 0.05, 0.10, 0.15$. For the sake of saving space, we only show a few lines, that correspond to the largest value of k for which $p(N, REM, k)$ is (rounded off to) zero, then the first value of k for which $p(N, REM, k)$ is non-zero, then some intermediate values, then the maximal value of $p(N, REM, k)$, then some intermediate values, then the last non-zero value of $p(N, REM, k)$, then the first value of k for which $p(N, REM, k)$ is (rounded off to) zero; it stays at zero until $k = N$. Interestingly (and not surprisingly) the value of k for which $p(N, REM, k)$ is maximal turns out to be $NEC(N, REM)$ for all three values of REM.

6. Mutation score

When we run M mutants of program P on some test suite T and we find that X mutants are killed, it is customary to view the ratio $MS(M, X) = \frac{X}{M}$ as a measure of effectiveness of T, called the *mutation score* of T. We argue that this formula suffers from two major flaws:

- The denominator ought to reflect the fact that some of the M mutants are equivalent to P, hence no test data can kill them.
- Both the numerator and the denominator ought to be quantified not in terms of the number of mutants, but instead in terms of the number of *distinct* mutants; a test suite cannot be credited for killing the same mutant repeatedly.

To address these flaws, we propose the following definition.

Definition 1. Given a program P and a set of M mutants thereof, of which N mutants are not equivalent to P, and given a test suite T. If execution of the M mutants on T causes X mutants to be killed, then the *essential mutation score* of T is denoted by $EMS(N, X)$ and defined as the ratio of distinct mutants in X over the total number of distinct mutants in N.

See Fig. 7: Whereas the traditional mutation score $MS(M, X)$ represents the ratio of X over M, the proposed mutation score $EMS(N, X)$ represents the ratio of equivalence classes that overlap with X (shaded in gray) over the total number of equivalence classes among mutants that are not equivalent to P (in this case $EMS(N, X) = \frac{12}{16} = 0.75$).

The denominator of EMS is already known, viz $NEC(N, REM)$. In Ayad et al. (2019c) Ayad et al. propose to compute the number

of equivalence classes that overlap with set X by introducing the following function: $COV(N, K, X)$, for a set of size N partitioned into K classes, and a subset thereof of size X, is the (expected) number of equivalence classes that overlap with set X (for the sake of simplicity, we may refer to a set and its cardinality by the same symbol); this function is called $COV()$, for *coverage*; see Fig. 7. Under some routine assumptions of statistical independence, this function is found to have the following expression:

$$COV(N, K, X) = K \times \left(1 - \left(\frac{K-1}{K}\right)^X \times \prod_{i=0}^{X-1} \frac{N - i \times \frac{K}{K-1}}{N-i}\right).$$

Letting EMS be the ratio of $COV(N, K, X)$ over $NEC(N, REM)$, where $K = NEC(N, REM)$ presents the following problem: whereas $COV(N, K, X)$ represents the number of equivalence classes of set N that overlap with set X, in practice set X does not overlap with equivalence classes at random; the elements of an equivalence class of set N are either all killed in a test, or all spared — since they are, by definition, equivalent.

Hence, rather than use the equivalence classes of N as a reference and ponder how many of these are covered by X, we use the number of equivalence classes defined in set X by the equivalence relation; their number is given by $NEC(X, REM)$. We find:

$$EMS = \frac{NEC(X, REM)}{NEC(N, REM)}.$$

We show an example of application of this formula in Section 9.1.

7. Minimal mutant sets

7.1. Extraction of minimal set

If we learn anything from Section 5, it is that the number of distinct mutants in a set of size N can be much smaller than N, even for very small values of REM. This raises the question: how can we identify a minimal set of distinct mutants that includes one representative from each equivalence class (to be as good as the whole set) and includes no more than one representative from each equivalence class (for the sake of minimality). In other words, given a set of size N partitioned by an equivalence relation, we want to select one and only one element from each class. If we do not know how many equivalence classes the set N has, then the algorithm for extracting a minimal set of mutants would have to scan all the elements of N, as shown below:

```
void min1(N)
{minset = emptyset;
forall (i in N)
  {bool equiv=false;
   forall (j in minset)
     {equiv = equiv||equivalent(i,j)}
   if (!equiv) {minset=minset+{i};}}}
```

Given that we know (albeit through an estimate, $K = NEC(N, REM)$) how many elements the minimal set has, it is not necessary to scan all the elements of set N; it suffices to iterate until we have identified K distinct elements. Whence the algorithm can be written as:

```
void min2(N, K)
{minset = emptyset;
 while ( |minset| < K)
  {bool equiv=false; i=nextelement(N);
   forall (j in minset)
     {equiv = equiv||equivalent(i,j)}
   if (!equiv) {minset=minset+{i};}}}
```

7.2. Number of inspections

This algorithm raises the question: what kind of a speedup do we achieve by stopping the iteration when we have found K distinct mutants? To answer this question, we introduce a new function, $NOI(N, K)$ (NOI: Number Of Inspections), which represents the estimated number of elements of a set of size N partitioned into K equivalence classes, that we need to inspect to get at least one element in each class. We let d_i, for $1 \leq i \leq K$, be the number of additional draws needed to cover the i^{th} equivalence class. Thus $d_1 \equiv 1$ since the first draw will necessarily cover a new class, while d_2 is the number of additional draws until a class other than the first class is drawn. We let D_K be defined as $D_K = \sum_{i=1}^{K} d_i$, the total number of draws in order to cover all K classes. Then our goal is to estimate the expected value of D_K. If N is very large relative to K, then the probability of covering a new equivalence class does not change with each draw, and each d_i is a geometric random variable with parameter $p_i = \frac{K-i+1}{K}$ and expected value $1/p_i$. The estimate of D_K can be written as

$$E(D_K) = 1 + \frac{K}{K-1} + \frac{K}{K-2} + \frac{K}{K-3} + \cdots + K.$$

If we allow probabilities to change with each draw, for example, if N is not large relative to K, then we resort to a recursive formula where the probabilities associated with d_i (to obtain its expected value) depends on the outcome of $d_1, \ldots, d_{i-1}$.

The recursive formula is obtained by considering a combinatorics problem, keeping track of the number of elements remaining of the already selected classes, which in turn affects the maximum number of additional draws to get a new class, and their corresponding probabilities.

As an example, we explain the process for d_2. Since $d_1 = 1$, there are now $N/K - 1$ elements of the first represented class, and $N - 1$ total elements. The possible values of d_2 are 1, if the next element is a new class, to N/K. The last case occurs if all elements of the represented class is selected before a new class is selected. We have

$$\begin{aligned}
P(d_2 = 1) &= \frac{N - N/K}{N-1} \\
P(d_2 = 2) &= \frac{N/K - 1}{N-1} \cdot \frac{N - N/K}{N-2} \\
&= \frac{N - N/K}{N-1} \cdot \frac{N/K - 1}{N-2} \\
&\vdots \quad \vdots \\
P(d_2 = N/K) &= \frac{N - N/K}{N-1} \cdot \frac{N/K - 1}{N-2} \\
&\quad \cdots \frac{N/K - (N/K - 1)}{N - N/K}
\end{aligned}$$

Each of the possible values of d_2 reflects a different state of the system, which in turn affects d_3. For example, if $d_2 = 1$, then there are $2N/K - 2$ elements of the two represented classes remaining, out of a total of $N - 2$ elements. This determines the possible values of d_3 and their probabilities.

We wrote computer code that recursively computes all the possible combinations for $d_2, \ldots, d_K$ and their probabilities. This allows us to compute their expected values, and hence $E(D_K)$, the expected number of draws needed to cover all K classes. With these probabilities, the standard deviation of D_K can also be computed.

Note that it is conceptually possible to consider the case with unequal numbers in the classes. However, in this case, the order in which each of the individual classes gets selected needs to

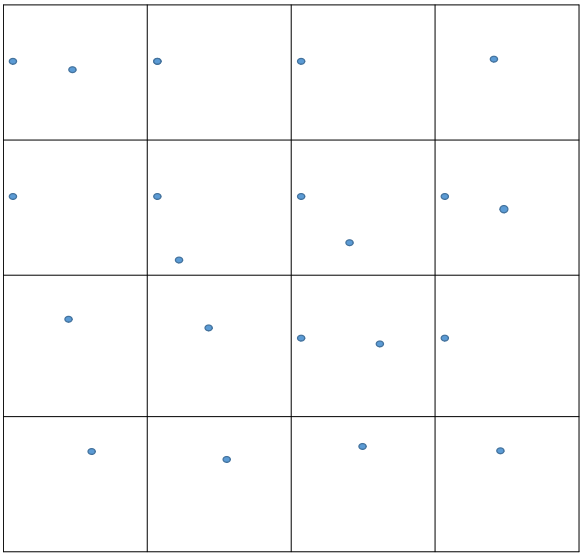


Fig. 8. Number of inspections: $NOI(N, K)$.

be taken into account, and the complexity of the combinatorics involved increases dramatically.

As a simple illustration, consider the example discussed in Section 5; for $N = 3000$ and $REM = 0.10$ we find $K = 55$. Whence we derive $NOI(3000, 55) = 252.649$; in other words, it is estimated that if we draw one mutant at a time in set of 3000 mutants partitioned into 55 equivalence classes, it takes about 253 draws before we cover all 55 classes (see Fig. 8).

7.3. Estimating the speedup

We briefly estimate the Big Oh performance of the two algorithms presented in Section 7, and evaluate the speed up achieved by adopting the second algorithm, as a function of $H = NOI(N, K)$. We consider the second algorithm, and note that the number of iterations of the inner loop equals the size of the minimal set, which ranges from 1 to K, over H iterations. So that the number of calls to function `equivalent(i,j)` is the sum:

$$1 + \cdots + 1 + 2 + \cdots + 2 + \cdots + (K-1) + \cdots + (K-1) + K.$$

This sum includes H terms ranging from 1 to K, where the number of '1's represents the number of iterations during which the minimal set had a single elements, the number of '2's represents the number of iterations during which the minimal set had two elements, etc. We approximate this sum with

$$\frac{H \times K}{2}.$$

Whereas the second algorithm ends when the minimal set reaches size K, the first algorithm continues until it has reviewed all N elements, checking all of them for equivalence against all K elements of the minimal set, hence its Big Oh performance can be estimated as:

$$\frac{H \times K}{2} + (N - H) \times K.$$

Hence the speed up is:

$$\frac{\frac{H \times K}{2} + (N - H) \times K}{\frac{H \times K}{2}}.$$

After simplification, we find:

$$\frac{2 \times N - H}{H}.$$

For $N = 3000$ and $H = 253$, we find a speed up of 22.72, or 2272%.

7.4. A safer approach

Given that $K = NEC(N, REM)$ is only an estimate, we cannot rely on it as if it were an exact value. For example, if $NEC(N, REM)$ is 12 but actually there are only 10 equivalence classes, then the second algorithm may scan all N elements looking for the remaining two equivalence classes, which do not exist; in such cases, we are not achieving any speed up. One possible remedy is to run the first algorithm, but applying H iterations of the outer loop rather than N; the danger in that case is that we may exit too early, before we have identified all the equivalence classes (since H is merely the expected mean of the number of inspections that are required). One possible remedy thereof is to use a boundary H' that is slightly larger than H (say, $H' = 1.5 \times H$); we are currently running experiments to see what multiplier of H (1.2, 1.5, 2.0) ensures that we cover all or most classes while minimizing overhead. We conducted a small-scale experiment in which we analyze the mutants of a base program to select a minimal set of distinct mutants, and we record how many equivalence tests we needed up to the last distinct mutant added; for example, if we have 40 mutants, and we find that it has only 10 distinct mutants, we record how many equivalence tests between pairs of mutants did we have to perform before we encountered the last (10th) distinct mutant; then we compare that number to H. Over several experiments involving several mutation experiments, we find that number to range between $0.65 \times H$ to $1.2 \times H$; hence if we let H' be $1.2 \times H$, we cover all equivalence classes; this bears further investigation.

8. Impact of mutant generation policy

So far, we have analyzed the REM of a program by focusing solely on the program, assuming a fixed mutant generation policy; but the REM also depends on the mutant generation policy, specifically, on the set of mutant operators that we deploy. We see two possible approaches to integrating the mutant generation policy with the analysis of the program's attributes:

- Either select some special mutant generation policies, such as those that are implemented in common tools (Coles, 2017; Ma and Offutt, 2020), or those that have some research interest (Andrews et al., 2005; Namin and Kakarla, 2011; Just et al., 2014b; Laurent et al., 2017; Offut et al., 1996); then develop a regression equation for each policy, giving the REM as a function of program attributes.
- Or select a set of individual mutation operators, develop a regression model for each operator, then infer the REM of a mutant generation policy from that of its individual operators.

The second option is more interesting, for two reasons:

- *Research Interest*. This approach gives us insights as to how individual mutation operators affect the REM of a program, and how they interact with each other (either by canceling each other or by accentuating each other's effect).
- *Practical Interest*. This option supports a much broader range of policies; if we select k operators, then we can support 2^k different sets of operators.

But this option depends on our ability to derive the REM of a set of operators from the REM's of the individual operators applied separately. We consider a set of operators, say op_1, op_2, ...op_k, and we let M_1, M_2, ...M_k be the number of mutants generated

Table 6
Mutation operators.

Op1	Increments_mutator;
Op2	Math_mutator;
Op3	Negate_conditionals_mutator;
Op4	Conditionals_ boundary_mutator;
Op5	Void_method_call_mutator;
Op6	Return_vals_mutator;
Op7	Invert_negs_mutator
Op8	Constructor_Calls_Mutator

Table 7
Sample functions.

#	Sample functions	Size (loc)
1	`long gcd(long p, long q)`	56
2	`long mulAndCheck(long a, long b)`	42
3	`double erfInv(final double x)`	88
4	`public ArrayRealVector ebeDivide(RealVector v)`	20
5	`public double getDistance(RealVector v)`	19
6	`ArrayRealVector(Double[] d, int pos, int size)`	12
7	`toBlocksLayout`	42
8	`public BlockRealMatrix getRowMatrix(final int row)`	27
9	`double abs()`	20
10	`void setSeed(final int[] seed)`	17
11	`ASINH`	17
12	`double atan(double xa,double xb,boolean leftPlane)`	143
13	`int gcdPositive(int a, int b)`	7
14	`mulAndCheck(int,int)`	42

from a base program P by the individual operators. The number of mutants that are equivalent to P for the mutant generation policy that deploys all these operators is:

$$REM_1 \times M_1 + REM_2 \times M_2 + \cdots + REM_k \times M_k,$$

where REM_i, $1 \leq i \leq k$ is the REM of operator Op_i. Hence the REM of P for the selected policy is:

$$REM = \frac{REM_1 \times M_1 + REM_2 \times M_2 + \cdots + REM_k \times M_k}{M_1 + M_2 + \cdots + M_k}.$$

In the absence of any information about the relative size of the M_i's, we assume that they are equal, which yields the following formula for REM:

$$REM = \frac{1}{k} \Sigma_{i=1}^{k} REM_i.$$

To validate this formula, we have run experiments with small values of k, viz. $k = 2, 3, 4$. Specifically, we have considered a set of 14 functions from the *Apache Common Mathematics Library* (http://apache.org/); each of these functions comes with a test suite and oracles in the form of assert statements; these functions are listed in Table 7. We use *PiTest* (http://pitest.org/) alongside *Maven* (http://maven.apache.org/) to generate mutants and test them for equivalence to the base program; from this data, we derive approximations of the REM of the base program under the active mutation operators. We run these experiments while activating individual mutation operators, then activating combinations of operators; and we compare the observed REM of the combinations of operators against the formula derived from the REM's of the individual operators. We run this experiment for two operators, considering all the pairs of operators of *Pitest*'s default operators; these operators are listed in Table 6. For each pair of operators, we compute the average and standard deviation of the relative error between the observed REM with the combined operators and the computed REM derived from the individual operators. Then we run the same experiments with selections of three operators, and some with four operators.

The results are surprisingly precise; when we consider the seven first operators of Table 6, extract the 21 pairs that can be drawn therefrom, and compare the observed REM vs. the REM computed as $\frac{REM1+REM2}{2}$, we find that 4 combinations have a predicted error (i.e. difference between observed and computed *REM*) of less than 0.01 and 12 have a predicted error less than 0.1; out of the remaining 5 combinations, four involve Operator 4, which is *Conditionals Boundary Mutator*. We consider four combinations of four operators by combining the set {Op1,Op2,Op3} with, respectively, Op4, Op5, Op6, and Op7; all four predicted errors (computed REM vs Observed REM) are less than 0.1. We also consider eleven combinations of three operators, of which we present ten, in Tables 8 and 9. Each entry in these tables represents the comparison of the observed REM (in blue) vs. the computed REM (in red) for the fourteen programs in our sample. Table 8 shows the results obtained for combinations of three mutation operators, and Table 9 shows the results obtained for combinations of up to seven operators. While the curves do not coincide for every program, they vary in unison, and the same programs seem to exhibit the largest gaps between the observed and the estimated REM's. This all raises intriguing questions that ought to be further investigated; a preliminary conclusion may be that it is fairly likely that we can estimate the REM of a program for a given mutant generation policy from its REM under each of the mutation operators that make up the policy. The data used for this experiment is available online at http://web.njit.edu/~mili/impact.pdf.

9. Assessment

In this section we present an assessment of our approach by showing a use-case of how we envision to compute and use the REM of a program (Section 9.1) then we discuss threats to the validity of our REM-based approach (Section 9.2).

9.1. A use case scenario

In this section, we briefly present a use-case of our quantitative approach, to reflect how we envision our approach to be used in practice. Many of the assumptions, equations, and models that we use for this purpose are tentative, in the sense that they are based on analytical arguments, but are not yet empirically validated.

We consider the method `fraction getfraction(double)` of the `Fraction` class, a 49 LOC method that converts a double into a fraction of integers. We deploy our RM calculator on this code, which yields the following redundancy metrics:

$$SR_I = 0.04199, SR_F = 0.9375, FR = 0.015.$$

Applying the regression formula

$$REM = -0.0271 - 1.2669 \times SR_I + 0.3434 \times SR_F + 0.0833 \times FR$$

yields $REM = 0.2427$. Execution of *PiTest* on this method yields 396 mutants; we estimate that among these, $396 \times 0.2427 = 96$ mutants are equivalent to `getfraction(double)`; while this number may sound excessive, when we actually tested the 396 mutants for equivalence to the base program, we found 92 to be equivalent (for the remainder of this discussion, we use our REM estimate of 96). This leaves 300 mutants who we estimate to be distinct from `getfraction(double)`.

The next question that we must address is: out of the 300 mutants that are not equivalent to the base program, how many are distinct from each other (i.e. how many equivalence classes are they partitioned into)? According to the discussions of Section 5, this can be estimated by $NEC(300, 0.2427)$. We find: $NEC(300, 0.2427) = 16.2$; in other words, the 300 mutants that are distinct from the base program are divided into about 16 equivalence classes; according to this finding, sixteen of these

Table 8
REM of composite mutation policies, Part I.

mutants are distinct from each other and the remaining 284 are all equivalent to one of these sixteen.

We use this data to better assess the effectiveness of a test suite; let us consider a test suite T that kills 220 out of the 396 mutants that were generated (according to our calculations, it cannot kill more than 300 anyway). The traditional mutation score of T would be:

$$MS = \frac{220}{396} = 0.556.$$

As we argue in Section 6, this score is flawed for two reasons: first because it does not take into account the fact that 96 out of the 396 mutants cannot be killed, hence test suite T cannot be "penalized" (re: getting a lower score) for failing to kill them; second, because MS counts individual mutants, whether they are equivalent to each other or not (thereby unduly "rewarding" a test suite for killing repeatedly the same mutant). The formula we propose is:

$$EMS = \frac{NEC(X, REM)}{NEC(N, REM)}.$$

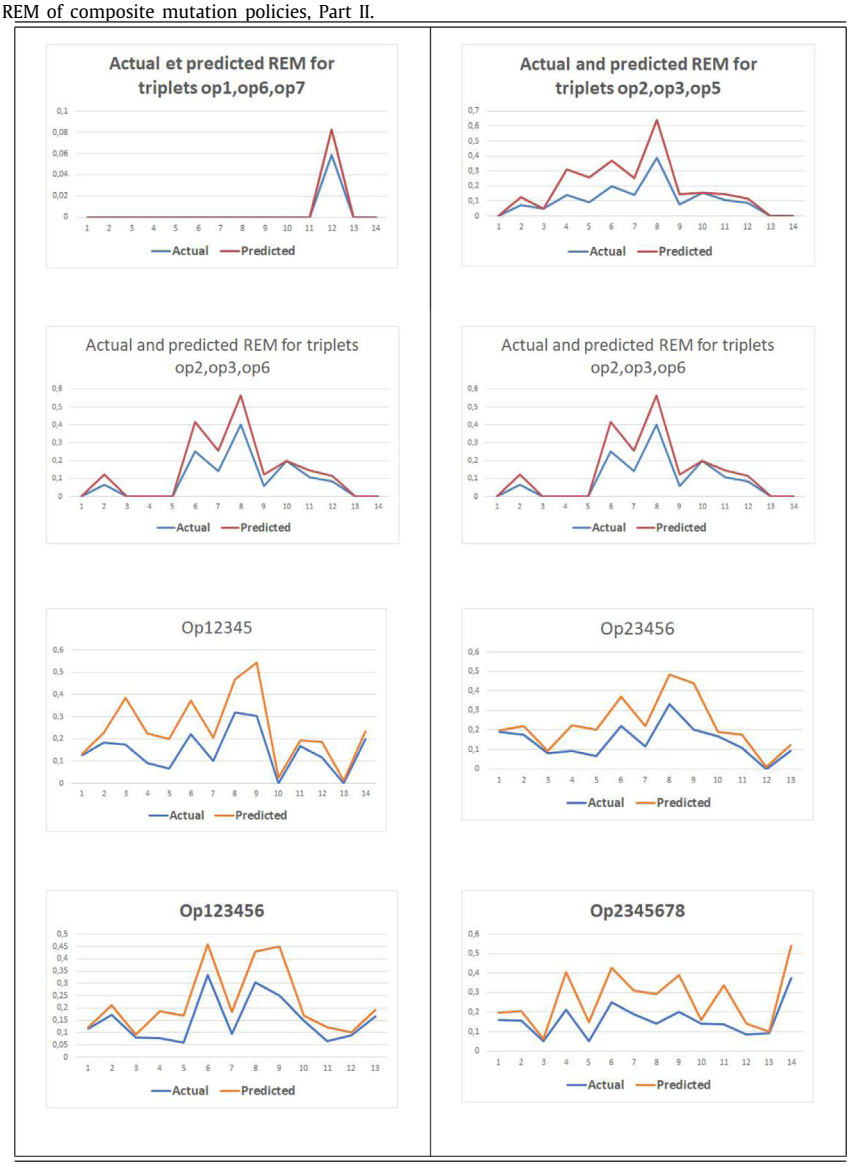
For $N = 300$, $REM = 0.2427$ and $X = 220$, we find:

$$EMS = \frac{15.1}{16.2} = 0.93.$$

Now, we have a set of 300 mutants, which we know are divided into 16 equivalence classes; this means that these 300 mutants are as good as 16, and 284 mutants are useless. We would like to select 16 distinct mutants in this set, and use them instead of all 300. If we did not know that we had 16 equivalence classes, then to select a minimal set of mutants we would have to consider each pair of mutants and test them for equivalence; this is clearly an expensive exercise. But now that we know the number of equivalence classes to be 16, we can start picking mutants one by one, testing each mutant against those that we have already picked, until we select a total of 16 distinct mutants. The question that arises then is: how many mutants do we need to pick, on average, to reach the number of 16. In Section 7 we introduce a function, called $H = NOI(N, K)$, which measures the expected number of mutants that we have to inspect in a set of N mutants divided into K equivalence classes before we pick at least one element from each class. We find:

$$H = NOI(300, 16) = 54.$$

Table 9
REM of composite mutation policies, Part II.



The number of mutants that we need to inspect before we find 16 distinct mutants is estimated to be 54. In Section 7 we find that by targeting the number of 16 equivalence classes, we achieve a speedup of

$$\frac{2 \times N - H}{H} = \frac{600 - 54}{54} = 10.11.$$

In other words, by resolving to identity distinct mutants until we reach the number of $NEC() = 16$, we achieve a speed-up of 10.11. But since $NEC = 16$ is only an estimate (derived from $REM = 02427$, which is itself an estimate), we cannot rely on it fully: if the actual number of equivalence classes is 15, then we would scan all 300 mutants looking for the missing equivalence class, which does not exist. A safer alternative is to simply inspect $H = 54$ mutants and return whatever number of distinct mutants we would have encountered by then, with the assurance that by then we normally would have covered all the equivalence classes; the risk with this approach is that if our estimate of the number of equivalence classes is lower than the actual number, we may terminate too early and miss equivalence classes. To remedy this possible flaw, we may resolve to inspect, not $H = 54$, but rather $H' = m \times H$, for some value of m greater than 1 (1.2, 1.5, 2.0) and let chips fall where they may; with greater values for m, we would increase the chance that we have covered all the equivalence classes without running the risk of having too many useless inspections. We choose the value of m according to the relative importance we assign to efficiency (shorter processing time) and effectiveness (ensuring that we have covered all the equivalence relations).

9.2. Threats to validity

The purpose of this paper is not so much to give definite ascertained answers as it is to raise questions and highlight the promise of a quantitative approach; we are highlighting a research direction which, in our view, offers the potential of useful insights into the mutation properties of programs, at a fraction of the cost and risk of current approaches. Whereas traditional approaches rely on semantic analysis to explore mutation equivalence, we rely on a quantitative, potentially automatable, static analysis of the program's source code. In this section we briefly discuss some weaknesses of our approach, from which we infer directions for future research.

9.2.1. Calculation of the redundancy metrics

The redundancy metrics used in this paper are defined on the basis of a model of computation where state variables are explicitly declared, have a well-known, fixed scope, have a fixed entropy which is known as compile time, and where variables are modified in the context of explicit assignment statements or function calls; but Java code is often more complex and messier than that. Also, because we aspire to compute redundancy metrics through static source code analysis, and because it is virtually impossible to infer detailed semantic information from static analysis, we have to make simplifying assumptions and approximations which are not always sound.

To remedy these weaknesses, we envision to revisit our definitions of redundancy metrics, possibly considering OOP-specific metrics (Chidamber and Kemerer, 1994); we are also considering to integrate metrics that capture a program's proneness to have unreachable code; one way to do so is to use metrics that measure the complexity of the program's control structures (McCabe, 1976). Also, we envision to revisit some of the simplifying assumptions we are making, to strike a more judicious compromise between effectiveness and precision (re: Section 3).

9.2.2. From redundancy metrics to REM

While the conjecture that the REM of a program is statistically correlated to its redundancy has been, we believe, borne out (Ayad et al., 2019a,b), the exact formula of *REM* as a function of the redundancy metrics is still under investigation. We have derived a number of regression models, which vary by mutant generation policy, program size, program type, regression model, assumptions, etc. We are still exploring an optimal classification of models, and the derivation of a minimal set of optimal validated models.

9.2.3. Integrating mutation policy

In Section 8 we argue that, if we know the REM of a program for individual mutant generation operators, say op_1, op_2, ... , op_k, then we can derive the REM of the program for the mutant generation policy that includes all these operators, using the formula:

$$REM = \frac{1}{k}\sum_{i=1}^{k} REM_i.$$

This formula is based on the assumption that all k operators generate approximately the same number of mutants. Empirical validation of this formula on two, three, and four operators, though anecdotal, bears out the proposed formula most of the time, but not always. To remedy this weakness, we envision to enhance the analysis of programs by cataloging the statements that are targeted by each operator, and quantifying the frequency of appearance of each statement in the program being analyzed. Then the formula of the REM would be:

$$REM = \frac{\sum_{i=1}^{k} f_i \times REM_i}{\sum_{i=1}^{k} f_i},$$

where f_i is the frequency of the statement(s) targeted by operator op_i in the base program being analyzed. We envision to conduct empirical experiments to test this formula.

9.2.4. Estimating vs. using the REM

This study raises a paradox, in the sense that at the same time that it highlights the importance of the REM in analyzing the mutation attributes of a program, it also highlights the challenges involved in estimating the REM with the required precision. This raises the question: does it make sense to explore/study/investigate all the uses of the REM if we do not have the means to estimate the REM with great precision? We answer this question with two premises:

- While we aspire to estimate the *REM* of a program P under a given mutant generation policy (say G) by a static analysis of P and G, it is possible to obtain an approximation of *REM* with a simple experiment. If we apply the mutant generation policy G to program P and test the mutants against P for equivalence using a test data T, then we obtain an upper bound for *REM*. The larger/more thorough/more diverse the test data T, the better the upper bound of *REM*.
- Second, we draw an analogy with COCOMO (Boehm, 1981; Boehm et al., 2000). The *COCOMO* and *COCOMO II* cost models provide a vast wealth of information about how to estimate the cost of a project in terms of person months and the length of the project in terms of months. They also detail how to distribute the cost and schedule across phases and activities, and how to manage software development teams according to the proposed estimates. All the cost and schedule equations of *COCOMO* and *COCOMO II* depend on our ability to estimate the size of the software product in KLOC's; but these equations hold a wealth of information on software development regardless of whether we can estimate product sizes.

9.2.5. Computing non determinacy

So far, we have endeavored to automate the calculation of the program-specific redundancy metrics (SR_I, SR_F, FR, NI), but not the redundancy metric that pertains to the oracle of equivalence. This remains a future research goal; it involves compiler generation technology, focused on analyzing Java functions that return a Boolean value.

10. Conclusion and prospects

Our goal in this paper is to draw researchers' attention to a venue of mutation testing research that has not been explored much so far, yet promises to yield useful results at low cost.

10.1. Summary

Research on mutation equivalence has so far focused primarily on the task of analyzing two programs (e.g. a program and a mutant, or two mutants) to determine whether they are semantically equivalent, despite being syntactically distinct. Notwithstanding that this task is known to be undecidable (Budd and Angluin, 1982), in practice it is also costly, tedious, and error-prone. This difficulty is compounded by the fact that in practice, we do not need to compare a base program with a single mutant, but with potentially a large number (say, M) of mutants, an $O(M)$ operation; Also, if we are interested in mutual redundancy between mutants, we need to test mutants against each other for equivalence, an $O(M^2)$ operation (see Table 10).

Fortunately, the need to test programs for semantic equivalence is not inevitable. We argue that for most intents and purposes, it is not necessary to individually identify those mutants that are equivalent to a base program; rather it is sufficient to estimate their number. This paper is based on two premises that stem from this claim, namely:

- Knowing the *Ratio of Equivalent Mutants* (REM) of a given program for a given mutant generation policy (defined by mutation operators) affords us significant insights into the mutation properties of the program and policy.
- It is possible to estimate the REM of a program under a mutant generation policy by analyzing the redundancy of the program.

We argue that many mutation-related questions can be solved economically if we estimate the ratio of equivalent mutants (*REM*) of a base program *P*. Indeed we find that once we have an estimate of the *REM* of a program *P*, we can:

- Estimate the number of mutants that are equivalent to *P*.
- Estimate the extent of mutual redundancy among the mutants that are not equivalent to *P*. In Section 5 we discuss how we use the REM of a program to assess the level of redundancy in a set of mutants. Specifically, we estimate the number of equivalence classes among those mutants that are not equivalent to the base program.
- Assess the effectiveness of a test suite in terms of the distinct mutants that it kills. In Section 6 we argue that the commonly used mutation score is flawed, and propose a new definition and a new formula for an improved mutation score, which assesses the performance of a test data not by the fraction of mutants that it kills, but by the fraction of *distinct* mutants that it kills. Empirical validation of this formula is under investigation; in particular, we are considering to check this formula against the criteria set forth in Gopinath et al. (2016).
- Extract a minimal set of mutants that is free of redundancy from a set of possibly redundant mutants. In Section 7 we argue that if a set of *N* mutants has only *K* equivalence classes, where *K* is much smaller than *N*, which is usually the case, then it is advantageous to limit the set of mutants to one element per equivalence class. Also, since we know ahead of time how many equivalence classes there are ($NEC(N, REM)$) then we can pick one mutant per equivalent class at little cost.

To derive the *REM* of a program, we ask the question: *What attribute of a program makes it prone to generate equivalent mutants?* We characterize programs that are prone to generate equivalent mutants as: *programs that preserve their function despite the presence of mutations*. Since *mutations* are introduced to simulate *faults*, we can replace the former by the latter in our characterization: *programs that preserve their function despite the presence of faults*. Such programs are known as *fault tolerant programs*, and the attribute that makes programs fault tolerant is well-known: it is *redundancy*. Hence if we can quantify the redundancy of a program, we can use it to estimate its *REM*. In earlier work, we had defined redundancy metrics of programs, and have used them as independent variables in empirically derived regression models whose dependent variable is *REM*. Also, we use compiler generation technology to produce a system that scans Java code to compute its redundancy metrics, from which it derives an estimate of *REM*. This system is currently being evolved to enhance its precision and scope.

10.2. Related work

The goal of the work of Zhang et al. (2019) titled *Predictive Mutation Testing* (PMT) overlaps to a large extent with our goal since both aim to estimate the outcome of a mutation experiment (in terms of how many mutants survive the test and how many are killed) without actually conducting the experiment; though we respectfully take exception to the claim of Zhang et al. (2019) that "*PMT is the first mutation testing approach that predicts mutant execution results (killed or alive) without mutant execution*"; in Marsit et al. (2017) Marsit et al. did just that, two years earlier. While the PMT approach and ours (to which we refer as the *REM* approach) have the same goal, they have distinct methods to achieve this goal: Whereas the PMT is purely empirical, and relies on machine learning to build its predictive model, the REM approach combines analytical and empirical methods; we attempt to identify what attributes of a program cause it to have a high *REM* value, we quantify these attributes, then we derive a regression model based on experimental data to compute the REM of a program as a function of its quantitative attributes. The identification of the quantitative attributes is based on a crucial argument: if mutations are surrogates for faults, then the attribute that makes a program prone to generate several equivalent mutants is the same as that which makes a program fault tolerant; that is *redundancy*. Whereas we use redundancy metrics as the independent variables on which the REM depends, Zhang et al. use the three factors of *Execution*, *Infection*, and *Propagation*. We readily concede that our redundancy metrics do not take into account the probability of *Execution*, and we envision to integrate this factor in our future research. On the other hand, we argue that our state redundancy metrics capture the probability of infection and our functional redundancy and non-injectivity metrics capture the probability of propagation.

Table 10
Functions and their meaning.

Function	Interpretation/Significance
$REM(SR_I, SR_F, FR, NI, ND)$	Ratio of equivalent mutants of a program, as a function of redundancy metrics.
$NEC(N, REM)$	Number of equivalence classes, set of size N where REM is the probability of equivalence of any two elements.
$COV(N, K, X)$	Number of equivalence classes covered by a set of size X in a set of size N partitioned into K classes.
$EMS(N, REM, X)$	Essential mutation score of a test data that has killed X mutants out of N for a program whose R. E. M. is REM.
$NOI(N, K)$	Number of draws in a set of size N partitioned into K classes to ensure that all classes are represented.

In Chekam et al. (2020) Titcheu Chekam et al. argue that the size of typical mutant pools is an obstacle to the widespread use of mutation testing in practice, and consider how to identify mutants that have the highest fault revealing capability so as to limit the set of useful mutants to those. They argue that the property of having a high fault revealing capability depends on static attributes of the mutants, and use a machine-learning algorithm to develop a tool, called *FaRM*, which analyzes a set of mutants and ranks them according to their fault revealing capability; they present empirical evidence to the effect that their tool outperforms random mutant selection and other more targeted tools. The work of Titcheu Chekam shares with our work the premise that mutation related attributes, though they are essentially dynamic attributes, can still be inferred from static analysis of relevant artifacts (the base program in our case, the mutants in the case of Chekam et al., 2020). Another interesting relationship may be that $NEC(REM, N)$ may offer an upper bound for the number of fault revealing mutants that one may want to seek; indeed, if the goal of *FaRM* is to seek the minimal number of mutants, then it should at the very least ensure that no two selected mutants are equivalent; hence any selected set must have no more than $NEC(REM, N)$, since any set of more than $NEC(REM, N)$ necessarily has equivalent mutants (re: The Pigeonhole Principle).

In Grano et al. (2019) Grano et al. argue that even though the mutation score of a test suite is a reliable indicator of test effectiveness, estimating the mutation score of a test suite is very costly, hence it is seldom used in practice. To remedy this flaw, they propose a way to estimate the test effectiveness of a test suite by a static analysis of the production code and the test code. To this effect they propose a set of 67 independent variables, which they divide into five broad classes: code coverage, test smells, production and test code metrics, code smells and readability. They use the mutation score as dependent variable, and

they deploy machine learning to derive an empirical model that estimates the mutation score of a production software product from the 67 independent variables pertaining to the production code and the test code. Like our approach, the approach of Grano et al. recognizes the inherent costs and inefficiencies of traditional mutation testing processes, and attempts to obviate them by focusing on static analysis of relevant artifacts; also, like our approach, this approach is based on the premise (which may or may not be valid) that the gain in efficiency exceeds and overrides the loss of precision. Of course, the mutation score is an imperfect measure of test effectiveness; a test suite may kill hundreds of mutants, yet whether this proves the effectiveness of the test suite depends on whether the mutants it has killed are distinct mutants or equivalent mutants; hence it is better to use a metric of test effectiveness that counts equivalence classes of mutants, rather than individual mutants.

An intriguing line of research aimed at minimizing the set of mutants without affecting its effectiveness relies on the concept of mutant subsumption, and defines mutant redundancy by the property of a mutant to be subsumed (hence deemed redundant) by another (Kurtz et al., 2014; Guimaraes et al., 2020). We find this line of research intriguing because while it aims to achieve the same goal as that of Section 7, namely to identify and weed out useless/redundant mutants, it uses a definition of redundancy that is totally different. According to Kurtz et al. a mutation m_1 subsumes a mutation m_2 if and only if any test that detects m_1 detects m_2. If we let M_i be the mutant obtained my applying mutation m_i, then the property of subsumption can be formulated as a property between mutants, rather than a property between mutations: M_1 subsumes M_2 if and only if:

$$\forall x : x \in \overline{dom(P \cap M_1)} \Rightarrow x \in \overline{dom(P \cap M_2)},$$

where P is the base program to which mutations are applied. By set theory, this can be rewritten as:

$$dom(P \cap M_2) \subseteq dom(P \cap M_1).$$

This is what Kurtz et al. call *true subsumption*; what they call *dynamic subsumption* with respect to some test data T can be defined by:

$$T \cap dom(P \cap M_2) \subseteq T \cap dom(P \cap M_1).$$

Interestingly, true subsumption has the same definition as the property of relative correctness proposed by Diallo et al. in Diallo et al. (2015): It is perfectly intuitive to consider that the best mutants are those that are least correct with respect to P; and the worst mutants are those that are correct with respect to (or equivalent to) P. As for dynamic subsumption, it is equivalent to relative correctness with respect to the restriction of P to T.

Whereas mutant equivalence is by definition an equivalence relation, mutant subsumption is an ordering relation (a pre-order, to be exact); whereas mutant equivalence minimizes the set of useful mutants by selecting one element per equivalence class, mutant subsumption minimizes the set of useful mutants by selecting the maximal elements of the ordering relation. An interesting question, to be pursued, is whether the REM of P can be used to estimate the size of a minimal set per the subsumption criterion the same way it can be used to estimate the size of a minimal set per the equivalence criterion.

10.3. Prospects: empirical validation

Many of the results we have presented in this paper stem from a mathematical analysis; we need empirical evidence to validate these results. Also, we are conducting empirical experiments to validate the analytical results discussed in this paper. These involve benchmark software that is traditionally used in testing experiments and common mutation generation systems, and include:

- Checking that our estimate of $NEC(N, REM)$ is borne out in practice.
- Checking the validity of our hypothesis that a given base program and its mutants have (approximately) the same *REM*. Even if the *REM* are not exactly identical, their effect on $NEC(N, REM)$ may be limited (e.g. if some have a higher *REM* than P, and other have a lower *REM*).
- Checking that our estimate of $NOI(N, K)$ is borne out in practice: Given a set of size N partitioned into K equivalence classes, how many inspections are needed, on average, before all the equivalence classes are visited. We test this hypothesis with and without the assumption that equivalence classes have the same size.
- Given a set of size N partitioned into K equivalence classes, how often are $NOI(N, K)$ inspections sufficient to cover all classes? How does the likelihood of covering all the classes vary with a multiplicative coefficient (1.1, 1.2, 1.5) of $NOI(N, K)$? We test this hypothesis with and without the assumption that equivalence classes have the same size.
- How realistic is our assumption (made occasionally throughout the paper) that equivalence classes of mutants have (approximately) the same size? Interestingly, this question may be at the heart of whether the mutants have the same *REM* (among themselves), and the same *REM* as the base program P.

These are clearly long term research goals; the aspiration of this paper is to show that these are worthwhile goals, given that they offer a cost-effective alternative, or complement, to prevailing approaches that are based on (undecidable/intractable) semantic analysis.

CRediT authorship contribution statement

Imen Marsit: The first to discover the correlation between redundancy metrics and REM, Responsible for much of the data collection, compilation and analysis, including the data in sections 3 and 8. **Amani Ayad:** Responsible for developing the system for calculating redundancy metrics and for deriving the regression model using the redundancy metrics as calculated by her tool. **David Kim:** An undergraduate directed study course on testing the hypothesis that all mutants have the same REM. **Monsour Latif:** Did a graduate directed study on estimating the number of equivalence tests that one must conduct before identifying all the distinct mutants in a pool. **JiMeng Loh:** Ensured that our statistical analysis is up to par. **Mohamed Nazih Omri:** Ensured the quality of Imen's data collection and analysis. **Ali Mili:** Introduced the redundancy metrics and coordinated the project.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partly supported by NSF, United States grant number DGE1565478. The authors are very grateful to the anonymous reviewers for their valuable/insightful feedback, which has greatly helped us to improve the paper and enrich its content.

References

Aadamopoulos, K., Harman, M., Hierons, R., 2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Proceedings, Genetic and Evolutionary Computation, pp. 1338–1349.

Abran, A., 2012. Software Metrics and Software Metrology. John Wiley and Sons.

Ammann, P., Delamaro, M., Offutt, J., 2014. Establishing theoretical minimal sets of mutants. In: Proceedings, ICST 2014.

Andrews, J., Briand, L., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments?. In: Proceedings, ICSE.

Androutsopoulos, K., Clark, D., Dan, H., Hierons, R.M., Harman, M., 2014. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proceedings, ICSE 2014.

Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1, 11–33.

Ayad, A., 2019. Quantitative Metrics for Mutation Testing Technical Report, (Ph.D. Dissertation). New Jersey Institute of Technology, Newark, NJ, USA.

Ayad, A., Marsit, I., Loh, J., Omri, M.N., Mili, A., 2019a. Estimating the number of equivalent mutants. In: Proceedings, MUTATION 2019, Xi'An, China.

Ayad, A., Marsit, I., Loh, J., Omri, M.N., Mili, A., 2019b. Quanatitative metrics for mutation testing. In: Proceedings, ICSOFT 2019. Prague, Czech Republic.

Ayad, A., Marsit, I., Loh, J., Omri, M.N., Mili, A., 2019c. Using semantic metrics to predict mutation equivalence. In: von Sinderen, M., Maciaszek, L.A. (Eds.), Communications and Computer Information Science. Springer Verlag.

Benchmark, 2007. Siemens Suite. Technical Report, Georgia Institute of Technology, URL: http://www-static.cc.gatech.edu/aristotle/Tools/subjects/.

Boehm, B., 1981. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ.

Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D., Steece, B., 2000. Software Cost Estimation with COCOMO II. Prentice Hall PTR, Upper Saddle River, NJ.

Boehme, M., 2018. Stads: Software testing as species discovery. ACM TOSEM 27, 1–52.

Budd, T.A., Angluin, D., 1982. Two notions of correctness and their relation to testing. Acta Inform. 18, 31–45.

Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F., 1980. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: Proceedings, POPL'80, 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 220–233.

Carvalho, L., Guimares, M., Fernandes, L., Hajjaji, M.A., Gheyi, R., Thuem, T., 2018. Equivalent mutants in configurable systems: An empirical study. In: Proceedings, VAMOS'18, Madrid, Spain.

Chao, A., Chiu, C.H., 2014. Species Richness: Estimation and Comparison. John Wiley and Sons.

Chekam, T.T., Papadakis, M., Bissiyande, T.F., LeTraon, Y., Sen, K., 2020. Selecting fault revealing mutants. In: Proceedings, FSE/ESEC 2020.

Chidamber, S.R., Kemerer, C.F., 1994. A suite for object oriented design. IEEE TSE 20.

Clark, D., Hierons, R.M., 2012. Squeeziness: An information theoretic measure for avoiding fault masking. Inform. Process. Lett. 112.

Clark, D., Hierons, R.M., Patel, K., 2019. Normalized squeeziness and failed error propagation. Inform. Process. Lett. 149.

Coles, H., 2017. Real World Mutation Testing. Technical Report, pitest.org.

Csiszar, I., Koerner, J., 2011. Information Theory: Coding Theorems for Discrete Memoryless Systems. Cambridge University Press.

Diallo, N., Ghardallou, W., Mili, A., 2015. Correctness and relative correctness. In: Proceedings, 37th International Conference on Software Engineering, NIER Track. Firenze, Italy, http://dx.doi.org/10.1109/ICSE.2015.200.

Fenton, N., 1991. Software Metrics: A Rigorous Approach. Chapman and Hall.

Fenton, N.E., Pfleeger, S.L., 1997. Software Metrics: A Rigorous and Practical Approach. PWS Publishing Company.

Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., Groce, A., 2016. Measuring effectiveness of mutant sets. In: Proceedints, Ninth International Conference on Software Testing, Chicago, IL..

Grano, G., Palomba, F., Gall, H.C., 2019. Lightweight assessment of test case effectiveness using source code quality indicators. IEEE Trans. Softw. Eng..

Gruen, B., Schuler, D., Zeller, A., 2009. The impact of equivalent mutants. In: Proceedings, MUTATION 2009. Denver, CO, USA.

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

Halstead, M., 1977. Elements of Software Science. North Holland, Amsterdam.

Hierons, R., Harman, M., Danicic, S., 1999. Using program slicing to assist in the detection of equivalent mutants. J. Softw. Test. Verif. Reliab. 9.

Just, R., Ernst, M.D., Fraser, G., 2013. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In: Proceedings, Dagstuhl Seminar 13021: Symbolic Methods in Testing.

Just, R., Jalali, D., Ernst, M.D., 2014a. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings, ISSTA 2014, San Jose, CA, USA. pp. 437–440.

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

Just, R., Kurtz, B., Ammann, P., 2017. Inferring mutant utility from program context. In: Proceedings, ISSTA 2017, Santa Barbara, CA.

Kintis, M., Papadakis, M., Jia, Y., Malveris, N., Le Traon, Y., Harman, M., 2018. Detecting trivial mutant equivalences via compiler optimizations. IEEE Trans. Softw. Eng. 44.

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

Kushigian, B., Rawat, A., Just, R., 2019. Medusa: Mutant equivalence detection using satisfiability analysis. In: Proceedings, ICST 2019, Xi'An, China.

Laprie, J.C., 1995. Dependability —its attributes, impairments and means. In: Predictably Dependable Computing Systems. Springer Verlag, pp. 1–19.

Laprie, J.C., 2004. Dependable computing: Concepts, challenges, directions. In: Proceedings, COMPSAC.

Laurent, T., Papadakis, M., Kintis, M., Henard, C., LeTraon, Y., Venturesque, A., 2017. Assessing and improving the mutation testing practice of pit. In: ICST 2017, Tokyo, Japan.

Ma, Y.S., Offutt, J., 2020. Mu Java. Technical Report, George Mason University, visited June.

Marsit, I., Omri, M.N., Loh, J., Mili, A., 2018. Impact of mutation operators on the ratio of equivalent mutants. In: Proceedings, SOMET, Granada, Spain.

Marsit, I., Omri, M.N., Mili, A., 2017. Estimating the survival rate of mutants. In: Cardoso, J.S. Maciaszek, L.A. van Sinderen, M. Cabello, E., (eds.), Proceedings, ICSOFT, Madrid, Spain.

McCabe, T., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 308–320.

Menendez, H., Boreale, M., Gorla, D., Clark, D., 2020. Output sampling for output diversity in automatic unit test generation. IEEE TSE..

Mili, A., Jaoua, A., Frias, M., Helali, R.G., 2014. Semantic metrics for software products. Innov. Syst. Softw. Eng. 10, 203–217.

Morell, L.J., Voas, J.M., 1993. A framework for defining semantic metrics. J. Syst. Softw. 20, 245–251.

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

Nica, S., Wotawa, F., 2012. Using constraints for equivalent mutant detection. In: Andres, C., Llana, L. (Eds.), Second Workshop on Formal Methods in the Development of Software. pp. 1–8, doi: 10.420/EPTCS.86.1.

Offut, J., Lee, A., Rothermel, G., Untch, R., Zapf, C., 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol 5, 99–118.

Offut, A.J., Pan, J., 1997. Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. 7, 165–192.

Papadakis, M., Delamaro, M., LeTraon, Y., 2014. Mitigating the effects of equivalent mutants with mutant clasification strategies. Sci. Comput. Program. 95, 298–319.

Papadakis, M., Jia, Y., Harman, M., LeTraon, Y., 2015. Trivial compiler equivalence: A large scale empirical study of simple, fast and effective mutant detection technique. In: Proceedings, International Conference on Software Engineering.

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

Parr, T., 1992. ANTLR: ANother Tool for Language Recognition, Technical Report, University of San Francisco. antlr.org.

Pullum, L.L., 2001. Software Fault Tolerance Techniques and Implementation. Artech House, Norwood, MA.

Schuler, D., Zeller, A., 2010. Covering and uncovering equivalent mutants. In: Proceedings, International Conference on Software Testing, Verification and Validation. pp. 45–54. http://dx.doi.org/10.1109/ICST.2010.30.

Shannon, C., 1948. A mathematical theory of communication. Bell Syst. Tech. J. 27, 379–423, 623–656.

Shin, D., Yoo, S., Bae, D.H., 2018. A theoretical and empirical study of diversity-aware mutation adequancy criterion. IEEE TSE 44.

Voas, J.M., Miller, K., 1993. Semantic metrics for software testability. J. Syst. Softw. 20, 207–216.

Wang, B., Xiong, Y., Shi, Y., Zhang, L., Hao, D., 2017. Faster mutation analysis via equivalence modulo states. In: Proceedings, ISSTA'17, Santa Barbara, CA, USA.

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

Zhang, J., Wang, Z., Zhang, L., Hao, D., Zhang, L., Cheng, S., Zhang, L., 2016. Predictive mutation testing. In: ISSTA 2016, Saarbrucken, Germany.

Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., Zhang, L., 2019. Predictive mutation testing. IEEE TSE 45, 898–918.

Dr. Imen Marsit is an assistant professor of computer science at the University of Sousse, Tunisia.

Dr. Amani Ayad is an assistant professor of computer science at SUNY Farmingdale in New York, USA.

David Kim is an Honors undergraduate student at NJIT, Newark, NJ USA.

Monsour Latif is a graduate student at NJIT, Newark, NJ USA.

Dr. JiMeng Loh is an associate professor of mathematics at NJIT, Newark NJ USA.

Dr. Mohamed Nazih Omri is a professor of computer science at the University of Sousse, Tunisia.

Dr. Ali Mili is a professor of computer science at NJIT, Newark NJ USA.