Estimating the Number of Equivalent Mutants

Amani Ayad
NJIT, Newark NJ
mili@njit.edu

Imen Marsit
MARS Lab, Sousse, Tunisia
imen.marsit@gmail.com

JiMeng Loh
NJIT, Newark NJ
loh@njit.edu

Mohamed Nazih Omri
MARS Lab, Sousse, Tunisia
mohamednazih.omri@fsm.rnu.tn

Ali Mili
NJIT, Newark NJ
mili@njit.edu

***Abstract*—Equivalent mutants are a constant source of aggravation in mutation testing because they distort mutation-based analysis; but the identification of equivalent mutants is known to be undecidable, in addition to being (in practice) tedious and error-prone. We argue that for most applications it is not necessary to individually identify equivalent mutants; rather it suffices to know/ estimate their number. In this paper, we discuss the specification and design of an automated tool that estimates the number of equivalent mutants generated from a base program by analyzing the source code of the program as well as the mutant generation policy.**

***Index Terms*—mutation testing, mutant generators, equivalent mutants, redundancy metrics, software tool.**

I. Introduction: Estimating the Number of Equivalent Mutants

A. Mutation Equivalence Research

Equivalent mutants are a major nuisance in mutation testing because they introduce a significant amount of bias in mutation-based analysis. Consequently, much research has been devoted to the identification of equivalent mutants in a pool of mutants generated from a base program by some mutation generation policy [1, 2, 7, 9–17, 22–25, 27, 29, 30]. In [26], Papadakis et al. present a sweeping survey of mutation testing, starting from its amergence in the late seventies to the present; they analyze the evolution of the level of interest in mutation testing, as reflected by the number of publications that appear in relevant venues. Papadakis et al. survey the various applications of mutation testing, the tools of mutant generation, the extensions of the mutation concept to other software artifacts, and the main technical challenges of mutation testing. Interestingly, they cite the problem of equivalent mutants among the research questions that remain largely unresolved.

We could not, in this paper, do a detailed survey of all this work; hence we content ourselves with some broad generic characterizations thereof. Like all generalizations, our characterizations may be unfair representations of individual works, but they provide a convenient abstraction for the sake of our discussions. At its core, the detection of equivalent mutants consists in analyzing a base program (say P) and a mutant (say M) to determine whether P and M are semantically equivalent, while they are (by construction) syntactically distinct. This is clearly a very difficult problem, since it relies on a detailed semantic analysis of P and M; if we knew how to perform a detailed semantic analysis of P alone (let alone P and M) we could probably determine whether P is correct, and do away with testing altogether. In the absence of a simple, general, practical solution, researchers have resorted to approximate solutions and heuristics. These include, for example, inferring equivalence from a local analysis of P and M in the neighborhood of the mutation(s); this produces sufficient but unnecessary conditions, hence leads to a loss of recall; definite equivalence requires, inconveniently, that we analyze the programs in full. Other approaches include overapproximations and comparison of the programs' functions (through slicing) or the programs' dynamic behavior (through execution traces); these approaches yield necessary but insufficient conditions of equivalence, hence lead to a loss of precision.

It is fair to claim that despite several decades of research, there is no general, simple, scalable solution for dealing with equivalent mutants in mutation testing.

B. Premises of Our Approach

Analyzing a program P and a mutant M to determine whether they are semantically distinct is easy, and can be very inexpensive: it suffices to find a test datum on which they produce different outputs/ outcomes; but no amount of testing can, in practice, enable us to infer that a mutant M is semantically equivalent to a base program P. Analyzing a program P and a mutant M to determine whether they are semantically equivalent is known to be undecidable [6]. Notwithstanding this theoretical limitation, practical efforts to identify equivalent mutants are often characterized by some combination of: prohibitive cost, limited automation, lack of precision, and lack of recall. These costs and penalties are compounded when we are talking, not about analyzing a single mutant to check for equivalence with a base program, but rather analyzing dozens, or hundreds of mutants (as large programs are prone to produce). Against this background, we propose the following premises as a basis for our approach:

- *For most applications of mutation testing, it is not necessary to identify individually those mutants that are equivalent to the base program; rather it suffices to estimate their number.* If we generate 100 mutants and we know that 20 of them are equivalent to P, then we can assess the mutation score of any test data set T by considering what percentage of the remaining 80 mutants it kills.
- *Even when it is important to single out those mutants that are equivalent to the base program, knowing their number ahead of time can be very helpful.* If we generate

100 mutants and we know that 20 of them are equivalent to P, then we know that we have to kill 80 mutants before we are assured (modulo the precision of the estimate) that the surviving mutants are equivalent to P.

- *It is possible to estimate the number of equivalent mutants generated from a base program by analyzing the source code of the program as well as the mutant generation policy*. More specifically, we aim to estimate the proportion of generated mutants that are equivalent to the base program; we refer to this quantity as the *Ratio of Equivalent Mutants*, or *REM* for short. In this paper we discuss how we can estimate the REM of a program under a given mutant generation policy, then we discuss the specification and design of an automated tool that does so.

In Section II we discuss what attributes of a base program are good indicators (predictors) of the REM of a program, and how we can quantify them. In Section III we present an empirical study in which we highlight evidence to the effect that the REM of a program can be estimated on the basis of an analysis of the program's source code, and how we envision to build on this evidence to develop an automated tool that estimates the REM of a program. In Section IV we present the requirements specification of the tool we envision for carrying out this function, and in Section V we discuss the ongoing design and implementation of this automated tool. In Section VI we revisit the model of Section II by showing how we can adapt our estimate to different mutant generation policies. Finally, in Section VII we summarize our findings and sketch future directions of research and development.

II. Mutation Equivalence and Redundancy

In [30], Yao et al put forth the following research question: *What are the causes of mutant equivalence?*. A priori, two factors affect the number of equivalent mutants generated in a mutation experiment: the attributes of the base program, P; and the mutant generation policy (defined, e.g. by the set of mutation operators that are deployed). For a fixed mutant generation policy, the research question of Yao et al may be reformulated as follows: *What attributes of a program make it prone to produce equivalent mutants?* Before we answer this question, we make the following observation: The attribute that makes a program prone to generate equivalent mutants is the same attribute that makes it fault tolerant; indeed, a fault tolerant program is one that can continue to deliver equivalent functionality despite the existence (and sensitization/ propagation) of faults, including faults simulated by mutations [2, 15, 22]. But we know very well what attribute makes programs fault tolerant: *Redundancy*.

Hence if we had a way to quantify the redundancy of a program, we could possibly use it to estimate the number of equivalent mutants produced from the program. Specifically, we are interested to estimate the *Ratio of Equivalent Mutants* (*REM*, for short) of a program P, i.e. the proportion of equivalent mutants out of a set of mutants generated from P by the relevant mutant generation policy.

The purpose of this section is to discuss metrics that quantify several aspects of the redundancy of a program. These metrics are defined by means of Shannon's entropy function [28]; we use the notations $H(X)$, $H(X|Y)$ and $H(X,Y)$ to denote, respectively, the entropy of random variable X, the conditional entropy of X given Y, and the joint entropy of random variables X and Y; we assume that the reader is familiar with these concepts, their interpretations, and their properties [8]. For each metric, we briefly present its definition, its interpretation, how we calculate it, and why we believe that it is correlated to (because it affects) the REM of a program. Because the REM is a ratio that ranges between 0 and 1, we resolve to define all our metrics as values between 0 and 1, so as to facilitate the derivation of a regression model. For the sake of simplicity, we compute all entropies under the assumption of equal probability. Under this assumption, our estimates are in fact upper bounds of the actual entropy; to get a more accurate estimate of the entropy, we would need to involve probability distributions for the various random variables that we refer to. This would make the model significantly more complex, without a commensurate increase in model quality.

Most of the metrics introduced below are due to [21], where they are presented as measures of fault tolerance in a program; this is hardly surprising since, as we argue above, the attributes that make a program fault tolerant are the same as the attributes that make it prone to produce equivalent mutants.

A. State Redundancy

What we want to represent: When we declare variables in a program, we do so for the purpose of representing the states of the program; for a variety of reasons, it is very common to find that the range of values that program variables may take is much larger than the range of values that actual/ feasible program states may take. For example, if we declare a variable x of type integer to represent the age of a person, then realistically x ranges between zero and (to be optimistic) 150 or so; yet as an integer, x may take values between $MinInt$ and $MaxInt$. Also, if we declare three integer variables, say $year$, $birthYear$ and age to represent (respectively) the current year, the year of birth of a person, and the person's age, then not only is each of these variables limited to a small range ($2019 \leq year \leq 2169$, $1869 \leq birthYear \leq 2019$, $0 \leq age \leq 150$), they also maintain an invariant relation between them ($year = birthYear + age$); yet as three integer variables, they range over a very broad state space ($[minInt...maxInt]^3$). We want *state redundancy* to reflect the gap between the declared state and the actual state of the program.

How we define it: If we let S be the declared state of the program, and σ be the actual state of the program, then the state redundancy of the program can be measured by the difference between their respective entropies; to normalize it (so that it ranges between 0.0 and 1.0) we divide it by the entropy of the declared state. Recognizing that the entropy of the actual state decreases (hence the redundancy increases) as the execution of the program proceeds from the initial state to the final state, we define, in fact two different measures of state redundancy, one for each state.

Definition 1: Given a program P whose declared state (defined by its variable declarations) is S, we let σ_I and σ_F be its initial and final actual states, we define its *initial state redundancy* and its *final state redundancy* as, respectively:

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

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

How we calculate it: To compute $H(S)$ we use a table that maps each data type to its width in bits,

Data type	Entropy (bits)
`bool`	1
`char`	8
`int`	32
`float`	64

Table 1: Entropy of Declared State

As an example, we consider the following program:

```
void P()
   {int year, birthYear, age;
    read (year, birthYear);
    assert ((2019 <= year <= 2169)     // initial
        && (1869 <= birthYear <= 2019));// state
    age = year - birthYear;        // final state
    }
```

The declared space of this program is defined by three integer variables, hence $H(S) = 96$ bits. Its initial state is defined by two variables ($year$ and $birthYear$) that have a range of 151 distinct values and an integer variable (age) that has free range, hence $H(\sigma_I) = 32 + 2 \times \log_2(151) = 46.48$ bits. As for the final state, it is determined fully by the values of $year$ and $birthYear$, since age is a function of these two, hence its entropy is merely: $H(\sigma_F) = 2 \times \log(151) = 14.48$ bits. Hence:

$$SR_I = \frac{96 - 46.48}{96} = 0.516.$$

$$SR_F = \frac{96 - 14.48}{96} = 0.85.$$

Of course, there is more redundancy in the final state than in the initial state.

Why we feel it is correlated to the REM of a program: State redundancy reflects the amount of duplication of the information maintained by the program, or the amount of extra bits of information that are part of the declared state; the more duplicated bits or unused bits are lying around in the program state, the greater the likelihood that a mutation affects bits that are not subsequently referenced in the execution of the program (hence do not affect its outcome). In the example above, if we did not have a variable for age, then whenever the age is needed, we would compute it as the difference between $year$ and $birthYear$; which means any modification (caused by a mutation operator) of either variable $year$ or variable $birthYear$ would produce an erroneous estimate of age. By contrast, with variable age, a modification of $year$ or $birthYear$ that intervenes after variable age is assigned, will not affect the estimate of the person's age.

B. Non Injectvity

What we want to represent: A function f is said to be *injective* if and only if it maps different inputs onto different outputs, i.e. $x \neq x' \Rightarrow f(x) \neq f(x')$. A function is non-injective if it violates this property; it is all the more non-injective that it maps a larger set of distinct inputs onto a common output. Typical programs are vastly non-injective: a sort routine over an array of size N maps $N!$ (N factorial) different input arrays (assuming all cells are distinct) onto a single array (the sorted permutation of the inputs).

How we define it: For the purposes of this metric, we view a program as mapping initial states onto final states. One way to quantify non-injectivity is to use the conditional entropy of the initial state given the final state: this entropy reflecte the uncertainty we have about the initial state given that we know the final state; this entropy increases as more initial states are mapped to the same final state; to normalize it, we divide it by the entropy of the initial state.

Definition 2: Given a program P on space S, the *non-injectivity* of P is denoted by NI and defined by:

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

where σ_I and σ_F are, respectively, the initial and final actual state of P.
Because σ_F is a function of σ_I, the conditional entropy can be simplified [8], yielding the following formula:

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

In [3], Androutsopoulos et al. introduce a similar metric, called *squeeziness*, which they find to be correlated to the probability that an error arising at some location in a program fails to propagate to the output. We differ from the thesis of Androutsopoulos et al. regarding how to interpret the probability of failure of error propagation from some program label L to the output: whereas a probability of 1.0 sounds like a massive failure, we think of it as a resounding success, since it means that the alleged error never causes a failure. Hence the issue for us is not *Why hasnt the error at* L *propagated to the output?*, but rather *Why are we claiming that there is an error at* L *when it never causes a failure?*.

How we calculate it: We have already discussed how to compute the entropies of the initial state and final state of a program. As an illustration, we find that the non-injectivity of the program cited in Section II-A is:

$$NI = \frac{H(\sigma_I) - H(\sigma_F)}{H(\sigma_I)} = \frac{46.48 - 14.48}{46.48} = 0.69.$$

Why we feel it is correlated to the REM of a program: One of the main sources of mutant equivalence is the ability of programs to mask errors that have infected the state, by mapping the erroneous state onto the same output as the correct state. This happens all the more frequently that the function of the program is more non-injective; hence non-injectivity measures exactly the capability of the program to mask errors caused by the sensitization of mutations. We fully expect non-injectivity to play a role in producing equivalent mutants.

C. Functional Redundancy

What we want to represent: Not all programs can be faithfully modeled as mappings from initial states to final states, as we do in Section II-B; sometimes a more faithful model of a program may be a heterogeneous function from some input space X to some output space Y. Programs exchange information with their environment through a wide range of channels: they receive input information (X) through read statements, passed by-value parameters, access to global variables, etc; and they send output information (Y) through write statements, passed by-reference parameters, return statements, access to global variables, etc. We want a metric that reflects redundancy for this model of computation.

How we define it: We let X be the random variable that represents all the input information used by the program, and we let Y be the random variable that represents all the output information that is delivered by P.

Definition 3: Given a program P that takes input X and returns output Y, the *functional redundancy* of P is denoted by FR and defined by:

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

Because Y is a function of X, we know [8] that the conditional entropy $(H(X|Y))$ can be written as $(H(X) - H(Y))$. Also, the entropy of Y is less than or equal to the entropy of X, and both are non-negative, hence FR ranges between 0 and 1 (we assume, of course, that $H(X) \neq 0$).

How we calculate it: The entropy of X is the sum of the entropies of all the input channels and the entropy of Y is the sum of the entropies of all the output channels. We consider the following program:

```
int P(int year, birthyear)
   {read (year, birthYear);
    assert ((2019 <= year <= 2169)
         && (1869 <= birthYear <= 2019));
    age = year - birthYear;
    return age;
    }
```

Random variable X is defined by program variables $year$ and $birthYear$, hence its entropy is $H(X) = 2 \times \log_2(151) = 14.48$. Random variable Y is defined by variable age, whose values range between 0 and 150, hence $H(Y) = \log_2(151) = 7.24$ bits. Hence

$$FR = \frac{14.48 - 7.24}{14.48} = 0.50.$$

Why we feel it is correlated to the REM of a program: Functional redundancy, like non-injectivity, reflects the program's ability to mask errors caused by mutations; whereas non-injectivity models the program as a homogeneous function on its state space, functional redundancy models it as a heterogeneous mapping from an input space to an output space.

All the metrics we have discussed so far pertain to the base program; we refer to them as the program's *intrinsic metrics*. The metric we present in the next section deals not with the base program, but rather with the oracle that is used to rule on equivalence.

D. Non Determinacy

What we want to represent: Whereas we would like to think that mutation equivalence is a well-defined property (as the property of a mutant to be semantically equivalent to the base program), we consider the following example as an illustration that there may be some ambiguity in what we mean by this term. Let P_1, P_2 and P_3 be the following three programs:

P_1 {`int x,y,z; z=x; x=y; y=z;`}.
P_2 {`int x,y,z; z=y; y=x; x=z;`}.
P_3 {`int x,y,z; x=x+y; y=x-y; x=x-y;`}.

We ask the question: are these three programs semantically equivalent? We all know that these three programs are intended to swap the values of x and y; hence we are tempted to rule that they are equivalent. But our interpretation is a subjective judgement that does not reflect any intrinsic property of these programs; we may want to think of them as equivalent only because we assume that their intent is to swap x and y, and that z is used (in P_1 and P_2) merely as an auxiliary variable.

Notwithstanding our interpretation of what these programs are intended to do, we argue that whether these programs are equivalent depends on the non-determinacy of the oracle that we use to test their output. Let us consider an execution of P_1, P_2, P_3 on some input data, and let x_i, y_i, z_i be the final value delivered by program P_i for this input data. If the oracle we use is:

$$\Omega_D \equiv (x_i = x_j) \wedge (y_i = y_j) \wedge (z_i = z_j)$$

for $1 \leq i, j \leq 3$, $i \neq j$, then we find that these programs are not equivalent, whereas if we use the oracle

$$\Omega_{ND} \equiv (x_i = x_j) \wedge (y_i = y_j)$$

then we find that they are equivalent; the difference between these oracles is that Ω_D is deterministic, whereas Ω_{ND} is not deterministic.

From this discussion we infer that the equivalence between a base program P and a mutant M may depend on what oracle is used to compare the output of P with the output of M, and we are interested to define a metric that reflects the degree of non-determinacy of the selected oracle.

We acknowledge that this broader definition of equivalence may be controversial, but it is a definition that generalizes, and does not contradict, the traditional understanding of (strict) equivalence. The traditional understanding is subsumed by this definition, by adopting an oracle of equivalence that tests all program variables for equality.

How we define it: We are given a program P on space S and a mutant M on the same space, and we consider an oracle $\Omega()$ on S defined by an equivalence relation on S. We want the *non-determinacy* of $\Omega()$ to reflect how much uncertainty we have about the output of M for a given input if we know the output of P for the same input.

Definition 4: Given a program P and a mutant M on space S, and given an oracle $\Omega()$ defined as an equivalence relation on S, we let S_P and S_M be the random variables that represent the final states of P and M for a common input. The *non-determinacy* of $\Omega()$ is denoted by ND and defined by:

$$ND = \frac{H(S_P|S_M)}{H(S_P)}.$$

How we calculate it: The conditional entropy $H(S_P|S_M)$ is really the entropy of the equivalence classes of S modulo the equivalence relation defined by $\Omega()$. It represents the amount of uncertainty we have about an element of S if all we know is its equivalence class; if $\Omega()$ is the identity relation then all equivalence classes are singletons and $ND = 0$; else it is the base 2 logarithm of the size of equivalence classes. As an example, we consider space S defined by three variables, say x, y, z of type integer, and we show in the following table a number of possible oracles with their respective non-determinacies. For all these oracles, $H(S_P) = 3 \times 32 = 96$; the only term that changes is $H(S_P|S_M)$.

$\Omega()$	$H(S_P\|S_M)$	ND
$(x_P = x_M) \wedge (y_P = y_M) \wedge (z_P = z_M)$	0	0
$(x_P = x_M) \wedge (y_P = y_M)$	32	0.33
$(x_P = x_M)$	64	0.66
true	96	1.0

Table 2: Non Determinacy of Sample Oracles

Why we feel it is correlated to the REM of a program: Of course, the weaker the oracle that tests for equivalence, the more mutants will be found to be equivalent to the base program.

III. Empirical Evidence

In the previous section we presented analytical arguments to the effect that our semantic metrics have an impact on the REM of a program; in this section we wish to consolidate these analytical speculations with empirical evidence. In [5] we report on a small scale experiment where we take 19 software methods from the *Apache Common Mathematics Library* (http://apache.org/), compute their metrics and estimate their REM; then we analyze statistical relationships between them. Specifically, we proceed as follows:

- For each method, we compute the intrinsic redundancy metrics, i.e. the initial and final state redundancy, the non-injectivity, and the functional redundancy.
- We use PITEST (http://pitest.org/) in conjunction with MAVEN (http://maven.apache.org/) to generate mutants and test them for possible equivalence with the base program using the associated test data. PITEST is deployed with its default mutant generation operators, which are: increments mutator, void method call mutator, return vals mutator, math mutator, negate conditionals mutator, invert negs mutator, and conditionals boundary mutator. To ensure that surviving mutants are most likely to be equivalent, we monitor coverage metrics and mandate that line coverage exceeds 90%; this may sound too low, but we are making provisions for the possibility of dead code, and exception handling code, etc. Still, in the vast majority of cases line coverage is 100%. Also, these tests for equivalence use an oracle that checks the identity of all the state variables, which yields a non-determinacy value of zero ($ND = 0$).
- For some cases, we also vary the oracle that we use to test for equivalence, whereby we replace the equality by a weaker equivalence relation, and record the correponding value of non-determinacy. As an example, we let the space of the program be defined by a single integer variable, say x, and we show below some examples of equivalence relations, along with the corresponding values of $H(S_P|S_M)$ and ND.

$\Omega()$	$H(S_P\|S_M)$ (bits)	ND
$(x_P = x_M)$	0	0.00
$(x_P \textbf{ mod } 2^{31} = x_M \textbf{ mod } 2^{31})$	1	0.03
$(x_P \textbf{ mod } 2^{16} = x_M \textbf{ mod } 2^{16})$	16	0.50
$(x_P \textbf{ mod } 2^{8} = x_M \textbf{ mod } 2^{8})$	24	0.75
$(x_P \textbf{ mod } 2 = x_M \textbf{ mod } 2)$	31	0.97
true	32	1.00

Table 3: Non Determinacy for Lone Integer Variable

By varying ND for the same component, we get different values of REM, and different data points; in total, we find 27 data points.

Using this small data sample, we analyze the statistical significance of the correlation between REM and the proposed metrics, and we derive a tentative regression model of REM:

$$\log(\frac{REM}{1-REM}) =$$

$$-2.765 + 0.459 \times FR + 2.035 \times NI + 0.346 \times ND.$$

The residuals of this regression formula with respect to the actuals fall below 0.1 for all of the sample except two elements.

This regression model was developed as a proof-of-concept, to provide empirical support for our conjecture that redundancy metrics are a means to estimate the REM of a program. In the absence of an automated tool to compute the redundancy metrics, we had to compute them by hand; this forced us to select a small sample (19) of small software components (a few dozen LOC at most). In order to develop a useful regression model, we need to automate the calculation of redundaqancy metrics; this is what we discuss in the next section.

IV. Requirememnts Specifications

A. System Requirements

To build a useful tool that provides reliable estimates of the REM of a program, we are taking the following steps:

- First, we use compiler generation technology to derive a Java compiler that computes the proposed redundancy metrics for Java methods in classes; this has been virtually completed as far as SR_I, SR_F, NI, and FR. Computing ND is a different matter because it requires that we analyze the test oracle, rather than the base program; this is currently in progress.
- We use the compiler to build a large data set of sample Java programs, for which we compute the intrisic metrics (SR_I, SR_F, NI and FR).
- We are concurrently computing the REM of selected software components, using PITEST and MAVEN (referenced above) under the test oracle that checks for identity ($ND = 0$). This is currently ongoing, and we are using the mutant generation policy that PITEST offers as default; it includes seven mutation operators, which are listed in Section III.
- We envision to revisit the statistical analysis of the correlation between the intrisic metrics and the REM, and build a new regression model that estimates REM as a function of the metrics, as they are computed by the compiler. Of course, this regression model is valid only for the mutant generation policy that is used to build the data set; in Section VI we discuss how we deal with varying mutation policies.
- Whereas the model of Section III includes ND in the regression, we resolve to exclude ND from the regression model because it plays a distinct role in determining equivalence.
- Let $\rho(SR_I, SR_F, NI, FR)$ be the regression formula that we derive from the statistical analysis; we resolve to let the actual REM estimate vary linearly between $\rho(SR_I, SR_F, NI, FR)$ and 1.0 as ND varies between 0.0 and 1.0. For $ND = 0$ we get $REM = \rho(SR_I, SR_F, NI, FR)$, since that is the regression formula derived when non-determinacy is 0.0; and for $ND = 1$ we get $REM = 1$, of course, since then the oracle is merely **true** , i.e. all the mutants pass the equivalence test regardless of their intrinsic metrics. Of course in practice most cases will have a value of ND closer to 0.0 than 1.0, but the linear formula allows REM to grow linearly with the non-determinacy of the oracle, which makes sense, intuitively. Hence we propose the following formula for the estimate of REM:
$$REM = \rho(SR_I, SR_F, NI, FR) \\ + ND \times (1 - \rho(SR_I, SR_F, NI, FR)).$$

B. Inputs and Outputs

We want our tool to compute the REM of individual methods within Java classes; to this effect, we envision the following set of inputs:

- A directory where the Java class is located.
- Once the directory is selected, the system opens a dropdown menu where the user selects a Java class.
- Once the class is selected, the system opens a drop down menu where the user selects a method.
- Once the method is selected, the system presents default values for a number of parameters that are used to compute the metrics, and gives the user the opportunity to override these defaults.
- Finally, the user selects the mutation generation policy. We have not decided yet how the user selects the mutation policy, and we are considering two options: either we offer a dropdown menu of alternative policies (each defined by a set of mutators) and we let the user select one; or we offer a menu of mutators and we let the user select those she/he wishes to deploy. This is discussed in Section VI.

For now, we estimate ND off-line, by inspection of the source code of the test oracle in PITEST/ MAVEN; but in the future we envision to automate the calculation of ND by submitting the code of the oracle as part of the input.

The output of our tool includes the following items:

- All the identifying information of the selected method, along with relevant details, such as the method size.
- The total number of mutants that are generated from the method by the selected mutation policy.
- The estimated number of equivalent mutants, obtained by mutiplying the estimated REM by the total number of mutants. This estimate of the REM stems from a temporary regression model that was developed using partial data, for the limited purposes of this experiment; we are waiting to complete and validate data collection before we derive a more reliable regression model (for PITEST's default mutation policy).

V. Design and Ongoing Implementation

The core component of our proposed tool, in terms of complexity and in terms of criticality, is the Java compiler that computes the intrinsic metrics of a method in a class; to compute these metrics, we need to evaluate the following quantities: $H(S)$, $H(\sigma_I)$, $H(\sigma_F)$, $H(X)$ and $H(Y)$. We briefly discuss these below.

A. Entropy of Declared State

From the standpoint of a method in a class, the declared space is made up of four components, yielding four terms of the entropy:

- HG: Entropy of the global space, i.e. the space defined by the declared fields of the class; these are class wide variables that are accessible to all the methods of the class.
- HP: Entropy of the space defined by the parameters that are passed to the method.
- HL: Entropy of the local space, i.e. the space defined within the scope of the method.

- HB: Entropy of variables that may occasionally be declared within blocks in the method; we hesitated whether such variables ought to be counted, but we decided to count them since mutators apply equally well to all the variables of a method, regardless of their scope.

These four compiler variables are set to zero at the start of the compilation and are incremented (by the right amount, according to the type of the variable, re: Section II-A) each time a relevant variable declaration is encountered in the source code. At the end of the compilation, we set:

$$H(S) = HG + HP + HL + HB.$$

B. Entropy of Initial Actual State

When we look at the source code of a method, we can see what variables are declared but we cannot see what actual state space the programmer intends to represent with these variables. Hence we depend on the programmer to give us some information about the actual space, by means of an `assert()`-like statement. In order for our compiler to recognize the `assert()` statement that is included for this purpose, as opposed to the normal use of `assert()` statements, we define a special syntax for this: we call this statement `preassert()`, and we let it take the same parameter as a normal `assert()` statement, i.e. a logical formula involving the variables that are in scope. This is the only modification we make to the syntax of Java.

If the user includes no `preassert()` statement in the method, then we take $H(\sigma_I) = H(S)$, else we take

$$H(\sigma_I) = H(S) - DH,$$

where DH measures the reduction in state entropy that results from the logical formula provided by the `preassert()` statement. Note that if the method includes an exception handling statement at the beginning of its body, to enforce a specific precondition, then we can use the negation of the condition that triggers the exception as the parameter of `preassert()`; this gives us a free/ trivial source of `preassert()` conditions, even when we know nothing else about the method.

We compute DH according to the following rules:

- $DH(\mathbf{true}\ ) = 0$.
- $DH(f1 \wedge f2) = DH(f1) + DH(f2)$, where $f1$ and $f2$ are logical formulae.
- $DH(f1 \vee f2) = max(DH(f1), DH(f2))$.
- $DH(Exp1\ ==\ Exp2)$, where $Exp1$ and $Exp2$ are expressions of the same type, equals to the entropy associated with the type of the expressions. So, for example if space S is defined by three integer variables, say x, y, z and we encounter the statement `preassert(x+1==y-1)` then we find $DH = 32$ bits, since the expressions are of type `int`, and 32 is the entropy of integers; hence, $H(\sigma_I) = 96 - 32 = 64$.
- As a simplifying assumption, we assume that all comparative formulas (<, >, <=, >=) yield $DH = 1$, becausse they divide the size of the state space by two, hence the base-2 log of the size is reduced by 1. Hence for example if space S is defined by a single integer variable, say x, and we have `preassert(x>0)`, we find $DH = 1$ and $H(\sigma_I) = 31$ bits, which is the entropy of a positive integer variable.

Note that this is an imperfect solution: if we are given a statement such as $preassert(x > 0 \wedge x < 100)$ where S is defined by the single integer variable x, then $H(\sigma_I)$ ought to be $\log_2(99) = 6.63$ bits, but our system will find 30 bits, since it will find $DH = 2$. But getting the compiler involved in detailed analysis of the logical formulas and their impact on DH would be very complicated, and would only yield minor enhancement in precision; hence we are not considering it for now.

C. Entropy of Final Actual State

The excution of the method applies a function to the initial state of the method to obtain the final state; with the exception of injective functions (which are very rare in practice, as most programs are vastly non-injective), any function application reduces the entropy of the state. To keep track of this reduction, we catalog all the variables that the method has access to, and keep a record of the functional dependencies that the execution of the method creates between state variables; at the end of the method, we review the independent variables that have an effect on the final state, and we compute their entropy. For example, consider space S defined by integer variables x, y and z, and suppose that initially all three variables may take any representable integer value. Hence $H(\sigma_I) = 96$ bits. We consider the following code segment:

`P1:` {`y=x;x=z;`}

The function of this program is defined by:

$x' = z \wedge y' = x \wedge z' = z$.

The final state depends on only two initial values, namely x and z, hence the entropy of the final actual state is: $H(\sigma_F) = 64$ bits. If we consider the following program:

`P2:` {`x=z;y=x;`}

The function of this program is defined by:

$x' = z \wedge y' = z \wedge z' = z$.

The final state depends only on z, hence the entropy of the final actual state is: $H(\sigma_F) = 32$ bits.

To automate this process, we introduce a Boolean matrix, which we call D (for dependency), which is a square matrix that has as many rows and columns as there are variables in S.

- Initially (as we start scanning the code of the method), D is initialized to the identity matrix ($\mathbf{true}$ on the diagonal, $\mathbf{false}$ everywhere else); initially, every variable depends only on itself.
- Whenever we encounter an assignment statement of the form {`x=E(x,y,z...)`}, where E is an expression, we replace the row of x in D with the Boolean sum (logical OR) of the rows of all the variables that appear in the expression (including x, if it does); this action updates the list of initial values on which the current value of x depends.

- At any point during the scan of the body of the function the matrix entry $M[i,j]$ contains **true** if and only if the value of variable i at the current location is affected by the initial value of variable j.
- Upon entering the body of an `if-ten` statement or a `while` statement, we record the list of variables that are referenced in the condition (of the `if` or `while`). Then whenever we encounter an assignment statement, we do the same as above, but we add the stored variable names to the list of variables in the right hand side of the assignment.
- Upon encountering an `if-then-else` statement, we record the list of variables that are referenced in the condition, and we duplicate the current D matrix, using one copy for the `then-clause` and one copy of the `else-clause`. Within the `then-clause` and the `else-clause`, we use the same rule as above: the variables that appear in the condition are considered as part of the right hand side of any assignment.
- At the end of the `if-then-else` statement, we compare the two dependency matrices (obtained from the `then-clause` and the `else-clause`) and select that which shows the smallest entropy (computed as detailed below).

When we reach the end of the method, we compute the entropy of the final actual state as follows: we define a boolean vector that has the same size as rows of D, and we use it to compute the Boolean sum of all the rows of D. This vector shows the set of state variables on which the final actual state depends; in the two simple examples discussed above, these vectors would be, respectively:

For $P1$: (**true** , **false** , **true**).

For $P2$: (**false** , **false** , **true**).

Then we compute the entropy of the final actual state as the sum of the entropies of the variables that have **true** in the vector. We find 64 bits for $P1$ and 32 bits for $P2$.

Implementation of this algorithm gets more complicated when we have variable names that refer to arrays; the discussion of arrays is beyond the scope of this paper, but basically we want to avoid counting the entropy of a whole array if we have used only one (or a few) of its cells.

D. Entropy of Input Channel

The input channel of a method includes: the parameters that are passed to the method by value; the parameters of type class (which, we understand, Java passes implicitly by reference), and the global variables that are referenced on the right hand side of assignment statements. The entropy of the input channel, $H(X)$, is the sum of the entropies of all these variables.

E. Entropy of Output Channel

The output channel of a method depends on whether the method is declared as `void` or has an explicitly declared return type:

- If the method has an explicitly declared return type, then the entropy of that type is the value for the output channel entropy.
- If the method is declared as a `void` method, then the output channel is made up of the following components: the parameters of type class (which are implicitly passed by reference); the global variables that appear on the left of an assignment statement. To compute the entropy of the output channel, we use the dependency matrix D introduced in Section V-C; whereas the entropy of the final state is computed by adding all the rows of D, the entropy of $H(Y)$ is computed by adding the rows of D that correspond to the output variables cited above.

VI. Varying the Mutant Generation Policy

When we produce a regression model based on empirical data obtained by deploying a particular mutant generation policy, then it stands to reason that our estimate is valid only as long as we use the same policy. How can we accommodate a variety of policies? We currently envision two possibilities to do this:

- Either we select a number of well-known, widely used and / or widely researched generation policies, and generate a regression model for each. Then our tool offers the user a menu of policies and asks the user to select one; then the tool uses the corresponding regression formula.
- Or we select a number of well known mutation operators, generate a regression for each mutator applied individually. Then our tool offers the user a menu of operators and asks him/ her to select all those she/ he wishes to apply. Then the tool estimates the REM that stems from each mutator; but then it needs to combine the individual REM's to estimate the overall REM obtained by combining the mutation operators. This approach raises the question of how we combine individual REM's corresponding to single mutators to obtain the REM of the aggregate policy. In [20] we speculate that the following formula is a good approximation, and we provide some empirical evidence to this effect:

$$REM = 1 - \prod_{i=1}^{N}(1 - REM_i),$$

where N is the number of operators, and REM_i is the REM obtained for operator i when it is deployed by itself. This approach, if it is indeed validated offers greater flexibility than the first, but also presents greater risk of imprecision; this matter is under investigation.

VII. Conclusion

A. Summary

There are three reasons for a mutant to be equivalent to a base program[1]:

[1]For the sake of this discussion, we use the terminology of *fault*, *error* and *failure* introduced by Laprie et al. [4, 18, 19].

- Either the mutation does not create a fault (the mutation does not cause the production of erroneous states at run-time).
- Or the mutation does create a fault but whenever the fault is sensitized and causes an error in the state of the program, the error is subsequently masked and does not cause a failure.
- Or the mutation does create a fault and the fault does cause errors and the errors do occasionally propagate to cause a failure, but the failure falls within the tolerance of the oracle that is used to rule on equivalence.

The metrics we propose in this paper take into account these three possibilities:

- State redundancy metrics (SR_I and SR_F) reflect the likelihood that a mutation does not create a fault; indeed, the more redundancy the state of the program carries, the greater the likelihood that a mutation only affects bits that are not critical to the representation of the state.
- Non Injectivity (NI) and functional redundancy (FR) reflect the likelihood that even if the mutation creates a fault and the fault is sensitized (to produce an error), it does not cause a failure (i.e. the mutant produces the same output as the base program).
- Non determinacy (ND) reflects the likelihood that even if the mutation creates a fault and the fault produces an error and the error propagates to cause a failure, the oracle finds the behavior of the mutant to be equivalent (albeit not identical) to the behavior of the base program.

Hence we do expect that the proposed metrics are useful in predicting the rate of equivalent mutants of a program for a given definition of equivalence.

B. Threats to Validity

We identify the following vulnerabilities in the approach of this work:

- All the metrics proposed in this paper apply to programs where the space is well defined, with a clearly defined entropy, and easy to model transformations; these attributes characterize numeric programs, but may not apply readily to programs where the space includes arbitrary data structures, and transformations involve method calls with a wide range of diverse effects.
- It is very difficult to perform an empirical validation of the statistical models, because we can never assert with certainty that a mutant is equivalent to the base program, regardless of how much testing we perform. Hence we are limited to analytical validation, and to approximate empirical validation.

C. A Research Agenda

Tasks that remain to be completed include the following:

- Complete the implementation and test of the Java analysis component, which computes the intrinsic metrics of a given method in a given Java class.
- Use the metrics calculator to build a large data set of arbitrarily large/ complex methods for which we have computed the intrinsic metrics.
- Run experiments on the selected methods/ classes to compute their actual REM.
- Performs a regression analysis to derive a formula for REM as a function of the intrinsic metrics.
- Use the generated regression formula to estimate the REM of a base program as a function of its intrinsic metrics.
- Resolve the dilemma of whether we want to vary mutation generation policy one set of mutators at a time, or one mutator at a time.
- Automate the calculation of the ND metric, by generating a compiler that analyzes the source code of the oracle used to determine equivalence.
- Integrate all these components into a tool that takes as input the base program, the oracle, and the mutant generation policy, and returns the set of generated mutants as well as well as the estimate of the number of these mutants that is equivalent to the base program.

Another intriguing venue that arises from the computation of the REM is the possibility that REM can be used to estimate the number of non-redundant mutants in a pool of generated mutants. Indeed, it is legitimate to assume that the REM of the base program is the same as the REM of every mutant in the pool, since mutants differs very little from the base. Hence the question of redundant mutants can be formulated as follows: Given a pool of N programs that have probability REM of being equivalent (and probability $(1 - REM)$ of being pairwise distinct), what is the mean number of distinct programs in the pool; this matter is currently under investigation. Ultimately, we want to use the approach adopted here to consider a base program P and a mutant generation policy, and be able to:

- Compute the number M of mutants obtained from P by the selected mutant generation policy.
- Estimate the REM of P for the selected mutation policy, and infer from it the number of equivalent mutants, say $EQ = REM \times M$.
- Given the number of non-equivalent mutants, $N = M - EQ$, estimate the number of distinct (non-redundant) mutants among these N mutants.

Acknowledgement

The authors are very grateful to the anonymous reviewers for their valuable feedback. This work is partially supported by the NSF grant number DGE1565478.

References

[1] Konstantinos Aadamopoulos, Mark Harman, and Robert Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings, Genetic and Evolutionary Computation*, volume 3103 of *LNCS*, pages 1338–1349, 2004.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings, ICSE*, 2005.

[3] Kelly Androutsopoulos, David Clark, HaiTao Dan, Robert M. Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings, ICSE 2014*, 2014.

[4] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl E Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] Amani Ayad, Imen Marsit, Nazih Mohamed Omri, and Ali Mili. Using semantic metrics to predict mutation equivalence. Technical report, NJIT, Newark, NJ, USA, http://web.njit.edu/˜mili/milimutation.pdf, 2018.

[6] Tim A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, March 1982.

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

[8] Imre Csiszar and Janos Koerner. *Information Theory: Coding Theorems for Discrete Memoryless Systems*. Cambridge University Press, 2011.

[9] Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Proteum /im 2.0: An integrated mutation testing environment. In W. Eric Wong, editor, *Mutation Testing for the New Century*, volume 24, pages 91–101. Springer Verlag, 2001.

[10] B.J. Gruen, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Proceedings, MUTATION 2009*, Denver, CO, USA, 2009.

[11] R.M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Journal of Software Testing, Verification and Reliability*, 9(4), 1999.

[12] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Procedings, 36th International Conference on Software Engineering*. ACM Press, 2014.

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

[14] R. Just, M.D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings, ISSTA'14*, San Jose, CA, USA, 2014.

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

[16] Rene Just, Michael D Ernst, and Gordon Fraser. Using state infection conditions to detect equivalent mutants and sped up mutation analysis. In *Proceedings, Dagstuhl Seminar 13021: Symbolic Methods in Testing*, 2013.

[17] M. Kintis, M. Papadakis, Y. Jia, N. Malveris, Le Traon Y, and M. Harman. Detecting trivial mutant equivalences via compiler optimizations. *IEEE Transactions on Software Engineering*, 44(4), 2018.

[18] Jean Claude Laprie. Dependability —its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–19. Springer Verlag, 1995.

[19] Jean Claude Laprie. Dependable computing: Concepts, challenges, directions. In *Proceedings, COMPSAC*, 2004.

[20] Imen Marsit, Mohamed Nazih Omri, Jimeng Loh, and Ali Mili. Impact of mutation operators on the ratio of equivalent mutants. In *Proceedings, SOMET*, Granada, Spain, 2018.

[21] Ali Mili, Ali Jaoua, Marcelo Frias, and Rasha Gaffer Helali. Semantic metrics for software products. *Innovations in Systems and Software Engineering*, 10(3):203–217, 2014.

[22] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings, ISSTA*, 2011.

[23] Simona Nica and Franz Wotawa. Using constraints for equivalent mutant detection. In C. Andres and L. Llana, editors, *Second Workshop on Formal methods in the Development of Software*, EPTCS, pages 1–8, 2012.

[24] A. J. Offut and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, September 1997.

[25] Mike Papadakis, Marcio Delamaro, and Yves LeTraon. Mitigating the effects of equivalent mutants with mutant clasification strategies. *Science of Computer Programming*, 95(P3):298–319, December 2014.

[26] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Compugters*. 2019.

[27] David Schuler and Andreas Zeller. Covering and uncovering equivalent mutants. In *Proceedings, International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[28] C. Shannon. A mathematical theory of communication. *Bell Syst. Tech. Journal*, 27:379–423, 623–656, 1948.

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

[30] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings, ICSE*, 2014.