Abstract

Mutation testing is the art of generating syntactic versions (called mutants) of a base program, and is used in several applications, 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 error prone task. In this paper we argue that for most intents and purposes, it is not necessary to identify equivalent mutants individually; it suffices to estimate their number. Specifically, we argue that the ratio of equivalent mutants (REM) that a base program is prone to yield can be inferred from a static analysis of the program; furthermore, we find that the REM of a program can be used to analyze many mutation related aspects of a program.

Keywords:
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 reality 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 data set \( T \) kills 80 of them, 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; on the other hand, if we know that only five mutants are equivalent, then \( T \) has missed 15 mutants.

- If test data set \( T \) 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.

The issue of mutant equivalence has been the focus of much research recently [13, 10, 38, 1, 31, 35, 21, 29, 2, 28, 23, 18, 20, 24, 37, 19, 12, 8, 17]. It is beyond the scope of this paper to do a survey of mutation equivalence research (see [33] 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:

- 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 distinct locally but still equivalent.

- Research that infers equivalence from comparing global behavioral envelopes; such methods are prone to loss of precision, because programs may have similar behavioral envelopes yet still be semantically distinct.

Determining whether two syntactically distinct programs are semantically equivalent is known to be undecidable [11]. Notwithstanding this theoretical result, attempting to determine whether two programs are semantically equivalent is a non-trivial, costly, and error-prone exercise; attempting to decide whether \( M \) mutants are equivalent to a base program is \( M \) times more difficult/impractical; attempting to decide whether \( M \) mutants are distinct from each other is \( M^2 \) 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 important questions.

2. Measures of Redundancy

In [38] 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 [2, 23, 28], 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 has a name: fault tolerance! We know what makes programs fault tolerant: redundancy.

Hence if only we could quantify the redundancy of a program, we can 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 for a program, over the total number of generated mutants. Of course, the ratio of equivalent mutants depends not only on the program, but also on the mutant generation policy; in this section, we assume a fixed mutant generation policy, and focus on the properties of a program that determine its REM, and in section 8 we discuss how the mutation operators factor into the REM of a program.

In this section, we discuss a number of quantitative measures of redundancy; 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 [36]; we assume the reader familiar with the main (simple) concepts and properties of this theory [16]. For the sake of simplicity, all entropies will be computed under the assumption of equal 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 $\sigma$ 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 $\sigma$ over the entropy of $S$. Since the entropy of $\sigma$ decreases as execution of the program proceeds, we resolve to define two versions of state redundancy, one for the initial actual state ($\sigma_I$) and one for the final actual state ($\sigma_F$); because $\sigma$ is a function of $S$, the conditional entropy $H(S|\sigma)$ can be written as $H(S) - H(\sigma)$ [16].

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

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

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

```c
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 $2 \times 32$ bits=64 bits. The initial actual state space $\sigma_I$ is defined by two positive integer variables, whose entropy is $2 \times 31$ bits=62 bits. The final actual state space $\sigma_F$ 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.

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

**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 fewer outputs there are for the artifact to choose from, the harder it is for a mutant operator to affect a change in the program’s output (as there are fewer outputs distinct from the original to choose from).

### 2.3. Non Injectivity

**How we define it.** We consider the program as defining 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 (\( \sigma_I \)) given the final actual state (\( \sigma_F \)). 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:

\[
H(\sigma_I) = 100 \times 32 + 2 \times \log(100) = 3213.29 \text{ bits}.
\]
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:

\[
H(\sigma_F) = 100 \times (32 - \log(100)) + \log(100) = 2528.97 \text{ bits.}
\]

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:

\[
\text{swap1()} \quad \text{swap2()} \quad \text{swap3()}
\]

\[
\begin{align*}
\{ &z=x; \\
&x=y; \\
&y=z; \}
\end{align*}
\begin{align*}
\{ &z=y; \\
&y=x; \\
&x=z; \}
\end{align*}
\begin{align*}
\{ &x=x+y; \\
&y=x-y; \\
&x=x-y; \}
\end{align*}
\]

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 (\( \sigma_F^P \)) and the final state of the mutant (\( \sigma_F^M \)). We define the non-determinacy of an oracle \( EQ(\sigma_F^P, \sigma_F^M) \) as the ratio of the conditional entropy of \( \sigma_F^P \) given \( \sigma_F^M \) over the entropy of \( \sigma_F^P \):

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

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

\[
\begin{align*}
EQ_1(\sigma_F^P, \sigma_F^M) &\equiv (x_F^P = x_F^M) \land (y_F^P = y_F^M) \land (z_F^P = z_F^M),
\end{align*}
\]

6
then $ND_1 = 0$ hence $H(\sigma_F^P|\sigma_M^F) = 0$. Indeed, if $EQ_1(\sigma_F^P, \sigma_M^F)$ holds then $H(\sigma_F^P|\sigma_M^F) = 0$ since knowing $\sigma_M^F$ leaves no uncertainty as to the value of $\sigma_F^P$.

On the other hand, if we define the oracle of equivalence as:

$$EQ_2(\sigma_F^P, \sigma_M^F) \equiv (x_F^P = x_M^F) \land (y_F^P = y_M^F),$$

then $H(\sigma_F^P|\sigma_M^F) = 32$ bits, hence $ND_2 = \frac{32}{96} = 0.33$. In this case, knowing $\sigma_M^F$ informs us about two variables of $\sigma_F^P$, $x$ 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 et al. [25, 26, 3] pertaining to faults, errors, and failures. We envision four 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 $(x \geq y)$; 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 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-redundancy ($ND$) reflects this exact property.

Table 1 summarizes the discussions of this section.

3. Estimating the Number of Equivalent Mutants

In the previous section, we argue that what causes a program to generate a large number of equivalent mutants is the amount of redundancy in the program; also, we define a number of quantitative measures of redundancy, and discuss why we believe that these may be statistically correlated to a program’s ratio of equivalent mutants (REM). In this section, we complement the analytical arguments presented in the previous section with empirical evidence to the effect that the REM of a program is statistically correlated to the redundancy metrics ($SR_I$, $SR_F$, $FR$, $NI$, and $ND$), and can be estimated through a regression model where these metrics are the independent variables.

Among these five 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:

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

Table 1: Mutant Equivalence vs. Redundancy Metrics

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, $ND$ 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 $EQ(\sigma^P_F, \sigma^N_M) \equiv \text{true}$ then we obtain an $REM$ of 1 regardless of all the other factors ($SR_I$, $SR_F$, $NI$, $FR$). Hence $REM$ varies between $\rho(SR_I, SR_F, NI, FR)$ and 1 as $ND$ varies between 0 and 1; see Figure 1.

To derive the regression model of $REM$ as a function of the program-specific redundancy metrics, we proceed as follows:

- We select components from the Apache Common Mathematics Library, version commons-math3-3.5-src. Our sample includes seven classes, totalling 147 methods.

- We use PITEST (http://pitest.org/) in conjunction with Maven (http://maven.apache.org/) to generate mutants of the methods in our sample, and test them for equivalence.

- We select the default mutation operators of PITEST, and we record, for each method, the number of generated mutants, as well as the number of mutants that were found to be equivalent, at least for the provided test data.

- We ensure that the test classes check for identity between all the state variables of the base program, to justify our hypothesis that $ND$ is zero.
• From the column of number of generated mutants and the column of number of equivalent mutants we derive a column that records the estimated REM of each component.

• We deploy the tool discussed in section 7 to compute the redundancy metrics of each method, and we record the results in a table.

• We derive the regression model \( REM = \rho(SRI, SR_F, NI, FR) \) from the resulting data table.

For the sake of precision, we derive several regression models, which vary by component size (different size ranges yield different models), mutant generation policy, component type, etc [4, 5].

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 data set \( T \) must be judged against the estimated number of non-equivalent mutants,

\[
N = (1 - REM) \times M
\]

rather than against \( M \).

4. Quantifying Mutation Redundancy

We consider a program \( P \) whose ratio of equivalent mutants is \( REM \) and we consider a test data set \( T \) 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?

Because the mutants differ only slightly from the base program (minor syntactic alterations) it is fair to assume that they have the same amount of redundancy as P, hence they also have the same REM. Hence we argue that REM represents the probability that any two mutants are equivalent to each other. 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.

- \( p(N, REM, N) = (1 - REM)^{N(N-1)/2} \).
  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 0.15 for the sample programs we have encountered in common benchmarks [22, 7] and in other empirical studies [32]), NEC(N, REM) is much smaller than \( N \); in particular, we could kill 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 the table below the value of $NEC(N, REM)$ for some sample values of $N$ and $REM$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>0.05</th>
<th>0.10</th>
<th>0.15</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>18</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>300</td>
<td>54</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>3000</td>
<td>98</td>
<td>55</td>
<td>38</td>
</tr>
</tbody>
</table>

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 eventually becomes zero for sufficiently large $k$. This is borne out by observation of $p(N, REM, k)$ in the examples shown above for $N = 3000$: The tables below 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 $p(N, \text{REM}, k)$ is maximal turns out to be $\text{NEC}(N, \text{REM})$ for all three values of $\text{REM}$.

For $\text{REM} = 0.05$:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$p(3000, 0.05, k)$</th>
<th>$k$</th>
<th>$p(3000, 0.05, k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>99</td>
<td>0.126499</td>
</tr>
<tr>
<td>14</td>
<td>4.94066e-324</td>
<td>110</td>
<td>0.000200261</td>
</tr>
<tr>
<td>87</td>
<td>8.35248e-05</td>
<td>111</td>
<td>6.43533e-05</td>
</tr>
<tr>
<td>88</td>
<td>0.000296773</td>
<td>240</td>
<td>8.67925e-320</td>
</tr>
<tr>
<td>97</td>
<td>0.114522</td>
<td>241</td>
<td>0</td>
</tr>
<tr>
<td><strong>98</strong></td>
<td><strong>0.126799</strong></td>
<td><strong>3000</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

For $\text{REM} = 0.10$:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$p(3000, 0.10, k)$</th>
<th>$k$</th>
<th>$p(3000, 0.10, k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>56</td>
<td>0.158642</td>
</tr>
<tr>
<td>7</td>
<td>4.94066e-324</td>
<td>63</td>
<td>0.000289045</td>
</tr>
<tr>
<td>47</td>
<td>0.00014182</td>
<td>64</td>
<td>5.79254e-05</td>
</tr>
<tr>
<td>54</td>
<td>0.169991</td>
<td>157</td>
<td>8.04833e-321</td>
</tr>
<tr>
<td><strong>55</strong></td>
<td><strong>0.182252</strong></td>
<td><strong>158</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

For $\text{REM} = 0.15$:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$p(3000, 0.15, k)$</th>
<th>$k$</th>
<th>$p(3000, 0.15, k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>39</td>
<td>0.207674</td>
</tr>
<tr>
<td>5</td>
<td>4.94066e-324</td>
<td>45</td>
<td>0.000310997</td>
</tr>
<tr>
<td>31</td>
<td>1.02128e-05</td>
<td>46</td>
<td>4.20583e-05</td>
</tr>
<tr>
<td>32</td>
<td>0.000164307</td>
<td>122</td>
<td>1.10127e-320</td>
</tr>
<tr>
<td>37</td>
<td>0.173966</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td><strong>38</strong></td>
<td><strong>0.22365</strong></td>
<td><strong>3000</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

5. Mutation Score

When we run $M$ mutants of program $P$ on some test data set $T$ and we find that $X$ mutants are killed, it is customary to view the ratio $\text{MS}(M, X) = \frac{X}{M}$ 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 data set 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 data set $T$. 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 Figure 3. 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 $NEC(N, REM)$. To compute the numerator, we introduce 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.

To compute $COV(N, K, X)$, we let $C_1, C_2, C_3, \ldots C_K$ be the equivalence classes of set $N$ and let $f_i$ be the Boolean function that takes value 1 if and only if $X$ has a non-empty intersection with class $C_i$. Then $COV(N, K, X)$ can be written as the expected value of the following random variable: $\sum_{i=1}^{K} f_i$. Therefore we have

$$COV(N, K, X) = E\left(\sum_{i=1}^{K} f_i\right)$$

$$= \sum_{i=1}^{K} p(f_i = 1)$$

$$= K \times (1 - p(f_1 = 0)),$$

where the last equation holds if $p(f_i = 0)$ is the same for all $i$. This happens if the classes are the same size and the events “set $X$ overlaps with class $C_i$”, for $1 \leq i \leq K$, are independent (which can be assumed if $N$ is very large). As for $p(f_1 = 0)$, it can be computed as $(\frac{K-1}{K})^X$ since each element of $X$ has probability $\frac{K-1}{K}$ to be outside class $C_1$ (again assuming independence and equal class size).

If we do not assume independence, and we still assume that all equivalence classes have the same size $(\frac{N}{K})$, then $p(f_1 = 0)$ can be evaluated as the product of the following probabilities, which represent the events: the first element of $X$ is not in $C_1$; the second element of $X$ is not in $C_1$; etc.:

$$\frac{N-K}{N},$$
$$\frac{N-K-1}{N},$$
$$\frac{N-K-2}{N},$$
$$\frac{N-K-3}{N},$$
Figure 3: Essential Mutation Score: $EMS(N, X)$
etc. Combining these expressions, we find the following closed formula for 
$COV(N, K, X)$:

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

The following table shows sample values for $COV(N, K, X)$ for $N = 75$, $K = 15$ and $X$ varying between 0 and 75; this quantity represents the number of distinct mutants that are estimated to be in $X$, given that $N$ is divided into 15 equivalence classes. As we can see from this table, $COV(N, K, X)$ converges very quickly towards $K$; it is nowhere near proportional to $X$ (by contrast with the traditional mutation score, $\frac{X}{M}$).

<table>
<thead>
<tr>
<th>$X$</th>
<th>$COV(75, 15, X)$</th>
<th>$X$</th>
<th>$COV(75, 15, X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
<td>40</td>
<td>14.7179</td>
</tr>
<tr>
<td>5</td>
<td>4.4814</td>
<td>45</td>
<td>14.8761</td>
</tr>
<tr>
<td>10</td>
<td>7.8214</td>
<td>50</td>
<td>14.9538</td>
</tr>
<tr>
<td>15</td>
<td>10.2534</td>
<td>55</td>
<td>14.9865</td>
</tr>
<tr>
<td>20</td>
<td>11.9766</td>
<td>60</td>
<td>14.9974</td>
</tr>
<tr>
<td>25</td>
<td>13.1586</td>
<td>65</td>
<td>14.9998</td>
</tr>
<tr>
<td>30</td>
<td>13.9382</td>
<td>70</td>
<td>15.0000</td>
</tr>
<tr>
<td>35</td>
<td>14.4281</td>
<td>75</td>
<td>15.0000</td>
</tr>
</tbody>
</table>

6. Minimal Mutant Sets

6.1. Extraction of Minimal Set

If we learn anything from section 4, 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:

```c
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:

```java
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};} }}
```

6.2. 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 \( d_1 = 1 \) 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.

An 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

\[
\begin{align*}
P(d_2 = 1) &= \frac{N - N/K}{N - 1} \\
P(d_2 = 2) &= \frac{N/K - 1}{N - 1} \cdot \frac{N - N/K}{N - 2} = \frac{N - N/K}{N - 1} \cdot \frac{N/K - 1}{N - 2} \\
&\quad \vdots \quad \vdots \\
P(d_2 = N/K) &= \frac{N - N/K}{N - 1} \cdot \frac{N/K - 1}{N - 2} \cdot \frac{N/K - (N/K - 1)}{N - N/K} \cdot \ldots 
\end{align*}
\]

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

As a simple illustration, consider the example discussed in section 4 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.

6.3. Estimating the Speedup

We briefly estimate the Big Oh performance of the two algorithms presented in section 6 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 \texttt{equivalent}(i,j) is the sum:

\[
1 + \ldots + 1 + 2 + \ldots + 2 + \ldots + (K - 1) + \ldots + (K - 1) + K.
\]

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:

\[ 2 \times \frac{N - H}{H} \]

For \( N = 3000 \) and \( H = 253 \), we find a speed up of 22.72, or 2272 %. Of course \( N = 3000 \) is atypically large; still, \( \frac{2 \times N - H}{H} \) is necessarily greater than 1, since \( N \geq H \).
6.4. 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. In a small-scale empirical experiment, we find that the number of inspections that we must conduct to cover all equivalence classes ranges from \( 0.65 \times H \) to \( 1.2 \times H \); hence if we let \( H' \) be \( 1.2 \times H \), we cover all equivalence classes; this bears further investigation.

7. Automation

We deploy compiler generation technology \([34]\) to automate the derivation of redundancy metrics for Java programs, which we use subsequently to drive 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 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.

7.1. 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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Entropy</th>
<th>Data Type</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 bit</td>
<td>int</td>
<td>32 bits</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>float</td>
<td>32 bits</td>
</tr>
<tr>
<td>char</td>
<td>16 bits</td>
<td>long</td>
<td>64 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>double</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

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 overridden by the user.

7.2. 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.

7.2.1. 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 \texttt{preassert(A)} , where \( A \) has the same syntax as in \texttt{assert(A)} , except that \texttt{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:

- \( \delta H(\text{true}) = 0 \).
- \( \delta H(\text{false}) = H(S) \).
- \( \delta H(E_1 == E_2) \), where \( E_1 \) and \( E_2 \) 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 \texttt{preassert(z==x+y)} , then we find:

\[
H(\sigma_I) = H(S) - \delta H(z = x + y) = 3 \times 32 - 32 = 64 \text{ bits}.
\]

- \( \delta H(E_1 > E_2) = \delta H(E_1 < E_2) = \delta H(E_1 \geq E_2) = \delta H(E_1 \leq E_2) = 1 \) bit, where \( E_1 \) and \( E_2 \) are two expressions of the same data type. Indeed,
all these comparisons exclude half of the possible values of \( E_1 \) and \( E_2 \), 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 \texttt{preassert(x>0)} then the entropy of the initial actual state is:

\[
H(\sigma_I) = H(S) - \delta H(x > 0) = 32 - 1 = 31 \text{ bits.}
\]

Indeed, we know the sign bit of \( x \), but we do not know its absolute value.

- \( \delta H(E_1 \neq E_2) = 0 \) bits. Indeed, the inequality excludes only one value for each expression (\( E_1 \) and \( E_2 \)), 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 \texttt{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.

- \( \delta H(A_1 \land A_2) = \delta H(A_1) + \delta H(A_2) \). Whereas this equation holds only if conditions \( A_1 \) and \( A_2 \) 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 \texttt{preassert(x==y && x==z)}. Then, we find:

\[
\delta H(x = y \land x = z) = \delta H(x = y) + \delta H(x = z) = 64 \text{ bits.}
\]

Whence, \( H(\sigma_I) = 96 - 64 = 32 \) bits, which reflects the fact that the entropy of the state is merely that of \( x \).

- \( \delta H(A_1 \lor A_2) = \min(\delta H(A_1), \delta H(A_2)) \). 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: \texttt{preassert(x=y || x==y && x==z)}. We have \( \delta H(x = y) = 32 \) bits, \( \delta H(x = y \land x = z) = 64 \) bits; hence \( \delta H(x = y \lor x = y \land x = z) = 32 \) bits, and

\[
H(\sigma_I) = 96 - 32 = 64 \text{ bits,}
\]

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 \texttt{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 great deal more analytical depth than we estimate we need for our purposes.
7.2.2. 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 catalogued variables.

To this effect we introduce a square Boolean matrix (called \( D \), for \textit{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 \textit{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 \textit{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 \textit{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 (\textit{true} on the diagonal, \textit{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 \textit{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 \{\textbf{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

\[
\text{if (cond) \{then-branch;}\text{ else \{else-branch;}\}}
\]

we derive the dependency matrix of its then-branch, say \( D_1 \), and the dependency matrix of its else-branch, say \( D_2 \), 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

\[
\text{if (cond) \{then-branch;\}}
\]

as a shorthand for

\[
\text{if (cond) \{then-branch;\} else \{skip;\}}
\]

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

\[
\text{while (cond) \{loop-body;\}}
\]

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.

Figures 5, 6 and 7 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.

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

Figure 5: Illustration of Dependency Matrix: Successive Assignments
<table>
<thead>
<tr>
<th>Statement</th>
<th>Dependency Matrix</th>
</tr>
</thead>
</table>
| \{int x, y, min, max; | \begin{tabular}{|c|c|c|c|c|}
| D | x | y | min | max \\
|---|---|---|---|---|
| x | T | F | F | F \\
| y | F | T | F | F \\
| min | F | F | T | F \\
| max | F | F | F | T \\
| \end{tabular} |
| if (x<y) \{min=x; | \begin{tabular}{|c|c|c|c|c|}
| D1 | x | y | min | max \\
|---|---|---|---|---|
| x | T | F | F | F \\
| y | F | T | F | F \\
| min | T | T | F | F \\
| max | F | F | F | T \\
| \end{tabular} |
| max=y; | \begin{tabular}{|c|c|c|c|c|}
| D1 | x | y | min | max \\
|---|---|---|---|---|
| x | T | F | F | F \\
| y | F | T | F | F \\
| min | T | T | F | F \\
| max | T | T | F | F \\
| \end{tabular} |
| } | \begin{tabular}{|c|c|c|c|c|}
| V1 | x | y | min | max \\
|---|---|---|---|---|
| T | T | F | F \\
| \end{tabular} |
| else \{max=y; | \begin{tabular}{|c|c|c|c|c|}
| D2 | x | y | min | max \\
|---|---|---|---|---|
| x | T | F | F | F \\
| y | F | T | F | F \\
| min | F | F | T | F \\
| max | T | T | F | F \\
| \end{tabular} |
| } | \begin{tabular}{|c|c|c|c|c|}
| V2 | x | y | min | max \\
|---|---|---|---|---|
| T | T | T | F \\
| \end{tabular} |

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

Figure 6: Illustration of Dependency Matrix: If-Then-Else Statement
<table>
<thead>
<tr>
<th>Statement</th>
<th>Dependency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>{int x, y, min;</td>
<td>$D$</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>min</td>
</tr>
<tr>
<td>if (x &lt; y) {min=x;</td>
<td>$D1$</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>min</td>
</tr>
<tr>
<td>}</td>
<td>$V1$</td>
</tr>
<tr>
<td></td>
<td>T</td>
</tr>
<tr>
<td>else {skip;</td>
<td>$D2$</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>min</td>
</tr>
<tr>
<td>}</td>
<td>$V2$</td>
</tr>
<tr>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

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

Figure 7: Illustration of Dependency Matrix: If-Then Statement
• Either select some special mutant generation policies, such as those that are implemented in common tools [15, 27], or those that have some research interest [2, 28, 23]; 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, because it 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 \( op_1, op_2, \ldots, op_k \), and we let \( M_1, M_2, \ldots, M_k \) be the number of mutants generated 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:

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

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 19 functions from the Apache Common Mathematics Library (http://apache.org/); each of these functions comes with a test data set and oracles in the form of assert statements. We use PiTest (http://pitest.org/) alongside Maven (http://maven.apache.org/) to generate mutants and test them for equivalent 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; 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.
<table>
<thead>
<tr>
<th>Operators</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1, Op2</td>
<td>0.05394231</td>
<td>0.08649908</td>
</tr>
<tr>
<td>Op1, Op3</td>
<td>0.01160714</td>
<td>0.07723429</td>
</tr>
<tr>
<td>Op1, Op4</td>
<td>0.27070767</td>
<td>0.48421531</td>
</tr>
<tr>
<td>Op1, Op5</td>
<td>0.06510417</td>
<td>0.27087339</td>
</tr>
<tr>
<td>Op1, Op6</td>
<td>0.07777778</td>
<td>0.25871068</td>
</tr>
<tr>
<td>Op1, Op7</td>
<td>-0.02901786</td>
<td>0.1259108</td>
</tr>
<tr>
<td>Op2, Op3</td>
<td>0.03292571</td>
<td>0.06985351</td>
</tr>
<tr>
<td>Op2, Op4</td>
<td>0.0168355</td>
<td>0.16486016</td>
</tr>
<tr>
<td>Op2, Op5</td>
<td>0.05475095</td>
<td>0.09854705</td>
</tr>
<tr>
<td>Op2, Op6</td>
<td>0.1729827</td>
<td>0.38869448</td>
</tr>
<tr>
<td>OP2, Op7</td>
<td>0.04274122</td>
<td>0.18378008</td>
</tr>
<tr>
<td>Op3, Op4</td>
<td>0.07184404</td>
<td>0.18378008</td>
</tr>
<tr>
<td>Op3, Op5</td>
<td>0.00677083</td>
<td>0.03692068</td>
</tr>
<tr>
<td>Op3, Op6</td>
<td>0.00416667</td>
<td>0.01666667</td>
</tr>
<tr>
<td>OP3, Op7</td>
<td>-0.02451475</td>
<td>0.12921422</td>
</tr>
<tr>
<td>Op4, Op5</td>
<td>0.40015212</td>
<td>0.44316312</td>
</tr>
<tr>
<td>Op4, Op6</td>
<td>0.22570767</td>
<td>0.33152983</td>
</tr>
<tr>
<td>OP4, Op7</td>
<td>0.23659948</td>
<td>0.32059105</td>
</tr>
<tr>
<td>Op5, Op6</td>
<td>0.0359375</td>
<td>0.09086614</td>
</tr>
<tr>
<td>Op5, Op7</td>
<td>0.00558036</td>
<td>0.16153478</td>
</tr>
<tr>
<td>Op6, Op7</td>
<td>0.00144433</td>
<td>0.16111891</td>
</tr>
<tr>
<td>Average</td>
<td>0.0825736</td>
<td>0.19450306</td>
</tr>
</tbody>
</table>

Table 2: Two Operators: Average Relative Residual of Computed vs Observed REM
<table>
<thead>
<tr>
<th>Operator Sets</th>
<th>average relative residual</th>
<th>Std dev</th>
<th>average absolute value of relative residual</th>
<th>Std dev</th>
<th>Correlation observed REM vs. Computed REM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1, Op2, Op3</td>
<td>0.04985883</td>
<td>0.06856532</td>
<td>0.049858829</td>
<td>0.82822952</td>
<td></td>
</tr>
<tr>
<td>Op1, Op2, Op5</td>
<td>0.05646967</td>
<td>0.06389703</td>
<td>0.06599348</td>
<td>0.80022051</td>
<td></td>
</tr>
<tr>
<td>Op1, Op2, Op6</td>
<td>0.08278866</td>
<td>0.10966936</td>
<td>0.08278664</td>
<td>0.899289</td>
<td></td>
</tr>
<tr>
<td>Op1, Op2, Op7</td>
<td>0.0849706</td>
<td>0.11365583</td>
<td>0.084970598</td>
<td>0.90049949</td>
<td></td>
</tr>
<tr>
<td>Op1, Op3, Op5</td>
<td>0.01083024</td>
<td>0.03603681</td>
<td>0.017632962</td>
<td>0.84750228</td>
<td></td>
</tr>
<tr>
<td>Op1, Op3, Op6</td>
<td>0.00411255</td>
<td>0.01060487</td>
<td>0.004112554</td>
<td>0.99867987</td>
<td></td>
</tr>
<tr>
<td>Op1, Op3, Op7</td>
<td>0.01020408</td>
<td>0.03162632</td>
<td>0.013605442</td>
<td>0.905375</td>
<td></td>
</tr>
<tr>
<td>Op1, Op5, Op6</td>
<td>0.0042517</td>
<td>0.01590841</td>
<td>0.004251701</td>
<td>0.97383186</td>
<td></td>
</tr>
<tr>
<td>Op1, Op5, Op7</td>
<td>0.0212585</td>
<td>0.03592082</td>
<td>0.021258503</td>
<td>0.98938827</td>
<td></td>
</tr>
<tr>
<td>Op1, Op6, Op7</td>
<td>0.002501</td>
<td>0.00935789</td>
<td>0.002501</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Op2, Op3, Op5</td>
<td>0.03267884</td>
<td>0.04894163</td>
<td>0.046902709</td>
<td>0.80495576</td>
<td></td>
</tr>
<tr>
<td>Op2, Op3, Op6</td>
<td>0.04776747</td>
<td>0.07640335</td>
<td>0.049354776</td>
<td>0.82547084</td>
<td></td>
</tr>
<tr>
<td>Op2, Op3, Op7</td>
<td>0.05452182</td>
<td>0.08079703</td>
<td>0.054521817</td>
<td>0.81781778</td>
<td></td>
</tr>
<tr>
<td>averages</td>
<td>0.03555492</td>
<td>0.03828869</td>
<td>0.038288695</td>
<td>0.89187867</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Three Operators: Average Relative Residual of Computed vs Observed REM

<table>
<thead>
<tr>
<th>Operator Sets</th>
<th>average relative residual</th>
<th>average absolute value of relative residual</th>
<th>std Dev average rel. res</th>
<th>std Dev abs. val. of avg. rel. res</th>
<th>Correlation observed REM vs. Computed REM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1, Op2, Op3, Op4</td>
<td>0.08752757</td>
<td>0.10965401</td>
<td>0.10020956</td>
<td>0.0730987</td>
<td>0.5418512</td>
</tr>
<tr>
<td>Op1, Op2, Op3, Op5</td>
<td>0.05289048</td>
<td>0.06063621</td>
<td>0.0618701</td>
<td>0.05375816</td>
<td>0.80390398</td>
</tr>
<tr>
<td>Op1, Op2, Op3, Op6</td>
<td>0.07477069</td>
<td>0.07477069</td>
<td>0.07873449</td>
<td>0.07873449</td>
<td>0.86348068</td>
</tr>
<tr>
<td>Op1, Op2, Op3, Op7</td>
<td>0.06373091</td>
<td>0.06373091</td>
<td>0.07169284</td>
<td>0.07169284</td>
<td>0.83527074</td>
</tr>
<tr>
<td>averages</td>
<td>0.06973</td>
<td>0.071197955</td>
<td>0.078126748</td>
<td>0.069321</td>
<td>0.0761127</td>
</tr>
</tbody>
</table>

Table 4: Four Operators: Average Relative Residual of Computed vs Observed REM
Tables 2, 3 and 4 show the results. These appear to bear out the proposed formula.

These tables refer to the following PiTest 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.

It appears that Op4, which is the Conditional\_boundary\_mutator introduces a higher level of residual error than the other operators. This may be because it violates our assumption that all $M_i$'s are of equal size; which may be because it affects a smaller (or larger) number of statements than other operators. An interesting remedy to this may be to have the system that computes redundancy metrics keep track of the density of statements affected by each operator, and use this information to derive different coefficients in the formula of $REM$ as a function of $REM_i$'s; each operator will have a coefficient that is proportional to the number of statements in the base program that it affects; this is a subject of further research.

9. 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.

9.1. 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 [11], 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.

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

Table 5: Functions and their Meaning

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 4 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 data set in terms of the distinct mutants that it kills. In section 5 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. In addition to the formula proposed in section 5, we are also considering the following formula:

\[
\frac{\text{COV}(X, \text{NEC}(N, \text{REM}))}{\text{COV}(N, \text{NEC}(N, \text{REM}))}
\]

Empirical validation of these formulas is under investigation; in particular, we are considering to check these two formulas against the criteria set forth in [17].

- Extract a minimal set of mutants that is free of redundancy from a set of possibly redundant mutants. In section 6 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 (\( \text{NEC}(N, \text{REM}) \)) then we can pick one mutant per equivalent class at little cost.

To derive the \( \text{REM} \) 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 \( \text{REM} \). 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 \( \text{REM} \). 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 \( \text{REM} \). This system is currently being evolved to enhance its precision and scope.

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; 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 [14]. 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 7).

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 [5, 6], 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 cataloguing 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 [8, 9]. 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 ($SR_1$, $SR_F$, $FR$, $NI$), 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.

9.3. 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 $COV(N, K, X)$ is borne out in practice: Given a set of size $N$ divided into $K$ equivalence classes and a subset thereof of size $X$, how many equivalence classes of the set overlap with $X$. We test this hypothesis with and without the assumption that equivalence classes have the same size.

• 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.

References


