# Estimating the Survival Rate of Mutants

Imen Marsit[1], Mohamed Nazih Omri[1] and Ali Mili[2]
[1] *Faculty of Science of Monastir, Tunisia*
[2] *New Jersey Institute of Technology, Newark NJ*
{*imeni4@yahoo.fr, mohamednazih.omri@fsm.rnu.tn, mili@njit.edu*}

Keywords: Mutation Testing, Mutant Survival Rate, Semantic metrics

Abstract: Mutation testing is often used to assess the quality of a test suite by analyzing its ability to distinguish between a base program and its mutants. The main threat to the validity/ reliability of this assessment approach is that many mutants may be syntactically distinct from the base, yet functionally equivalent to it. The problem of identifying equivalent mutants and excluding them from consideration is the focus of much recent research. In this paper we argue that it is not necessary to identify individual equivalent mutants and count them; rather it is sufficient to estimate their number. To do so, we consider the question: what makes a program prone to produce equivalent mutants? Our answer is: redundancy does. Consequently, we introduce a number of program metrics that capture various dimensions of redundancy in a program, and show empirically that they are statistically linked to the rate of equivalent mutants.

## 1 Equivalent Mutants

Mutation testing is typically used to assess the effectiveness of test suites, by analyzing to what extent a test suite $T$ can distinguish between a base program $P$ and a set of mutants thereof, say $P_1$, $P_2$, ... $P_N$ (Debroy and Wong, 2010; Debroy and Wong, 2013; V. and W.E., 2010; Ma et al., 2005; Zemín et al., 2015). A recurring source of aggravation in mutation testing is the presence of equivalent mutants: some mutants may be syntactically distinct from the base program, yet be functionally indistinguishable from it (i.e. compute the same function). Equivalent mutants distort the analysis of the test suite's effectiveness, because when a test suite fails to distinguish a mutant $P_i$ from the base program $P$, we do not know whether this is because $P_i$ is equivalent to $P$, or because $T$ is not sufficiently effective at detecting faults. Ideally, we want to quantify the effectiveness of a test suite $T$, not by the ratio of the mutants it distinguishes over the total number of mutants ($N$), but rather by the ratio of the mutants it distinguishes over the number of non-equivalent (distinguishable) mutants. Many researchers (Aadamopoulos et al., 2004; Papadakis et al., 2014; Schuler and Zeller, 2010; Just et al., 2013; Nica and Wotawa, 2012) have addressed this problem by proposing means to identify (and exclude from consideration) mutants that are equivalent to the base program.

In this paper, we propose an alternative approach, which does not seek to identify which mutants are equivalent to the base, but merely to estimate their number. To do so, we consider the research question (RQ3) raised by Yao et al (Yao et al., 2014): What are the causes of mutant equivalence? Our answer: Redundancy in the program. Consequently, we define a number of software metrics that capture various forms of redundancy, discuss why we believe they are prone to produce equivalent mutants, then run an experiment that appears to bear out conjecture out. This is work in progress; we are fairly confident that our analytical arguments are sound, and we are encouraged by the preliminary empirical results.

Following common usage, we say that a mutant has been *killed* by a test data set $T$ if and only if execution of the mutant on $T$ exhibits a distinct behavior from the original program $P$; consequently, when a mutant goes through test data $T$ and shows the same behavior as $P$, we say that it has *survived* the test. Also, we use the term *survival rate* of a program $P$ to designate the ratio (or percentage) of mutants of $P$ that are found to survive the execution of test data $T$.

For the purpose of our study, we use the semantic metrics introduced in (Mili et al., 2014), which we briefly review in section 2. In section 3 we discuss why we feel that the semantic metrics introduced in section 2 are good indicators of the number of potentially equivalent mutants that a program $P$ may yield.

In sections 4 and 5 we present the details of our empirical study by discussing in turn the experimental set up we have put together, then the preliminary results we have found. Finally in section 6 we assess our prelimnary findings and sketch directions of further research.

## 2 Semantic Metrics

Whereas traditional software metrics usually capture properties of the source code of a program, semantic software metrics reflect functional properties of programs and specifications, regardless of how these programs and specifications are represented (Mili et al., 2014; Morell and Voas, 1993; Voas and Miller, 1993; Gall et al., 2008; Etzkorn and Gholston, 2002). We consider four semantic metrics, due to (Mili et al., 2014), and we discuss in the next section why we believe that these metrics may give us some indication on the survival rate of the mutants of a program. To discuss these metrics, we consider a program $P$ on space $S$, where $S$ is defined by the variable declarations that appear in program $P$; elements of $S$ are denoted by lower case $s$, and are referred to as *states* of the program; by abuse of notation, we use the same symbol ($P$) to denote the program as well as the function that the program defines from its initial states to its final states. We also consider a specification $R$ in the form of a binary relation on $S$, which represents the (initial state, final state) pairs that are considered correct by $R$. All the metrics we discuss are based on the entropy function, with which we assume the reader to be familiar (Csiszar and Koerner, 2011).

### 2.1 State Redundancy

It is very common for programs to have a state space that is much larger than the range of values that it takes in any execution. Examples abound: we declare an integer variable ($2^{32}$ values) to represent a boolean variable (2 values); we declare an integer variable ($2^{32}$ values) to represent the age of a person in years (120 values, to be optimistic); we declare three integer variables ($2^{96} \approx 10^{29}$ values) to represent the current year, birth year, and current age of a person (about $10^4$ values if we consider that one of the three variables is redundant, and take 120 for maximum age and 100 for longevity of the application). The state redundancy of a program is the difference between the entropy of the declared state and the entropy of the actual state; because the entropy of the actual state may change (decrease) during the execution of the program, we distinguish between the initial state redundancy and the final state redundancy. If we let $S$, $\sigma_I$ and $\sigma_F$ be (respectively) the random variable that ranges over the declared space, the initial actual space and the final actual space, then we define:

- Initial state redundancy: $\kappa_I(P) = H(S) - H(\sigma_I)$.
- Final state redundancy: $\kappa_F(P) = H(S) - H(\sigma_F)$.

### 2.2 Functional Redundancy

In the fault tolerant computing literature, it is common to see references to modular redundancy, whereby the same function is computed by different modules and their results compared to detect failures (for double modular redundancy) and recover from them (for triple modular redundancy, or higher). But we do not need deliberate, pre-planned, dependability-driven measures to have functional redundancy; we consider that we observe functional redundancy whenever a function produces more information than is strictly necessary to represent its result. If we let $Y$ be the random variable that represents the range of the program's function, then the functional redundancy of program $P$ is denoted by $\phi$ and defined by:

$$\phi(P) = \frac{H(S) - H(Y)}{H(Y)}.$$

For double modular redundancy, this formula yields $\phi(P) = 1$ and for triple modular redundancy it yields $\phi(P) = 2$; but it takes arbitrary (not necessarily integer) non-negative values.

### 2.3 Non-Injectivity

The injectivity of a function is the property of the function to change its image as its argument changes; the non-injectivity of a function is the property of the function to map several distinct arguments onto the same image. Programs are usually vastly non-injective: a routine that sorts an array of size $N$ maps $(N!)$ distinct arrays (corresponding to $(N!)$ distinct permutations of an array of $N$ distinct cells) into a single (sorted) permutation. We consider program $P$ and we let $X$ be the random variable that represents the domain of $P$. A natural way to quantify the non-injectivity of function $P$ is to consider the conditional entropy of $X$ given $P(X)$; in other words, if we know what $P(X)$ is, how much uncertainty do we have about $X$. If $P$ is injective then the conditional entropy is zero, since $P(X)$ uniquely determines $X$; but if $P$ is not injective then the conditional entropy reflects the number of antecedents that $Y = P(X)$ may have; in the case of the sorting routine, for example, if we look at the output of the routine in the form of a sorted

array, then we know that the input could be any (of the $(N!)$) permutations of this array. Hence we define non-injectivity as: $\theta(P) = H(X|P(X))$.

### 2.4 Non Determinacy

A specification is non-deterministic if and only if it assigns multiple final states to a given initial state; let $S$ be the space defined by three variables, say $x$, $y$ and $z$, and let $R$ be a specification of the form

$$R_0 = \{(s,s') | x' = f_x(s) \wedge y' = f_y(s) \wedge z' = f_z(s)\},$$

where $s$ stands for the aggregate $\langle x,y,z \rangle$ and $f_x()$, $f_y()$, $f_z()$ are three functions that can be evaluated for any $s$ in $S$. Then $R_0$ is deterministic since for each $s$ it assigns a single $s'$. The following three specifications are increasingly non-deterministic, since they restrict final states less and less, hence allow more and more final states.

$R_1 = \{(s,s') | x' = f_x(s) \wedge y' = f_y(s)\}$,
$R_2 = \{(s,s') | x' = f_x(s)\}$,
$R_3 = \{(s,s') | \textbf{true}\}$.

If we let $X$ be a random variable that takes its values in the domain of $R$, and let $R(X)$ be the set of images of $X$ by $R$, then we define the non-determinacy of $R$ as the conditional entropy of $R(X)$ given $X$, i.e. how much uncertainty we have about the image of $X$ if we know $X$: $\chi(R) = H(R(X)|X)$.

## 3 Analytical Study

The question that we address here is: what reason do we have to believe that the semantic metrics we have discussed in the previous section have any relation with the survival rate of mutants of a program?

### 3.1 State Redundancy

State redundancy reflects to what extent the aggregate of program variables carry more bits than necessary to represent the actual program state; the increase in state redundancy yields an increase in the volume of bits that have no effect on the execution of the program. Hence with an increase in state redundancy, we may expect an increase in the likelihood that the execution of a mutant will affect irrelevant parts of the state, hence will yield equivalent behavior.

### 3.2 Functional Redundancy

To the extent that added functional redundancy is equated with greater tolerance to faults, it is normal to expect that a program that has greater functional redundancy is better able to recover from deviations caused by a mutated statement. Even in the absence of deliberate fault tolerance measures, increased functional redundancy means increased likelihood that the final state is determined by non-affected functional information.

### 3.3 Non Injectivity

The non-injectivity of a program reflects the breadth of distinct initial (or intermediate) states that map to the same final state. The higher the non-injectivity of a program, the greater the likelihood that a state generated by a base program and a state generated by a mutant are mapped to the same final state.

### 3.4 Non Determinacy

Let $S$ be the space defined by three integer variables $x$, $y$ and $z$, and let $R$ and $R'$ be the following specifications on $S$:

$R = \{(s,s') | x' = y \wedge y' = x\}$.
$R' = \{(s,s') | x' = y \wedge y' = x \wedge z' = x\}$.

Let $P$ (base) and $M$ (mutant) be the following programs on space $S$:

```
p:  {z=x; x=y; y=z;}
m:  {z=y; y=x; x=z;}
```

Whether $M$ is found to be equivalent to $P$ or not depends on which oracle we use to test $M$: If we use specification $R$ (which focuses on $x$ and $y$) then we find it to be equivalent, and if we use $R'$ we find it to be distinct. Specification $R$ differs from $R'$ in its degree of non-determinacy:

$\chi(R) = 32$ bits,
$\chi(R') = 0$ bits.

As the non-determinacy of a specification increases the determination of equivalence weakens, and more programs are considered equivalent.

## 4 Empirical Study

### 4.1 Computing Survival Rates

In order to validate our conjecture that the semantic metrics are correlated to the survival rate of a program's mutants, we consider a number of programs, compute their semantic metrics, then we deploy mutant generators to produce a set of mutants for each program, and check how many mutants survive the test in each case. To this effect, we consider a sample of thirteen (13) programs from the *Common Math Library*, a library of open source Java

projects, along with their corresponding JUnit automated unit test data sets. For each class in the library, say `Myclass.java`, corresponds a test class `MyclassTest.java`, which contains a number of test cases `TestMethod()` for each method of the class.

In order to generate mutants, we use the *Pitest* tool (Coles, 2017), which integrates smoothly with JUnit; we use Maven (Inozemtseva and Holmes, 2014) to build and run tests. First, we specify to *Pitest* which class to mutate, along with the corresponding test class; upon deployment, it generates mutants according to the specified parameters and runs the selected test suite on each one of them. Then it produces a report that shows: the line of code it mutates for each mutant; the total number of mutants; and the number of killed and live mutants. Because we have no control over the test oracle (this is handled automatically by the test environment) we do not include non-determinacy in our current study.

### 4.2 Computing Semantic Metrics

Given a random variable $X$ that ranges over a set of values $\{x_1, x_2, ...x_N\}$, the entropy of $X$ is defined with respect to a probability distribution, say $\pi()$, which assigns a probability of occurrence $\pi(x_i)$ to each value of $X$. The formula of the entropy is:

$H(X) = -\sum_{i=1}^{N} \pi(x_i)\log(\pi(x_i))$,

and is given in *bits*. When the probability distribution $\pi()$ is uniform, we have $\pi(x_i) = \frac{1}{N}$ for all $i$, which yields $H(X) = \log(N)$. For a random variable $X$ that takes values of a 32-bit integer, $N$ equals $2^{32}$ and $\log(N)$ is then merely equal to 32 bits. For the sake of simplicity, we assume uniform probability throughout, so that the entropy of any program variable is basically the number of bits in that variable. As for computing conditional entropies, we use the definition $H(X|Y) = H(X,Y) - H(Y)$, and the property that if $Y$ is a function of $X$ then $H(X,Y) = H(X)$, so that the conditional entropy in these cases is $H(X|Y) = H(X) - H(Y)$.

### 4.3 Raw Data

The raw data stemming from our experiment is given in table 1. Columns 1 to 4 are filled using the data collected in section 4.1 and the last column is filled using the methods described in section 4.2.

## 5 Statistical Observations

### 5.1 Correlations

Table 2 shows the correlation between the five factors of table 1; of greatest interest to us is the correlation between the survival rate and the semantic metrics. As shown in the rightmost column of table 2 these correlations are very high, ranging from 0.728 for initial state redundancy to 0.764 for final state redundancy to 0.802 for functional redundancy to 0.921 for non-injectivity; this bears out our conjectures.

### 5.2 Regression

We run a linear regression on this data, using the survival rate as a dependent variable and the semantic metrics as the independent variables, and we find the following equation:

$SR = 2.417 + 0.012 \times SR_i + 0.001 \times SR_f + 0.116 \times FR + 0.172 \times NI$,

where $SR$, $SR_i$, $SR_f$, $FR$ and $NI$ represent, respectively, the survival rate, the initial state redundancy, the final state redundancy, the functional redundancy and the non-injectivity. As far as regression statistics, we find 0.881 for $R^2$ and 0.821 for adjusted $R^2$. Table 3 shows the comparison between the estimated values and the actual values of the survival rates (as percentages), along with the residuals. We find the average of residuals to be -0.132 and the standard devitaion to be 0.516; except for some outliers,most residuals are within 20%, and many are well within 10%.

## 6 Assessment and Prospects

Mutation testing is commonly used as a means to assess the effectiveness of a test data set: Given a program $P$ and a test data set $T$, if we generate $N$ mutants of $P$ and run them on data $T$ and find that they all exhibit a different behavior from $P$, we can conclude that $T$ is very effective at detecting faults in $P$. But if test data $T$ can only distinguish, say 60% of the mutants from $P$, we have no way to tell whether it is because $T$ is ineffective or because 40% of the generated mutants are equivalent to $P$.In order to distinguish between these two situations, we need to know how many mutants of $P$ are distinguishable from $P$, and assess the effectiveness of $T$ with respect to this number, rather than with respect to $N$.

While most other approaches address the problem of equivalent mutants by trying to characterize / identify equivalent mutants and counting them, we attempt to estimate how many equivalent mutants a

| Function | Initial State Redundancy | Final State Redundancy | Functional Redundancy | Non-Injectivity | Survival Rate |
|---|---|---|---|---|---|
| Vector1D | 229.44 | 293.44 | 11.04 | 128.00 | 33.33 |
| subandcheck | 50.72 | 114.72 | 8.63 | 64.00 | 16.67 |
| addandcheck | 50.72 | 32.00 | 2.40 | 64.00 | 33.33 |
| multiplyentry | 681.68 | 685.00 | 8.25 | 128.00 | 33.33 |
| trigamma | 55.36 | 57.36 | 8.63 | 64.00 | 20.83 |
| distanceinf | 688.32 | 792.12 | 19.80 | 192.00 | 50.00 |
| pow | 114.72 | 121.36 | 0.90 | 128.00 | 11.11 |
| getreducedfraction | 50.20 | 82.71 | 6.22 | 64.00 | 11.76 |
| linearCombination | 1092.00 | 1509.44 | 56.83 | 256.00 | 66.67 |
| gamma | 234.08 | 313.36 | 2.32 | 64.00 | 18.75 |
| normalizeArray | 702.00 | 812.88 | 15.90 | 0.00 | 6.25 |
| addSub | 101.72 | 447.80 | 13.84 | 64.00 | 16.67 |
| fraction | 545.00 | 609.00 | 1.58 | 64.00 | 28.12 |

Figure 1: Raw Data

| | Initial State Redundancy | Final State Redundancy | Functional Redundancy | Non-Injectivity | Survival Rate |
|---|---|---|---|---|---|
| Initial State Redundancy | 1 | 0.985752 | 0.722747 | 0.573626 | 0.727965 |
| Final State Redundancy | | 1 | 0.80808 | 0.621823 | 0.763944 |
| Functional Redundancy | | | 1 | 0.704976 | 0.801607 |
| Non-Injectivity | | | | 1 | 0.92079 |
| Survival Rate | | | | | 1 |

Figure 2: Correlation Table

| Function | Actual Survival Rate | Estimated Survival Rate | Relative Residual |
|---|---|---|---|
| Vector1D | 33.33 | 28.760 | 0.137 |
| subandcheck | 16.67 | 15.149 | 0.091 |
| addandcheck | 16.67 | 14.344 | 0.139 |
| multiplyentry | 33.33 | 34.255 | -0.028 |
| trigamma | 20.83 | 15.148 | 0.273 |
| distanceinf | 50.00 | 46.789 | 0.064 |
| pow | 11.11 | 26.035 | -1.343 |
| getreducedfraction | 11.76 | 14.831 | -0.261 |
| linearCombination | 66.67 | 67.654 | -0.015 |
| gamma | 18.75 | 16.816 | 0.103 |
| normalizeArray | 6.25 | 13.498 | -1.159 |
| addSub | 16.67 | 16.353 | 0.019 |
| fraction | 28.12 | 20.757 | 0.262 |

Figure 3: Actuals, Estimates, and Residuals

program is prone to produce; in other words, we do not need to know which mutants are equivalent, all we need to know is how many equivalent mutants we think there are. Our approach proceeds by analyzing several forms of redundancy within the program, which can be quantified by a set of semantic metrics. In this paper, we present the semantic metrics in question, discuss why we believe that they are related to the survival rate of mutants of a program, then show empirical evidence to support our conjecture, and provide an approximate formula that estimates the survival rate of a program as a function of its semantic metrics. We envision the following extensions to our work:

- Because the dependent variable is normalized (a percentage), we argue that the independent variables ought to be normalized as well; hence we envision to redo the statistical analysis using normalized metrics (most can be normalized to the entropy of the declared state of the program).
- Building a larger base of programs, involving larger size programs.
- Validate the new regression formula on a realistic set of programs, distinct from the programs with which we build the predictive model.
- Automating the calculation of the semantic metrics, ideally by the generation of a compiler that analyzes the source code of the program to compute its semantic metrics.
- Consider the oracle that is used in identifying equivalent mutants, and integrate the nondeterminacy of the oracle in the statistical model.
- Consider a possible standardization of the test data we use to test mutants: currently, we are relying on the test data provided by the Commons Math Library. It is possible that some test classes are more thorough than others; yet in order for our statistical study to be meaningful, all test classes need to be equally thorough; we need to define standards across classes.

# REFERENCES


Aadamopoulos, K., Harman, M., and Hierons, R. (2004). How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings, Genetic and Evolutionary Computation*, volume 3103 of *LNCS*, pages 1338–1349.

Coles, H. (2017). Real world mutation testing. Technical report, pitest.org.

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

Debroy, V. and Wong, W. E. (2010). Using mutations to automatically suggest fixes for faulty programs. In *Proceedings, ICST*, pages 65–74.

Debroy, V. and Wong, W. E. (2013). Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60.

Etzkorn, L. H. and Gholston, S. (2002). A semantic entropy metric. *Journal of Software Maintenance Evolution: research and Practice*, 14:293–310.

Gall, C. S., Lukins, S., Etzkorn, L., Gholston, L., Farrington, P., Utley, D., Fortune, J., and Virani, S. (2008). Semantic software metrics computed from natural language design specifications. *IET Software*, 2(1).

Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Procedings, 36th International Conference on Software Engineering*. ACM Press.

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

Ma, Y.-S., Offutt, J., and Kwon, Y.-R. (2005). Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133.

Mili, A., Jaoua, A., Frias, M., and Helali, R. G. (2014). Semantic metrics for software products. *Innovations in Systems and Software Engineering*, 10(3):203–217.

Morell, L. J. and Voas, J. M. (1993). A framework for defining semantic metrics. *Journal of Systems and Software*, 20(3):245–251.

Nica, S. and Wotawa, F. (2012). Using constraints for equivalent mutant detection. In Andres, C. and Llana, L., editors, *Second Workshop on Formal methods in the Development of Software*, EPTCS, pages 1–8.

Papadakis, M., Delamaro, M., and LeTraon, Y. (2014). Mitigating the effects of equivalent mutants with mutant clasification strategies. *Science of Computer Programming*, 95(P3):298–319.

Schuler, D. and Zeller, A. (2010). (un-)covering equivalent mutants. In *Proceedings, International Conference on Software Testing, Verification and Validation*, pages 45–54.

V., D. and W.E., W. (2010). Using mutation to automatically suggest fixes to faulty programs. In *Proceedings, ICST 2010*, pages 65–74.

Voas, J. M. and Miller, K. (1993). Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216.

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

Zemín, L., Guttiérrez, S., de Rosso, S. P., Aguirre, N., Mili, A., Jaoua, A., and Frias, M. (2015). Stryker: Scaling specification-based program repair by pruning infeasible mutants with sat. Technical report, ITBA, Buenos Aires, Argentina.