Redundancy: The Mutants' Elixir of Immortality

Amani Ayad, NJIT, Newark, NJ USA

Imen Marsit and Nazih Mohamed Omri, University of Monastir, Tunisia

JiMeng Loh and Ali Mili, NJIT, Newark, NJ USA

***Abstract*—<u>Context</u>.** Equivalent mutants are a major nuisance in the practice of mutation testing, because they introduce a significant amount of bias and uncertainty in the analysis of test results. Yet, despite several decades of research, the identification of equivalent mutants remains a tedious, inefficient, ineffective and error prone process.

<u>Objective</u>. Our objective is two-fold: First, to show that for most practical applications it is not necessary to identify equivalent mutants individually, rather it is sufficient to estimate their number. Second, to show that it is possible to estimate the number of equivalent mutants that a mutation experiment is likely to generate, by analyzing the base program and the mutation operators.

<u>Method</u>. We argue that the ratio of equivalent mutants that a program is prone to generate depends on the amount of redundancy of the program, and we introduce metrics that quantify various dimensions of redundancy in a program. Then we use empirical methods to show that our redundancy metrics are indeed correlated to the ratio of equivalent mutants, and that the latter can be estimated from the former by means of a regression model.

<u>Results</u>. We provide a regression formula for the ratio of equivalent mutants generated from a program, using the redundancy metrics of the program as independent variables. While this regression model depends on the mutation generation policy, we also study how to produce a generic estimation model that takes into account the set of mutation operators.

<u>Conclusion</u>. Trying to identify equivalent mutants by analyzing them individually is very inefficient, but also unnecessary. Estimating the number of equivalent mutants can be carried out efficiently, reliably, and is often sufficient for most applications.

Keywords—redundancy; equivalent mutants; software metrics; mutant survival ratio.

I. EQUIVALENT MUTANTS

Mutation is used in software testing to analyze the effectiveness of test data or to simulate faults in programs, and is meaningful only to the extent that the mutants are semantically distinct from the base program [1] [2] [3] [4]. But in practice mutants may sometimes be semantically equivalent to the base program while being syntactically distinct from it [5] [6] [7] [8] [9] [10] [11]. The issue of equivalent mutants has mobilized the attention of researchers for a long time.

Given a base program P and a mutant M, the problem of determining whether M is equivalent to P is known to be undecidable [12]. If we encounter test data for which P and M produce different outcomes, then we can conclude that M is not equivalent to P, and we say that we have *killed* mutant M; but no amount of testing can prove that M is equivalent to P. In the absence of a systematic/ algorithmic procedure to determine equivalence, researchers have resorted to heuristic approaches. In [7], Gruen et al. identify four sources of mutant equivalence: the mutation is applied to dead code; the mutation alters the performance of the code but not its function; the mutation alters internal states but not the output; and the mutation cannot be sensitized. This classification is interesting, but it is neither complete nor orthogonal, and offers only limited insights into the task of identifying equivalent mutants. In [13] Offutt and Pan argue that the problem of detecting equivalent mutants is a special case of a more general problem, called the *feasible path problem*; also they use a constraint-based technique to automatically detect equivalent mutants and infeasible paths. Experimentation with their tool shows that they can detect nearly half of the equivalent mutants on a small sample of base programs. Program slicing techniques are proposed in [14] and subsequently used in [15] [16] as a means to assist in identifying equivalent mutants. In [17], Ellims et al. propose to help identify potentially equivalent mutants by analyzing the execution profiles of the mutant and the base program. Howden [18] proposes to detect equivalent mutants by checking that a mutation preserves local states, and Schuler et al. [19] propose to detect equivalent mutants by testing automatically generated invariant assertions produced by Daikon [20]; both the Howden approach and the Daikon approach rely on local conditions to determine equivalence, hence they are prone to generate sufficient but unnecessary conditions of equivalence; a program P and its mutant M may well have different local states but still produce the same overall behavior; the only way to generate necessary and sufficient conditions of equivalence between a base program and a mutant is to analyze the programs in full (vs analyze them locally). In [21], Nica and Wotawa discuss how to detect equivalent mutants by using constraints that specify the conditions under which a test datum can kill the mutant; these constraints are submitted to a constraint solver, and the mutant is considered equivalent whenever the solver fails to find a solution. This approach is as good as the generated constraints, and because the constraints are based on a static analysis of the base program and the mutant, this solution has severe effectiveness and scalability limitations. In [22] Carvalho et al. report

on empirical experiments in which they collect information on the average ratio of equivalent mutants generated by mutation operators that focus on preprocessor directives; this experiment involves a diverse set of base programs, and is meant to reflect properties of the selected mutation operators, rather than the programs per se. In [23] Kintis et al. put forth the criterion of *Trivial Compiler Equivalence* (TCE) as a "simple, fast and readily applicable technique" for identifying equivalent mutants and duplicate mutants in C and Java programs. They test their technique against a benchmark ground truth suite (of known equivalent mutants) and find that they detect almost half of all equivalent mutants in Java programs.

It is fair to argue that despite several years of research, the problem of automatically and efficiently detecting equivalent mutants for programs of arbitrary size and complexity remains an open challenge. In this paper we adopt a totally orthogonal approach, based on the following premises:

- For most practical applications of mutation testing, it is not necessary to identify equivalent mutants individually; rather it is sufficient to know their number. If we generate 100 mutants and we want to use them to assess the quality of a test data set, then it is sufficient to know how many of them are equivalent: if we know that 20 of them are equivalent, then the test data will be judged by how many of the remaining 80 mutants it kills.
- Even when it is important to identify individually those mutants that are equivalent to the base, knowing their number is helpful: as we kill more and more non-equivalent mutants, the likelihood that the surviving mutants are equivalent rises as we approach the estimated number of equivalent mutants.
- For a given mutant generation policy, it is possible to estimate the ratio (over the total number of generated mutants) of equivalent mutants that a program is prone to produce, by static analysis of the program. We refer to this parameter as the *ratio of equivalent mutants* (*REM*, for short); because mutants that are found to be distinct from the base program are said to be killed, we may also refer to this parameter as the *survival rate* of the program.

In section II we argue that, for a given mutant generation policy, what determines the REM of a program P is the amount of redundancy of program P; based on this conjecture, we claim that if we can quantify the redundancy of a program, we can find statistical relations between the redundancy metrics of a program and its REM. In section III we present a number of entropy-based measures of program redundancy, and put forth analytical arguments to the effect that these are reliable indicators of the preponderance of equivalent mutants in a program. In section IV we report on an empirical study that bears out our analysis; specifically, we find significant correlations between the redundancy metrics and the REM's of sample benchmark programs, and we derive a regression model that has the REM as dependent variable and the redundancy metrics as independent variables.

II. THE KEY TO IMMORTALITY

The agenda of this paper is not to identify and isolate equivalent mutants, but instead to estimate their number. To estimate the number of equivalent mutants, we consider question RQ3 raised by Yao et al. in [5]: What are the causes of mutant equivalence? For a given mutant generation policy, this question can be reformulated more precisely as: what attribute of a program makes it likely to generate more equivalent mutants?

To answer this question, we consider that the attribute that makes a program prone to generate equivalent mutants is the same attribute that makes a program fault tolerant: indeed, a fault tolerant program is a program that continues to deliver correct behavior (e.g. by maintaining equivalent functionality) despite the presence and sensitization of faults (e.g. faults introduced by mutation operators). We know what feature causes a program to be fault tolerant: redundancy. Hence if only we could find a way to quantify the redundancy of a program, we could conceivably relate it to the rate of equivalent mutants generated from that program.

But the ratio of equivalent mutants of a program does not depend exclusively on the program, it also depends on the mutation generation policy; in section V, we discuss the impact of the mutation generation policy on the REM; in the meantime, we assume that we have a default/ fixed mutation generation policy, and we focus on the impact of the program's redundancy metrics.

Because our measures of redundancy use Shannon's entropy function [24], we briefly introduce some definitions, notations and properties related to this function, referring the interested reader to more detailed sources [25]. Given a random variable X that takes its values in a finite set, which for convenience we also designate by X, the *entropy* of X is the function denoted by H(X) and defined by:

$$H(X) = -\sum_{x_i \in X} p(x_i) \log\big(p(x_i)\big),$$

where $p(x_i)$ is the probability of the event $X = x_i$. Intuitively, this function measures (in bits) the uncertainty pertaining to the outcome of X, and takes its maximum value $H(X) = log(N)$ when the probability distribution is uniform, where N is the cardinality of X.

We let X and Y be the two random variables; the *conditional entropy* of X *given* Y is denoted by $H(X|Y)$ and defined by:

$$H(X|Y) = H(X,Y) - H(Y),$$

where $H(X,Y)$ is the joint entropy of the aggregate random variable (X,Y). The conditional entropy of X given Y reflects the uncertainty we have about the outcome of X if we know the outcome of Y. If Y is a function of X, then the joint entropy $H(X,Y)$ is equal to $H(X)$, hence the conditional entropy of X given Y can simply be written as:

$$H(X|Y) = H(X) - H(Y).$$

All entropies (absolute and conditional) take non-negative values. Also, regardless of whether Y depends on X or not, the conditional entropy of X given Y is less than or equal to the entropy of X (the uncertainty on X can only decrease if we know Y). Hence for all X and Y, we have the inequality:

$$0 \leq \frac{H(X|Y)}{H(X)} \leq 1.0.$$

III. ANALYTICAL STUDY

In this section, we review a number of entropy-based redundancy metrics of a program, reflecting a number of dimensions of redundancy. For each metric, we discuss, in turn:

- How we define this metric.
- Why we feel that this metric has an impact on the rate of equivalent mutants.
- How we compute this metric in practice (by hand for now).

Because our ultimate goal is to derive a formula for the REM of the program as a function of its redundancy metrics, and because the REM is a fraction that ranges between 0 and 1, we resolve to let all our redundancy metrics be defined in such a way that they range between 0 and 1.

A. *State Redundancy*

***What is State Redundancy*?** State redundancy is the gap between the declared state of the program and its actual state. Indeed, it is very common for programmers to declare much more space to store their data than they actually need, not by any fault of theirs, but due to the limited vocabulary of programming languages. An extreme example of state redundancy is the case where we declare an integer variable (entropy: 32 bits) to store a Boolean variable (entropy: 1 bit). More common and less extreme examples include: we declare an integer variable (entropy: 32 bits) to store the age of a person (ranging realistically from 0 to 128, to be optimistic, entropy: 7 bits); we declare an integer variable to represent a calendar year (ranging realistically from 2018 to 2100, entropy: 6.38 bits).

Definition: State Redundancy. Let P be a program, let S be the random variable that takes values in its declared state space and σ be the random variable that takes values in its actual state space. The *state redundancy* of Program P is defined as:

$$\frac{H(S) - H(\sigma)}{H(S)}$$

Typically, the declared state space of a program remains unchanged through the execution of the program, but the actual state space (i.e. the range of values that program variables may take) grows smaller and smaller as execution proceeds, because the program creates more and more dependencies between its variables with each assignment. Hence we are interested in defining two versions of state redundancy: one pertaining to the initial state, and one pertaining to the final state.

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

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

where σ_I and σ_F are (respectively) the initial state and the final state of the program, and S is its declared state. Since the entropy of the final state is typically smaller than that of the initial state (because the program builds relations between its variables as it proceeds in its execution), the final state redundancy is usually larger than the initial state redundancy.

Why is state redundancy correlated to survival rate? State redundancy measures the volume of data bits that are accessible to the program (and its mutants) but are not part of the actual state space. Any assignment to/ modification of these extra bits of information does not alter the state of the program. Consider the extreme case of using an integer to store a Boolean variable *b*, where 0 represents false and 1 represents true. If the base program tests the condition

```
P:  { if (b==0) {…}  else {…} }
```

and the mutant tests the condition

```
M: { if (5*b==0) {…}  else {…} }
```

then M would be equivalent to P.

How do we compute state redundancy? We must compute the entropies of the declared state space ($H(S)$), the entropy of the actual initial state ($H(\sigma_I)$) and the entropy of the actual final state ($H(\sigma_F)$). For the entropy of the declared state, we simply add the entropies of the individual variable declarations, according to the following table (for Java):

Data Type	Entropy (bits)
Boolean	1
Byte	8
Char, short	16
Int, float	32
Long, double	64

Table 1. Entropies of Basic Variable Declarations

For the entropy of the initial state, we consider the state of the program variables once all the relevant data has been received (through read statements, or through parameter passing, etc.) and we look for any information we may have on the incoming data (range of some variables, relations between variables, assert statements specifying the precondition, etc.); the default option being the absence of any condition. When we automate the calculation of redundancy metrics, we will rely exclusively on assert statements that may be included in the program to specify the precondition.

For the entropy of the final state, we take into account all the dependencies that the program may create through its execution. When we automate the calculation of redundancy metrics, we may rely on any assert statement that the programmer may have included to specify the program's post-condition; we may also keep track of functional dependencies between program variables by monitoring what variables appear on each side of assignment statements. As an illustration, we consider the following simple example:

```
public void example(int x, int y)
   {assert (1<=x && x<=128 && y>=0);
    long z = reader.nextInt();
    //  initial state
    Z = x+y; //  final state
   }
```

We find:

- $H(S) = 32 + 32 + 64 = 128\ bits.$
  Entropies of x, y, z, respectively.
- $H(\sigma_I) = 10 + 31 + 64 = 105\ bits$
  Entropy of x is 10, because of its range; entropy of y is 31 bits because half the range of int is excluded.
- $H(\sigma_F) = 10 + 31 = 41\ bits.$
  Entropy of z is excluded because z is now determined by x and y.

Hence

$$SR_I = \frac{128 - 105}{128} = 0.18,$$

$$SR_F = \frac{128-41}{128} = 0.68.$$

B. Non Injectivity

What is Non Injectivity. A major source of program redundancy is the non-injectivity of program functions. An injective function is a function whose value changes whenever its argument does; and a function is all the more non-injective that it maps several distinct arguments into the same image. A sorting routine applied to an array of size N, for example, maps N! different input arrays (corresponding to N! permutations of N distinct elements) onto a single output array (the sorted permutation of the elements). To introduce non-injectivity, we consider the function that the program defines on its state space from initial states to final states. A natural way to define non-injectivity is to let it be the conditional entropy of the initial state given the final state: if we know the final state, how much uncertainty do we have about the initial state? Since we want all our metrics to be fractions between 0 and 1, we normalize this conditional entropy to the entropy of the initial state. Hence we write:

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

Since the final state is a function of the initial state, the numerator can be simplified as $H(\sigma_I) - H(\sigma_F)$. Hence:

Definition: Non Injectivity. Let P be a program, and let σ_I and σ_F be the random variables that represent, respectively its initial state and final state. Then the *non-injectivity* of program P is denoted by NI and defined by:

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

Why is non-injectivity correlated to survival rate? Of course, non-injectivity is a great contributor to generating equivalent mutants, since it increases the odds that the state produced by the mutation be mapped to the same final state as the state produced by the base program.

How do we compute non-injectivity? We have already discussed how to compute the entropies of the initial state and final state of the program; these can be used readily to compute non-injectivity. For illustration, we consider the sample program above, and we find its non-injectivity as:

$$NI = \frac{105-41}{105} = 0.61\,.$$

C. Functional Redundancy

What is Functional Redundancy? A program can be modeled as a function from initial states to final states, as we have done in sections A and B above, but can also be modeled as a function from an input space to an output space. To this effect we let X be the random variable that represents the aggregate of input data that the program receives (through parameter passing, read statements, global variables, etc.), and Y the aggregate of output data that the program delivers (through parameter passing, write statements, return statements, global variables, etc.).

Definition: Functional Redundancy. Let P be a program, and let X be the random variable that ranges over the aggregate of input data received by P and Y the random variable that ranges over the aggregate of output data delivered by P. Then the *functional redundancy* of program P is denoted by FR and defined by:

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

Why is Functional Redundancy Related to Survival Rate? Functional redundancy is actually an extension of non-injectivity, in the sense that it reflects not only how initial states are mapped to final states, but also how initial states are affected by input data and how final states are projected onto output data. Consider for example a program that computes the median of an array by first

sorting the array, which causes an increase in redundancy due to the drop in entropy, then returning the element stored in the middle of the array, causing a further massive drop in entropy by mapping a whole array onto a single cell. All this drop in entropy creates opportunities for the difference between a base program and a mutant to be erased, leading to mutant equivalence.

How do we compute Functional Redundancy? To compute the entropy of X, we analyze all the sources of input data into the program, including data that is passed in through parameter passing, global variables, read statements, etc. Unlike the calculation of the entropy of the initial state, the calculation of the entropy of X does not include internal variables, and does not capture initializations. To compute the entropy of Y, we analyze all the channels by which the program delivers output data, including data that is returned through parameters, written to output channels, or delivered through return statements. For illustration, we consider the following program:

```
public void example(int u, int v)
   {assert (v>=0);
    int z = 0;
    while (v!=0) {z=z+u; v=v-1;}
    return z;
   }
```

We compute the entropies of the input space and output space:

- $H(X) = 32 + 31 = 63\ bits.$

  Entropy of u, plus entropy of v (which ranges over half of the range of integers).
- $H(Y) = 32\ bits.$

  Entropy of z.

Hence,

$$FR = \frac{32}{63} = 0.51\,.$$

D. Non Determinacy

What is Non Determinacy? In all the mutation research that we have surveyed, mutation equivalence is equated with equivalent behavior between a base program and a mutant; but we have not found a precise definition of what is meant by *behavior*, nor what is meant by *equivalent* behavior. We argue that the concept of *equivalent behavior* is not precisely defined: we consider the following three programs,

```
P1: {int x,y,z; z=x; x=y; y=z;}
P2: {int x,y,z; z=y; y=x; x=z;}
P3: {int x,y,z; x=x+y;y=x-y;x=x-y;}
```

We ask the question: are these programs equivalent? The answer to this question depends on how we interpret the role of variables x, y, and z in these programs. If we interpret these as programs on the space defined by all three variables, then we find that they are distinct, since they assign different values to variable z (x for P1, y for P2, and z for P3). But if we consider that these are actually programs on the space defined by variables x and y, and that z is a mere auxiliary variable, then the three programs may be considered equivalent, since they all perform the same function (swap x and y) on their common space (formed by x, y). Consider a slight variation on these programs:

```
Q1: {int x,y;{int z; z=x; x=y; y=z;}}
Q2: {int x,y;{int z; z=y; y=x; x=z;}}
Q3: {int x,y; x=x+y;y=x-y;x=x-y;}
```

Here it is clear(er) that all three programs are defined on the space formed by variables x and y; and it may be easier to be persuaded that these programs are equivalent.

Rather than making this a discussion about the space of the programs, we wish to turn it into a discussion about the test oracle that we are using to check equivalence between the programs (or in our case, between a base program and its mutants). In the example

above, if we let xP, yP, zP be the final values of x, y, z by the base program and xM, yM, zM the final values of x, y, z by the mutant, then oracles we can check include:

```
O1:{return xP==xM && yP==yM && zP==zM;}
O2:{return xP==xM && yP==yM;}
```

Oracle O1 will find that P1, P2 and P3 are not equivalent, whereas oracle O2 will find them equivalent. The difference between O1 and O2 is their degree of non-determinacy; this is the attribute we wish to quantify.

Whereas all the metrics we have studied so far apply to the base program, this metric applies to the oracle that is being used to test equivalence between the base program and a mutant. We want this metric to reflect the degree of latitude that we allow mutants to differ from the base program and still be considered equivalent. To this effect, we let σ^P be the final state produced by the base program for a given input, and we let σ^M be the final state produced by a mutant for the same input. We view the oracle that tests for equivalence between the base program and the mutant as a binary relation between σ^P and σ^M. We can quantify the non-determinacy of this relation by the conditional entropy $H(\sigma^M|\sigma^P)$: Intuitively, this represents the amount of uncertainty (or: the amount of latitude) we have about (or: we allow for) σ^M if we know σ^P. Since we want our metric to be a fraction between 0 and 1, we divide it by the entropy of σ^M. Hence the following definition.

Definition: Non Determinacy. Let O be the oracle that we use to test the equivalence between a base program P and a mutant M, and let σ^P and σ^M be, respectively, the random variables that represent the final states generated by P and M for a given initial state. The *non-determinacy* of oracle O is denoted by ND and defined by:

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

Why is Non Determinacy correlated with survival rate? Of course, the weaker the oracle of equivalence, the more mutants pass the equivalence test, the higher the ratio of equivalent mutants.

How do we compute non determinacy? All equivalence oracles define equivalence relations on the space of the program, and $H(\sigma^M|\sigma^P)$ represents the entropy of the resulting equivalence classes. As for $H(\sigma^M)$, it represents the entropy of the whole space of the program. For illustration, let the space of the program be defined by three integer variables, say x, y, z. Then $H(\sigma^M) = 96\ bits$. As for $H(\sigma^M|\sigma^P)$, it will depend on how the oracle is defined, as it represents the entropy of the resulting equivalence classes. We show a few examples below:

O#	Oracle	$H(\sigma^M\|\sigma^P)$	ND
O1	xP==xM&&yP==yM&&zP==zM	0 bits	0.0
O2	xP==xM&&yP==yM	32 bits	0.33
O3	xP==xM&&zP==zM	32 bits	0.33
O4	yP==yM&&zP==zM	32 bits	0.33
O5	xP==yM	64 bits	0.66
O6	yP==yM	64 bits	0.66
O7	zP==zM	64 bits	0.66
O8	true	96 bits	1.00

Table 2. Non Determinacy of Sample Oracles

Explanation: Oracle O1 is deterministic (assuming the space is made up of x, y, z only), hence its equivalence classes are of size 1; the corresponding conditional entropy is zero, and so is ND. Oracles O2, O3, O4 check for two variables but leave one variable unchecked, leading to a conditional entropy of 32 bits and a non-determinacy of 0.33 (32/96). Oracles O5, O6, O7 check for one variable but leave two variables unchecked, leading to a conditional entropy of 64 bits and a non-determinacy of 0.66 (64/96). Oracle O8 returns true for any σ^M. Hence knowing that a mutant passes this test does not inform us on any of xM, yM, nor zM. Total uncertainty is 96, hence ND=1.

Imagine now, for the sake of illustration, that we have a single integer variable, say x. Then we can define the following oracles, in the order of decreasing strength, and increasing non-determinacy.

O#	Oracle	$H(\sigma^M \mid \sigma^P)$	ND
O1	`xP==xM`	0 bits	0.000
O2	`xP % 4096 == xM % 4096`	20 bits	0.625
O3	`xP % 1024 == xM % 1024`	22 bits	0.687
O4	`xP % 64 == xM % 64`	26 bits	0.812
O5	`xP % 16 == xM % 16`	28 bits	0.875
O6	`xP % 4 == xM % 4`	30 bits	0.937
O7	`xP % 2 == xM % 2`	31 bits	0.969
O8	`True`	32 bits	1.000

Table 3. Non Determinacy of Sample Integer Oracles

The interpretation of rows O1 and O8 is the same as the table above. For O7, for example, consider that if we know that xM satisfies oracle O7, then we know the rightmost bit of xM, but we do not know anything about the remaining 31 bits; hence the conditional entropy is 31 bits, and the non-determinacy is 0.969, which is 31/32. Oracle O2 informs us about the 12 rightmost bits of xM hence leaves us uncertain about the remaining 20 bits. The non-determinacy of the other oracles can be interpreted likewise.

IV. EMPIRICAL STUDY

A. *Experimental Conditions*

In order to validate our conjecture, to the effect that the survival rate of mutants generated from a program P depends on the redundancy metrics of the program and the non-determinacy of the oracle that is used to determine equivalence, we consider a number of sample programs, compute their redundancy metrics then record the ratio of equivalent mutants that they produce under controlled experimental conditions, for a fixed mutant generation policy. Our hope is to reveal significant statistical relationships between the metrics (as independent variables) and the ratio of equivalent mutants (as a dependent variable). Because we currently compute the redundancy metrics by hand, we limit ourselves to programs that are relatively small in size.

We consider functions taken from the *Apache Common Mathematics Library* (http://apache.org/); each function comes with a test data file. The test data file includes not only the test data proper, but also a test oracle in the form of assert statements, one for each input datum. Our sample includes 19 programs.

We use PITEST (http://pitest.org/), in conjunction with maven (http://maven.apache.org/) to generate mutants of each program and test them for possible equivalence with the base program. The mutation operators that we have chosen include the following:

- Op1: Increments_mutator.
- Op2: Void_method_call_mutator,
- Op3: Return_vals_mutator,
- Op4: Math_mutator,
- Op5: Negate_conditionals_mutator,
- Op6: Invert_negs_mutator,
- Op7: Conditionals_boundary_mutator.

Of course, we realize that the ratio of equivalent mutants may depend on the choice of mutation operators; but because the focus of our study is to analyze how the ratio of equivalent mutants depends on the base program and the oracle used for determining equivalence, we use a fixed set of mutants for the time being, and postpone the analysis of the impact of mutant operators to section V.

When we run a mutant M on a test data set T and we find that its behavior is equivalent (per the selected oracle) to that of the base program P, we may not conclude that M is equivalent to P unless we have some assurance that T is sufficiently thorough. In practice, it is impossible to ascertain the thoroughness of T short of letting T be all the input space of the program, which is clearly impractical. As an alternative, we mandate that in all our experiments, line coverage of P and M through their execution on test data T equals or exceeds 90%. This measure also reduces the risk of having mutants that are equivalent to the base program by virtue of the mutation being applied to dead code.

In order to analyze the impact of the non-determinacy of the equivalence oracle on the ratio of equivalent mutants, we revisit the source code of PITEST to control the oracle that it uses. As we discuss above, the test file that comes in the Apache Common Mathematics Library includes an oracle that takes the form of assert statements in Java (one for each test datum). These statements have the form:

```
Assert.assertEqual(yP,M(x))
```

where x is the current test datum, yP is the output delivered by the base program P for input x, and M(x) is the output delivered by mutant M for input x. For this oracle, we record the non-determinacy (ND) as being zero. To test the mutant for other oracles, we replace

```
AssertEqual(yP,M(x))
```

with

```
AssertEquivalent(yP,M(x))
```

for various instances of equivalence relations. If the space of the base program includes several variables, we use some of the oracles listed in Table 2, and we take note of their non-determinacy. Also, if `yP` and `M(x)` are integer variables, then we use some of the equivalence relations discussed in Table 3, and we take note of their non-determinacy.

B. *Raw Data*

The raw data that results from this experiment is displayed in Table 4. This table also shows (in the last row) the correlations between the redundancy metrics and the ratio of equivalent mutants, as defined in our experiment.

Function	***LOC***	***Oracle***	***SRi***	***SRf***	***FR***	***NI***	***ND***	***COV***	***S/T***	***REM***
gcd	56	Equal	0.889	0.943	0.50	0.49	0	90%	16/103	0.155
		Eq%2					0.984		22/103	0.214
		Eq%4					0.953		19/103	0.184
		Eq%16					0.938		16/103	0.155
muland check	42	Equal	0.862	0.931	0.50	0.43	0	95%	6/43	0.14
		Eq%2					0.984		6/43	0.14
fraction	68	Equal	0.88	0.961	0.33	0.66	0	96%	22/95	0.234
		dEq					0.5		23/95	0.242
		dEq%2					0.84		26/95	0.273
reduced fraction	26	Equal	0.86	0.98	1.00	0.77	0	96%	17/46	0.37
		dEq					0.5		19/46	0.413
erfInv	88	Equal	0.62	0.63	1.00	0.03	0	99%	9/126	0.071
ebeDiv	20	Equal	0.898	0.90	0.50	0.10	0	97%	1/13	0.077
getDist	19	Equal	0.89	0.90	0.50	0.10	0	97%	1/17	0.059
ArRealVec	12	Equal	0.901	0.951	0.90	0.48	0	97%	2/10	0.020
ToBlocks	42	Equal	0.896	0.904	1.00	0.08	0	95%	3/31	0.097
getRowM	27	Equal	0.877	0.949	0.98	0.59	0	95%	7/23	0.304
orthogM	87	Equal	0.908	0.933	0.75	0.28	0	100%	20/151	0.132
Equals	31	Equal	0.852	0.935	0.20	0.56	0	90%	6/21	0.286
Density	18	Equal	0.883	0.957	0.25	0.23	0	95%	5/30	0.167
Abs()	20	Equal	0.896	0.931	0.50	0.33	0	96%	2/20	0.10
Pow	55	Equal	0.51	0.61	0.67	0.20	0	97%	6/52	0.115
setSeed	17	Equal	0.805	0.905	1.00	0.51	0	100%	4/16	0.25
Asinh	17	Equal	0.898	0.914	1.00	0.15	0	97%	13/82	0.159
Atan	143	Equal	0.90	0.92	0.40	0.08	0	97%	14/136	0.103
nextPrime	35	Equal	0.793	0.896	0.40	0.5	0	94%	3/58	0.05
		Eq%2					0.96		34/58	0.58
Correlations vs REM			0.055	0.308	0.1401	0.65	0.431			

Table 4: Raw Data, REM vs Redundancy Metrics

The oracles labeled $Equal$ represent strict equality, i.e. $\sigma^P = \sigma^M$; their non-determinacy is $ND = 0$. The oracles labeled *Eq%N*, for varying values of *N*, apply when the output is an integer, say X, and represent the condition $X^P mod\ N = X^M\ mod\ N$; their non-determinacy is $ND = 1 - \frac{log_2(N)}{H(X)}$. The column labeled *S/T* represents the ratio of S (number of equivalent mutants) over T (total number of generated mutants).

C. Statistical Analysis

Table 5 shows a matrix of scatter plots between each pair of the metrics and the REM. For example, in the bottom row of scatter plots, the y-axis is the REM (S/T), and the x-axis are, going from left to right, for metrics SRI, SRF, FR, NI and ND. On inspection of the plots, each of the metrics seems to show some positive correlation with S/T, the strongest being NI. We note that the ND values are confined to 0 or values very close to 1. In our models below, we assume a linear relationship, even though there is no data with moderate values of ND. Finally, we also note that SRI and SRF appear to be highly correlated. Inclusion of both variables in a model can result in unstable estimates. However, it turns out (see below) that both variables are not included in the final model.

Since the response, REM, is a proportion, we use a logistic linear model for the survival rate so that the response will be constrained to be between 0 and 1. More specifically, the logarithm of the odds of equivalence $\left(\frac{REM}{1-REM}\right)$ is a linear function of the predictors:

$$\log\left(\frac{REM}{1 - REM}\right) = \alpha + \beta \times X.$$

For any model M consisting of a set of the covariates X, we can obtain a residual deviance D(M) that provides an indication of the degree to which the response is unexplained by the model covariates. Hence, each model can be compared with the null model of no covariates to see if they are statistically different. Furthermore, any pair of nested models can be compared (using a chi-squared test).

We fit the full model with all five covariates, which was found to be statistically significant, and then successively drop a covariate, each time testing the smaller model (one covariate less) with the previous model. We continue until the smaller model was significantly different, i.e. worse than the previous model. Using the procedure described above, we find that the final model contains the metrics FR, NI and ND, with coefficient estimates and standard errors given in the table below:

Metric	**Estimate**	**Standard Error**	**P Value**
Intercept	-2.765	0.246	<<0.001
FR	0.459	0.268	0.086
NI	2.035	0.350	<<0.001
ND	0.346	0.152	0.023

Hence, the model is

$$\log\left(\frac{REM}{1-REM}\right) = -2.765 + 0.459 \times FR + 2.035 \times NI + 0.346 \times ND.$$

Each of the estimates are positive, hence, the survival rate increases with each of the three metrics. An increase in FR of 0.1 results in an expected increase in the odds by a factor of exp(0.1 x 0.459), or approximately 5%. Similarly increases of 0.1 in NI and ND each yields an expected increase of 22% and 3.5% respectively in the odds of survival.

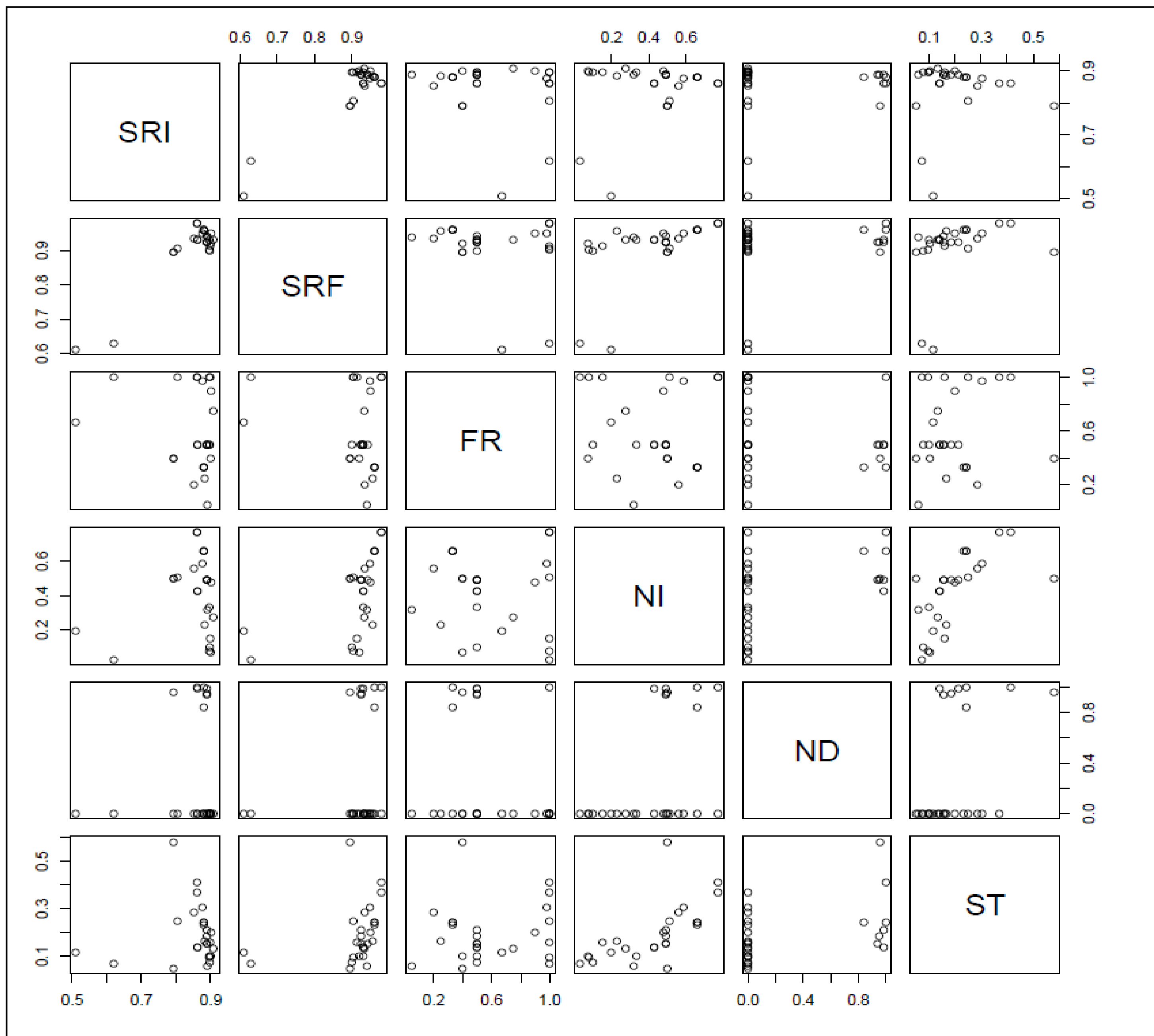


Table 5: Scatter Plot, Redundancy Metrics and Ratio of Equivalent Mutants

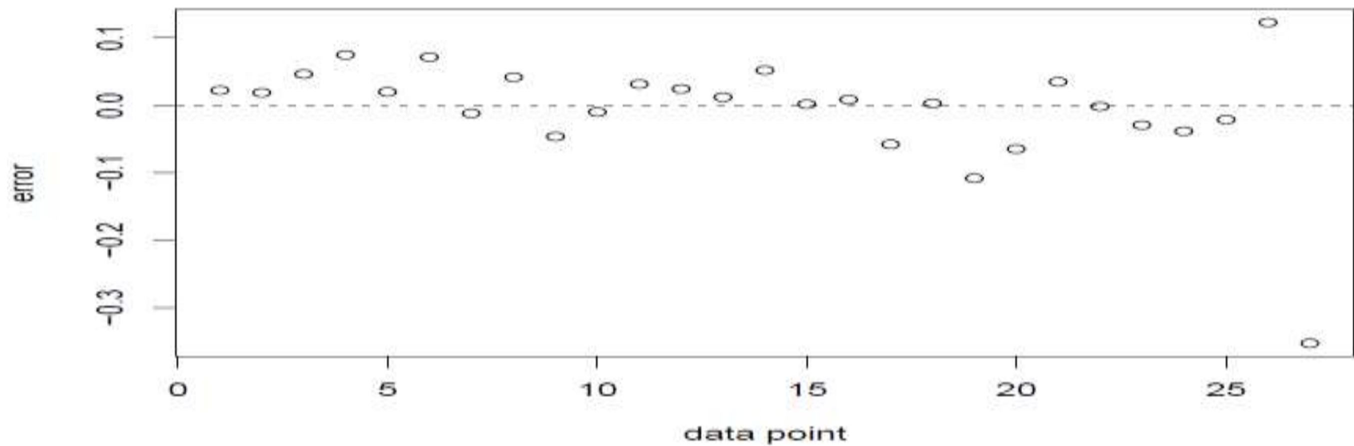
The sequence of models we tested, including their residual deviances, as well as the results of comparisons between them, are shown in the table below:

No.	Model	Deviance	Degrees of freedom	Test	P value
1	Null model	122.856	26		
2	SRI, SRF, FR, NI, ND	42.888	21	Models 2 and 1	<< 0.001

3	SRF, FR, NI, ND	57.447	22	Models 3 and 2	0.0001
4	SRI, FR, NI, ND	57.484	22	Models 4 and 2	0.0001
5	FR, NI, ND	57.74	23	Models 5 and 3	0.588
6	NI, ND	60.667	24	Models 6 and 5	0.087
7	FR, NI	62.955	24	Models 7 and 5	0.022

For the training data, the mean square error of the survival rate is 0.0069 and the mean absolute error is 0.049. We re-checked the analysis by performing take-one-out cross-validation, i.e., we removed each row of data in turn, fit the list of models from our previous analysis on the remaining data, then used the fitted models to predict the data point that was removed. For each model, the error is the difference between the predicted value from that model, and the actual value. The mean squared and absolute errors of 0.0087 and 0.057 respectively for the above final model were the smallest out of the list of models.

The plot below shows the relative errors of the model estimates with respect to the actuals; virtually all the relative errors are within less than 0.1 of the actuals.



V. IMPACT OF MUTANT GENERATION POLICY

A. Analyzing the Impact of Individual Operators

For all its interest, the regression model we present above applies only to the mutant generation policy that we used to build the model. This raises the question: how can we estimate the REM of a base program P under a different mutant generation policy? Because there are dozens of mutation operators in use by different researchers and practitioners, it is impossible to consider building a different model for each combination of operators. We could select a few sets of operators, that may have been the subject of focused research, or have a documented practical value [3] [2] [4] [24] and derive a specific estimation model for each. While this may be interesting from a practical standpoint, it presents limited interest as a research matter, as it does not enhance our understanding of how mutation operators interact with each other. What we are interested to understand is: if we know the REM's of a program P under individual mutation operators $op_1, op_2, \ldots, op_n$, can we estimate the REM of P if all of these operators are applied jointly?

Answering this question will enable us to produce a generic solution to the automated estimation of the REM of a program under an arbitrary mutant generation policy:

- We select a list of mutation operators of interest (e.g. the list suggested by Laurent et al [24] or by Just et al. [2], or their union).

- Develop a regression model (similar to the model we derived in section **Error! Reference source not found.**) based on each individual operator.
- Given a program P and a mutant generation policy defined by a set of operators, say $op_1, op_2, \dots, op_n$, we apply the regression models of the individual operators to compute the corresponding ratios of equivalent mutants, say $REM_1, REM_2, \dots, REM_n$.
- Combine the REM's generated for the individual operators to estimate the REM that stems from their simultaneous application.

B. *Combining Operators*

For the sake of simplicity, we first consider the problem above in the context of two operators, say op_1, op_2. Let REM_1, REM_2 be the REM's obtained for program P under operators op_1, op_2. We ponder the question: can we estimate the REM obtained for P when the two operators are applied jointly? To answer this question, we interpret the REM as the probability that a random mutant generated from P is equivalent to P. At first sight, it may be tempting to think of REM as the product of REM_1 and REM_2 on the grounds that in order for mutant M_{12} (obtained from P by applying operators op_1, op_2) to be equivalent with P, it suffices for M_1 to be equivalent to P (probability: REM_1), and for M_{12} to be equivalent to M_1 (probability: REM_2). This hypothesis yields the following formula of REM:

$$REM = REM_1 \times REM_2.$$

But we have strong doubts about this formula, for the following reasons:

- This formula assumes that the equivalence of P to M_1 and the equivalence of M_1 to M_{12} are independent events; but of course they are not. In fact we have shown in section IV that the probability of equivalence is influenced to a considerable extent by the amount of redundancy in P.
- This formula ignores the possibility that mutation operators may interfere with each other; in particular, the effect of one operator may cancel (all of or some of) the effect of another, or to the contrary may enable it.
- This formula assumes that the ratio of equivalent mutants of a program P decreases with the number of mutation operators; for example, if we have five operators that yield a REM of 0.1 each, then this formula yields a joint REM of 10^{-5}. We do not see why that should be the case; in fact we suspect that the REM of combined operators may be larger than that of individual operators.
- This formula also assumes that if a mutant by itself has an REM of 0, then any set of operators that includes it also has an REM of zero; but that is not consistent with our observations: it is very common for single operators to produce an REM of zero by themselves, but a non-trivial REM once they are combined with another.

For all these reasons, we expect $REM_{1\times}REM_2$ to be a loose (remote) lower bound for REM, but not be a good approximation thereof. Elaborating on the third item cited above, we argue that in fact, whenever we deploy a new mutation operator, we are likely to make the mutant more distinct from the original program, hence it is the probability of being distinct that we ought to compose, not the probability of being equivalent. This is captured in the following formula:

$$(1 - REM) = (1 - REM_1)(1 - REM_2),$$

which yields:

$$REM = REM_1 + REM_2 - REM_1 REM_2.$$

In the following sub-section we test our assumption regarding the formula of a combined REM.

C. *Empirical Validation*

In order to evaluate the validity of our proposed formula, we run the following experiment:

- We consider the sample of seventeen Java programs that we used to derive our model of sectionIV.
- We consider the sample of seven mutation operators that are listed in section **Error! Reference source not found.**.
- For each operator Op, for each program P, we run the mutant generator Op on program P, and test all the mutants for equivalence to P. By dividing the number of equivalent mutants by the total number of generated mutants, we obtain the REM of program P for mutation operator Op.
- For each mutation operator Op, we obtain a table that records the programs of our sample, and for each program we record the number of mutants and the number of equivalent mutants, whence the corresponding REM.

- For each pair of operators, say (Op1, Op2), we perform the same experiment as above, only activating two mutation operators rather than one. This yields a table where we record the programs, the number of mutants generated for each, and the number of equivalent mutants among these, from which we compute the corresponding REM. Since there are seven operators, we have twenty one pairs of operators, hence twenty one such tables.

- For each pair of operators, we build a table that shows, for each program P, the REM of P under each operator, the REM of P under the joint combination of the two operators, and the residuals that we get for the two tentative formulas:
  F1: $REM = REM_1 REM_2$,
  F2: $REM = REM_1 + REM_2 - REM_1 REM_2$.
  At the bottom of each such table, we compute the average and standard deviation of the residuals for formulas F1 and F2.
- We summarize all our results into a single table, which shows the average of residuals and the standard deviation of residuals for formulas F1 and F2 for each (of 21) combination of two operators.

D. Analysis

The final result of this analysis is given in Table 6. The first observation we can make from this table is that, as we expected, the expression $F1$: $REM_1 REM_2$ is indeed a lower bound for REM, since virtually all the average residuals (for all pairs of operators) are positive, with the exception of the pair (Op1, Op2), where the average residual is virtually zero. The second observation is that, as we expected, the expression $F2$: $REM_1 + REM_2 - REM_1 REM_2$ gives a much better approximation of the actual REM than the F1 expression; also, interestingly, the F2 expression hovers around the actual REM, with half of the estimates (11 rows) below the actuals and half above (10 rows). With the exception of one outlier (Op4,Op5), all residuals are less than 0.2 in absolute value, and two thirds (14 out of 21) are less than 0.1 in absolute value. The average (over all pairs of operators) of the absolute value of the average residual (over all programs) for formula F2 is 0.080

We have conducted similar experiments with three and four operators, and our results appear to confirm the general formula of the combined REM for N operators as:

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

Still, this matter is under further investigation.

VI. PROSPECTS

The model that we present in this paper is not the end of our study, but rather the beginning; we do not view it as a readily useful model, but rather as a proof of concept, i.e. empirical evidence to support our tentative conjecture about the correlation between redundancy (as quantified by our metrics) and the ratio of equivalent mutants.

A. Automation

Whereas in this paper we have computed the redundancy metrics by hand, by inspecting the base program and the oracle used to determine equivalence, we envision to build a tool that performs these calculations automatically. We envision to use compiler generation tools to this effect, to produce a simple compiler whose responsibility is to monitor variable declarations, assert statements, assignments statements, return statements, and parameter passing declarations to gather the necessary information. From this data, the compiler computes all the relevant entropies, then uses them to compute the metrics. Availability of this tool will make it possible for us to study larger size programs than we have done so far.

Table 6. Residuals for Candidate Formulas

	Residuals, F1		Residuals, F2		Abs(Residuals)	
Operator Pairs	average	Std dev	average	Std dev	F1	F2
Op1, op2	0.1242467	0.1884347	-0.0163621	0.0459150	0.1242467	0.0163621
Op1, op3	-0.0008928	0.0936731	0.0241071	0.0740874	0.0008928	0.0241071
Op1, op4	0.3616666	0.4536426	0.1797486	0.5413659	0.3616666	0.1797486
Op1, op5	0.1041666	0.2554951	0.0260416	0.3113869	0.1041666	0.0260416
Op1, op6	0.0777777	0.2587106	0.0777777	0.2587106	0.0777777	0.0777777
Op1, op7	0.0044642	0.0178571	-0.0625	0.25	0.0044642	0.0625
Op2, op3	0.1194726	0.122395	0.0658514	0.1397070	0.1194726	0.0658514
Op2, op4	0.1583097	0.1416790	-0.1246387	0.2763612	0.1583097	0.1246387
Op2, op5	0.1630756	0.1588826	-0.0535737	0.1469494	0.1630756	0.0535737
Op2, op6	0.2479740	0.4629131	0.0979913	0.332460	0.2479740	0.0979913
OP2,op7	0.1390082	0.1907661	-0.0535258	0.2445812	0.1390082	0.0535258
Op3, op4	0.1601363	0.1411115	0.1436880	0.3675601	0.1601363	0.1436880
Op3, op5	0.0583333	0.0898558	-0.0447916	0.1019656	0.0583333	0.0447916
Op3, op6	0.0166666	0.1409077	-0.0083333	0.0845893	0.0166666	0.0083333
OP3,op7	0.0152173	0.0504547	-0.0642468	0.2496315	0.0152173	0.0642468
Op4, op5	0.5216666	0.4221049	0.2786375	0.4987458	0.5216666	0.2786375
Op4, op6	0.3166666	0.2855654	0.1347486	0.4101417	0.3166666	0.1347486
OP4,op7	0.3472951	0.3530456	0.125903	0.3530376	0.3472951	0.125903
Op5, op6	0.075	0.1194121	-0.003125	0.1332247	0.075	0.003125
Op5, op7	0.078125	0.1760385	-0.0669642	0.2494466	0.078125	0.0669642
Op6, op7	0.0349264	0.0904917	-0.0320378	0.2735720	0.0349264	0.0320378
Averages	0.1487287		0.0297332		0.1488137	0.0802188

B. Statistical Model Revisited

Once we have built an automated tool for computing the redundancy metrics, we envision to revisit the statistical model for two purposes:

- First to calibrate our model to the metrics that are actually computed by the tool, rather than those that we want to compute (but may have to approximate as we automate).
- Second, to calibrate our model to programs of arbitrary size and complexity.

C. Identifying Individual Equivalent Mutants

While the focus of our work so far has been to estimate the number of equivalent mutants, we acknowledge that there are applications where it is important to single out those mutants that are found to be equivalent. To this effect, we envision to expand our tool for estimating the REM of a program as follows:

- The user provides the base program P, the mutation operators, and a probability threshold that represents the degree of certainty with which they want equivalent mutants to be identified.
- The tool computes the REM of P for the selected mutation operators using the models discussed in sections IV and V.
- Then the tool generates random test data and starts killing non-equivalent mutants, re-evaluating the probability that the surviving mutants are equivalent, until the estimated probability matches or exceeds the threshold dictated by the user.

VII. CONCLUDING REMARKS

A. Summary

The presence of equivalent mutants is a constant source of aggravation in mutation testing, because equivalent mutants distort our analysis and introduce biases that prevent us from making assertive claims. This has given rise to much research aiming to identify equivalent mutants by analyzing their source code or their run-time behavior. Analyzing their source code usually provides sufficient but unnecessary conditions of equivalence (as it deals with proving locally equivalent behavior); and analyzing run-time behavior usually provides necessary but insufficient conditions of equivalence (just because two programs have comparable run-time behavior does not mean they are functionally equivalent). Also, static analysis of mutants is generally time-consuming and error-prone, and wholly impractical for large and complex programs and mutants.

In this paper, we submit four simple premises for the study of equivalent mutants:

- First, for most practical purposes, determining which mutants are equivalent to a base program (and which are not) is not important, provided we can estimate their number.
- Second, even when it is important to single out equivalent mutants, knowing their number can greatly facilitate the task of singling them out; it could in fact be automated.
- Third, what makes a program prone to produce equivalent mutants is the same attribute that makes it fault tolerant, since fault tolerance is by definition the property of maintaining correct behavior in the presence of faults. The attribute that makes programs fault tolerant is well-known: redundancy. Hence we can estimate the ratio of equivalent mutants of a program by analyzing/ quantifying its level of redundancy.
- Fourth, it may be possible to estimate the ratio of equivalent mutants of a program for an arbitrary set of mutation operators, assuming we have a regression model for estimating the REM of a program for each individual operator.

B. Assessment and Prospects

Whereas the focus of the paper is the derivation of a statistical model that enables us to estimate the ratio of equivalent mutants from the redundancy metrics, we do not consider that the model is the main contribution of the paper. Rather, the main contribution of the paper is the premise that the ratio of equivalent mutants is determined, to a large extent, by the amount of redundancy in the base program and the test oracle that we use to determine equivalence; section III gives analytical justification of this claim, and

section IV supports this claim with empirical evidence. As for the statistical model, we view it as a proof of concept more than a readily usable tool; instead, we envision to revisit it and refine it once we have developed an automatic tool to compute the redundancy metrics of programs and the non-determinacy of oracles.

Another noteworthy contribution of this paper, perhaps controversial, is the claim that the concept of equivalent mutant is not as clear-cut as we have been assuming; while we agree that equivalence between a base program and a mutant is equated with equivalent behavior, it depends on what we mean by *behavior* (i.e. what functional details are captured) and what we mean by *equivalent* behavior (what aspects are captured by the oracle). On the basis of this claim, we have included the oracle non-determinacy as a factor in our statistical analysis.

Threats to the validity of our study include the fact that it fails to take into account mutations that are applied to dead code. This is inherent to our approach, in that it is based on an analysis of the functional properties of the program at hand, whereas dead code is essentially a structural attribute of the program. We address this issue partially by monitoring the line coverage provided by the test data: we exclude from our statistical analysis any execution that does not ensure a line coverage greater than or equal to 90%; but only 100% coverage ensures the absence of dead code.

Our prospects for future research include the automation of computing redundancy metrics, the derivation of validated statistical models (in particular: regression models), the analysis of the impact of mutation operators on the ratio of equivalent mutants, and the inclusion of OOP-specific metrics of redundancy.

Acknowledgement. This research is partially supported by a grant to the last author from the (US) National Science Foundation, number DGE 1565478.

VIII. REFERENCES

[1] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering,* vol. 37, no. 5, pp. 649-678, 2011.

[2] 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 *Foundations of Software Engineering*, Hong Kong, China, 2014.

[3] J. H. Andrews, L. C. Briand and I. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments," in *International Conference on Softwaare Testing*, St Louis, MO USA, 2005.

[4] A. S. Namin and S. Kakarla, "The Use of Mutation in Testing Experiments and its Sensitivity to External Threats," in *ISSTA'11*, Toronto Ont Canada, 2011.

[5] X. Yao, M. Harman and Y. Jia, "A Study of Equivalent and Stubborn Mutation Operators using Human Analysis of Equivalence," in *Proceedings, International Conference on Software Engineering*, Hyderabad, India, 2014.

[6] D. Schuler and A. Zeller, "Covering and Uncovering Equivalent Mutants," *Journala of Software Testing, Verificaation and Reliability,* vol. 23, no. 5, pp. 353-374, 2012.

[7] B. J. Gruen, D. Schuler and A. Zeller, "The Impact of Equivalent Mutants," in *MUTATION 2009*, Denver, CO USA, 2009.

[8] 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.

[9] R. Just, M. D. Ernst and G. Fraser, "Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States," in *ISSTA '14*, San Jose, CA USA, 2014.

[10] B. Wang, Y. Xiong, Y. Shi, L. Zhang and D. Hao, "Faster Mutation Analysis via Equivalence Modulo States," in *ISSTA'17*, Santa Barbara CA USA, 2017.

[11] M. Papadakis, M. Delamaro and Y. Le Traon, "Mitigating the Effects of Equivalent Mutants with Mutant Classification Strategies," *Science of Computer Programming,* vol. 95, no. 12, pp. 298-319, 2014.

[12] T. A. Budd and D. Angluin, "Two Notions of Correctness and their Relation to Testing," *Acta Informatica,* vol. 18, no. 1, pp. 31-45, 1982.

[13] J. A. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *The Journal of Software Testing, Verification and Reliability,* vol. 7, no. 3, pp. 164-192, 1997.

[14] J. Voas and G. McGraw, Software Fault Injection: Inoculating Programs Against Errors, New York, NY: John Wiley and Sons, 1997.

[15] M. Harman, R. Hierons and S. Danicic, "The Relationship Between Program Dependence and Mutation Analysis," in *MUTATION '00*, San Jose, CA USA, 2000.

[16] 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,* vol. 9, no. 4, pp. 233-262, 1999.

[17] M. Ellims, D. C. Ince and M. Petre, "The Csaw C Mutation Tool: Initial Results," in *MUTATION '07*, Windsor, UK, 2007.

[18] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Softwaare Engineering,* vol. 8, no. 4, pp. 371-379, 1982.

[19] D. Schuler, V. Dallmaier and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in *ISSTA '09*, Chicago, IL USA, 2009.

[20] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering,* vol. 27, no. 2, pp. 99-123, 2001.

[21] S. Nica and F. Wotawa, "Using Constraints for Equivalent Mutant Detection," in *Workshop on Formal Methods in the Development of Software*, 2012.

[22] L. Carvalho, M. A. Guimaraes, L. Fernandes, M. Al Hajjaji, R. Gheyi and T. Thuem, "Equivalent Mutants in Configurable Systems: An Empirical Study," in *VAMOS 2018*, Madrid, Spain, 2018.

[23] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon and M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimizations," *IEEE Transactions on Software Engineering,* vol. 44, no. 4, pp. 308-333, 2018.

[24] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol. 27, no. July/October, pp. 379-423, 1948.

[25] I. Csiszar and J. Koerner, Information Theory: Coding Theorems for Discrete Memoryless Systems, Cambridge, UK: Cambridge University Press, 2011.