Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

QUANTITATIVE METRICS FOR MUTATION TESTING

by
Amani Ayad

Program mutation is the process of generating versions of a base program by applying elementary syntactic modifications; this technique has been used in program testing in a variety of applications, most notably to assess the quality of a test data set. A good test set will discover the difference between the original program and mutant except if the mutant is semantically equivalent to the original program, despite being syntactically distinct.

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; indeed, mutants are useful only to the extent that they define distinct functions from the base program. Yet, despite several decades of research, the identification of equivalent mutants remains a tedious, inefficient, ineffective and error prone process.

The approach that is adopted in this dissertation is to turn away from the goal of identifying individual mutants which are semantically equivalent to the base program, in favor of an approach that merely focuses on estimating their number. To this effect, the following question is considered: What makes a base program P prone to produce equivalent mutants? The position taken in this work is that what makes a program prone to generate equivalent mutants is the same property that makes a program fault tolerant, since fault tolerance is by definition the ability to maintain correct behavior despite the presence and sensitization of faults; whether these faults stem from poor design or from

mutation operators does not matter. Hence if the redundancy of the program could be quantified, the redundancy metrics could be used to estimate the ratio of equivalent mutants (REM) of a program.

Using redundancy metrics that were previously defined to reflect the state redundancy of a program, its functional redundancy, its non injectivity and its non-determinacy, this dissertation makes the following contributions:

- The design and implementation of a Java compiler, using compiler generation technology, to analyze Java code and compute its redundancy metrics.
- An empirical study on standard mutation testing benchmarks to analyze the statistical relationships between the REM of a program and its redundancy metrics.
- The derivation of regression models to estimate the REM of a program from its compiler generated redundancy metrics, for a variety of mutation policies.
- The use of the REM to address a number of mutation related issues, including: estimating the level of redundancy between non-equivalent mutants; redefining the mutation score of a test data set to take into account the possibility that mutants may be semantically equivalent to each other; using the REM to derive a minimal set of mutants without having to analyze all the pairs of mutants for equivalence.

The main conclusions of this work are the following:

- The REM plays a very important role in the mutation analysis of a program, as it gives many useful insights into the properties of its mutants.
- All the attributes that can be computed from the REM of a program are very sensitive to the exact value of the REM; Hence the REM must be estimated with great precision.

Consequently, the focus of future research is to revisit the Java compiler and enhance the precision of its estimation of redundancy metrics, and to revisit the regression models accordingly.

QUANTITATIVE METRICS FOR MUTATION TESTING

by
Amani Ayad

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

December 2019

APPROVAL PAGE

QUANTITATIVE METRICS FOR MUTATION TESTING

Amani Ayad

Dr. Ali Mili, Dissertation Co-Advisor — Date
Professor of Computer Science, NJIT

Dr. Ji Meng Loh, Co-Advisor — Date
Assistant Professor of Mathematical Sciences, NJIT

Dr. Jason T. Wang, Committee Member — Date
Professor of Computer Science, NJIT

Dr. James Geller, Committee Member — Date
Professor of Professor of Computer Science, NJIT

Dr. Vincent Oria, Committee Member — Date
Professor of Professor of Computer Science, NJIT

Dr. Iulian Neamtiu, Committee Member — Date
Assistant Professor of Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Amani Ayad

Degree: Doctor of Philosophy

Date: December 2019

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2019
- Master of Computer Science,
  Monmouth University, West Long Branch, NJ, 2014
- Bachelor of Computer Science,
  Garyounis University, Benghazi, Libya, 2000

Major: Computer Science

Presentations and Publications:

Personal Biography:

A. Ayad, I. Marsit, J. M.Loh, M.N. Omri and A. Mili, "Estimating the Number of Equivalent Mutants". *The 14th International Workshop on Mutation Analysis 2019.*

A. Ayad, I. Marsit, J. M.Loh, M.N. Omri and A. Mili, "Quantitative Metrics for Mutation Testing". *The 14th International Conference on Software Technologies 2019.*

A. Ayad, I. Marsit, J. M.Loh, M.N. Omri and A. Mili, "Using Semantic Metrics to Predict Mutation Equivalence". *in Communication in Computers and Information Science Series, Heidelberg, Germany:Springer- Verlag, 2019, Ch.1.*

To my beloved father Maraja Ayad, his soul is always surrounding me. To my loving mum, Salmeen Bader and my dearest sister Dr.Eman Ayad for continuing motivation, and support...Without you, this would not have been possible.

ACKNOWLEDGMENT

I would like to thank my dissertation advisor, Dr. Ali Mili, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. Also, I must thank Dr. Ji Meng Loh, my second mentor, for helping me in analyzing my data and improving the accuracy of my results. I am also extremely grateful to Dr. James Geller for supporting and giving me the opportunity to do research during my first semester and recently spent his time fixing my writing. In addition, I would like to extend special thanks to Dr. Jason Wang for always supporting, inspiring me, being his TA for two years, doing research, and giving me great academic advice.

Special thanks are expressed to Dr. Vincent Oria and Dr. Iulian Neamtiu for serving on my defense committee. I want to thank them for the time they have spent to provide me with their valuable feedback and suggestions on my research.

Finally, I thank my parents, sister, my friends, NJIT, and the CS department staff for their love and support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Survey of Mutation Testing

Modifying a program syntactically generates artificial defects, called mutants [1]. Mutation testing analysis is the process of assessing the strength, effectiveness and ability of test suites to detect mutants. It has been a research topic for over four decades. Early in the 1970s, mutation analysis was developed [1,2,3] and it has gradually increased in academia and in industry. DeMillo [1,4] (1989) summarizes the work of mutation testing in a survey.

Also, Jia and Harman [1,5] (2011) provides the evidence that mutation testing techniques and tools are reaching a state of maturity and applicability, while the topic of mutation testing itself is the subject of increasing interest.

Moreover, there are specific surveys that discusses various issues in mutation testing. For instance, Madeyski et al. [1,6] (2014) studies the equivalent mutant problem which is introduced in section [1.2]. Souza et al. [1,7] (2014) proposes a systematic mapping of mutation-based test generation. Belli et al. [1-8] (2016) writes a survey on model-based mutation testing. Silva et al. [1, 9] (2017) presents a methodical review on mutation testing. Papadakis et al. [1] (2017) collects and analyzes a set of 502 papers that are published in various conferences from 2008 to 2017.

In Figure 1.1, Papadakis provides the number of mutation testing publications per year (years: 2008-2017). Furthermore, in Figure 2, Papadakos provides the number of

mutation publications by scientific conferences. Therefore, mutation testing remains one of the popular challenges and open problems for future work.

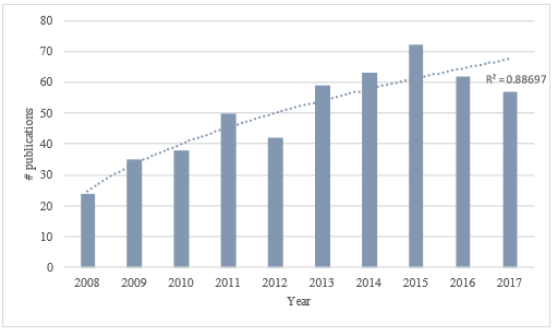

Figure 1.1 Number of mutation testing publications per year.

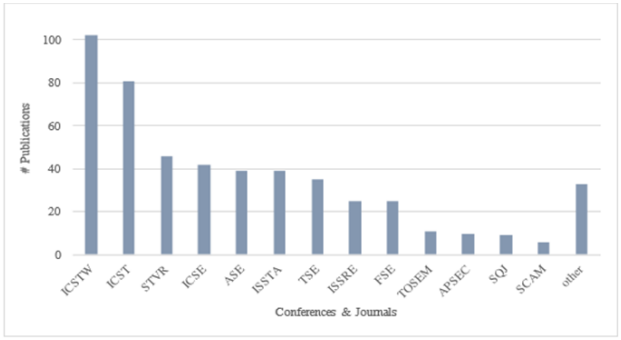

Figure 1.2 Number of mutation testing publications per scientific venue.

The are some applied mutation-based techniques that would support various software engineering approaches.

- Kaplan et al. [10] (2008) proposes mutant operators for UML domain models.
- El-Fakih et al. [11] (2008) uses mutation-based techniques to generate the test cases to propose Extended Finite State Machines (EFSMs).
- Trakhtenbrot [12] (2010) implements oriented mutation testing of state_chart models.
- Adra et al. [13] (2010) uses a mutation-based technique to test agent-based systems.

- Belli et al. [14] (2011) tests “go-back” functions, modelled by pushdown automata, by using a mutation-based technique.

- Aichernig et al. [15,16] (2011) presents the techniques and results of a novel model-based test case generation approach that automatically derives test cases from UML state machines

- Henard et al. tests [17] (2013) software product lines by using a mutation-based technique.

- Arcaini et al. [18-19] (2015) uses a mutation-based technique to assess fault detection capability of model review. Arcaini generates tests for detecting faults in feature mutants models.

- Filho et al. [20] (2016) proposes a multi-objective test data generation approach for mutation testing of feature models.

- Devroey et al. [21] (2016) presents featured models based on mutation that optimized generation, configuration and execution of mutants.

- Su et al. [22] (2017) implements stochastic model-based GUI testing of Android applications by using mutation-based techniques.

The syntax modification elementary that is applied on original program to generate mutants is called mutant operators. A set of basic mutant operators are introduced by Offut in Table 1.1. Selecting a variant set of mutant operators results in creating a different set of mutant instances [1].

Table 1.1 Popular Five-operator Set

Names	Description	Specific mutation operator
ABS	Absolute Value Insertion	$\{(e,0),(e,\texttt{abs}(e)),(e,\texttt{-abs}(e))\}$
AOR	Arithmetic Operator Replacement	$\{((\text{a } op \text{ b}),\ a),((\text{a } op \text{ b}),\ b),(x,y) \mid x,y \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\} \wedge x \neq y\}$
LCR	Logical Connector Replacement	$\{((\text{a } op \text{ b}),\ a),((\text{a } op \text{ b}),\ b),((\text{a } op \text{ b}),\ \texttt{false}),((\text{a } op \text{ b}),\ \texttt{true}),(x,y) \mid x,y \in \{\texttt{\&},\texttt{\|},\texttt{\^{}},\texttt{\&\&},\texttt{\|\|}\} \wedge x \neq y\}$
ROR	Relational Operator Replacement	$\{((\text{a } op \text{ b}),\ \texttt{false}),((\text{a } op \text{ b}),\ \texttt{true}),(x,y) \mid x,y \in \{\texttt{>},\texttt{>=},\texttt{<},\texttt{<=},\texttt{==},\texttt{!=}\} \wedge x \neq y\}$
UOI	Unary Operator Insertion	$\{(cond,\texttt{!}cond),(v,\texttt{-}v),(v,\texttt{\~{}}v),(v,\texttt{--}v),(v,v\texttt{--}),(v,\texttt{++}v),(v,v\texttt{++})\}$

Source: [23].

The test suite's effectiveness can be the measured by mutation score. The mutation score or mutation coverage is defined by the ratio of mutants that are killed by test suits to the total number of mutants [1]. The more mutants that are killed, the more effective the test suite is.

Redundant mutants are mutants that are killed, and they are semantically different from the original program, but they are equivalent to each other. The redundant mutants could distort the accuracy of mutant score criteria. Therefore, considering the mutation score for measuring test suite effectiveness is controversial [1].

1.2 Mutant Equivalence

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 [24-27]. But in practice, mutants may sometimes be semantically equivalent to the base program while being syntactically distinct from it [28-

34]. The issue of equivalent mutants has affected 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 [35]. 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 [30], 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 [36] 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 [37] and subsequently used in [38-39] as a means to assist in identifying equivalent mutants. In [40], Ellims et al. propose to help identify potentially equivalent mutants by analyzing the execution profiles of the mutant and the base program.

Howden [41] proposes to detect equivalent mutants by checking that a mutation preserves local states, and Schuler et al. [42] propose to detect equivalent mutants by

testing automatically generated invariant assertions produced by Daikon [43]; both the Howden approach and the Daikon approach rely on local conditions to determine equivalence, hence they are prone to generate sufficient but not necessary 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 [44], 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 [45] 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 [45] 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.

1.3 A Quantitative Approach

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 dissertation, we adopt a totally orthogonal approach to prior research, 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.

CHAPTER 2

BACKGROUND

2.1 Entropy of Random Variables

Our main source for this section is [47], to which the interested reader is referred, for further details. Given a variable X on a finite set X (by abuse of notation we use the name to represent the random variable and the set from which the random variable may take its

$$H(X) = -\sum_{i}^{n=1} \pi_X(X_i) \log\big(\pi X(x_i)\big), \tag{2.1}$$

values), we let the entropy of X be the following function:

where

- log is the base 2 logarithm,
- X = {x_1, x_2, x_3, ...x_N },
- P= πX (x_i) is the probability of the event: X = x_i.

We state without proof that H(X) ≥ 0; also, we take as a convention that the expression p*log(p) equals zero when p equals 0, hence we may apply the entropy function to probability distributions that are not necessarily non-zero for all x_i.

Intuitively, the entropy of random variable X represents the amount of uncertainty regarding the outcome of the random variable and takes its maximal value (which is log(n)) when all the outcomes are equally likely ($\pi_X(x_i)$ = 1/n for all i).

Given two random variables X and Y on sets X and Y, we define π_X and π_Y to be probability distributions of X and Y over their respective sets; we let π_{XY} be the probability distribution of the events ($X = x_i \wedge Y = y_j$) over the Cartesian Product $X \times Y$. Then we denote by H (X, Y) the entropy of the aggregate random variable (X, Y) over the set ($X \times Y$), and we refer to it as the *joint entropy* of X and Y. Using this definition, we let the *conditional entropy* of X with respect to Y be denoted by H(X|Y) and be defined as follows

$$H(X|Y) = H(X,Y) - H(Y) \tag{2.2}$$

Whereas the entropy of X represents the amounts of uncertainty about the outcome of X, the conditional entropy of X with respect to Y represents the amount of uncertainty about the outcome of X once we know the outcome of Y. We have an identity to the effect that the joint entropy of (X, Y) is greater than or equal to the entropy of Y, hence the conditional entropy is non-negative.

Given a random variable X that takes its values in some space S, and given a function G on X, we let Y be the random variable Y = G(X), whose probability distribution is derived from that of X, i.e.,

$$\pi_Y(\mathrm{Y} = \mathrm{y}) = \sum_{\forall \mathrm{x}:\mathrm{G}(\mathrm{x})=\mathrm{y}} \pi x(X = x) \tag{2.3}$$

Then, we have the inequality [47]: $H(X) \geq H(Y)$. In other words, applying a function to a random variable reduces its entropy (due to possible loss of information). If G is total and injective, then H(G(X)) = H(X).

To conclude this section, we introduce a concept that we use throughout the dissertation to assign intuitive interpretations to our metrics.

Definition 1 *We consider a set S and a predicate A on S, and we let S_A be the subset of S defined by elements of S that satisfy A(s). B is the bandwidth of assertion A is defined as $H(S) - H(S_A)$.*

Consider a set S defined by three integer variables, say x, y and z. Under the hypothesis of uniform probability distribution, and assuming that integers are represented by 32-bit words, the entropy of S is H(S)=32+32+32=96 bits. We consider a number of possible assertions, and compute their corresponding bandwidths:

- We define A(s) as x = y. Then space S_A is defined by variables y and z only. The entropy of S_A under the hypothesis of uniform probability distribution is $H(S_A)$ = 64 bits, which is the entropy of data type x and y. Hence the bandwidth of A is 32 bits, which is the width of the two expressions (x and y) involved in assertion A. In other words, it's B= 96-64=32 bits.

- We define A(s) as $x = z \wedge y = z$. Then space S_A can be defined by a single variable, say z. The entropy of S_A under the hypothesis of uniform probability distribution is $H(S_A)$ = 32 bits, hence the bandwidth of A is 64 bits, which is B=96-32, the combined width of the expressions that are involved in assertion A.

- We define A(s) as $x = 0 \wedge y = 10 \wedge z = 20$. $H(S_A)$=H(x=0)+H(y=0)+ H(z=20)= width of variable x+ width of variable y+ width of variable z=32+32+32=96. Hence the bandwidth of A is B=96-96=0 bits.

As another brief example, consider the binary representation of characters in a byte; seven bits out of eight are used to represent data, and the eighth bit is used for parity checking. We let S be the set of 8-bit patterns and we let A be the parity test, which can be written as parity (b1..b7) = b8.

The bandwidth of this assertion is $H(S) - H(S_A)$, which is 8−7 = 1 bit. Indeed, assertion A is an equality between two 1-bit expressions.

2.2 Fault, Error and Failure

Our main source for this section is [49], to which the interested reader is referred, for further details. We consider a program g on some space S, of the form

g = {g1; L: g2}; where g1 and g2 are subprograms and L is a label preceding g2.

We let R be a relation on S that represents the specification that g must meet, and we let s0 be an arbitrary initial state of g.

- A *fault* in program g is a feature of g that precludes it from satisfying its specification (in the sense of [50], for example).

- An *error* of the program at label L for initial state s0 is a state that is distinct from the expected state at this label; a fault of the program may or may not cause an error at label L, depending on the initial state s0; when a fault does cause an error, we say that it has been *sensitized* by the initial state s0.

- A *failure* of program g occurs whenever the error that arises at label L causes the program to fail to produce a correct (with respect to R) final state for initial state s0. An error at label L may cause a failure of the program, in which case we say that the error has been propagated; it may also cause no failure, in which case we say that the error has been *masked*.

We say that program g is *fault tolerant* if and only if it has provisions for avoiding failure after faults have caused errors. We consider three phases in the fault tolerance process:

- Error detection, when the program detects an inconsistency that indicates that the program state is erroneous.

- Damage assessment, when the program analyzes the current state to determine whether it is maskable (in which case recovery is unnecessary) or recoverable (in which case recovery is necessary and sufficient) or unrecoverable (in which case recovery is insufficient).

- Error recovery, when a recovery is invoked to map the recoverable state into a maskable state and let the computation resume from label L.

As an illustration, consider the space S defined by a natural variable, let the specification be relation R defined by R = {(s, s′)|s′ MOD 3 = s2 MOD 3},

The remainder of the division of s by 3 is the same as the remainder of the division of s2 by 3. We have chosen the example for the purpose of illustrating that when the program fails to produce the expected output, it may still be correct with respect to the specification.

let g be the program

g = {read(s); s=2*s; L: s = s mod 6; write(s);}

The intent of the programmer was for g to compute the following function:

$$G = \{(s, s')|s' = s2 \text{ MOD } 6\},$$

Which would have been correct with respect to R (in the sense of [51]), since G and R are both total, and $G \subseteq R$, as shown below:

$s' = s2 \text{ MOD } 6 \Rightarrow s' \text{ MOD } 3 = (s2 \text{ MOD } 6) \text{ MOD } 3 = s2 \text{ MOD } 3.$

But the programmer wrote the statement s = 2*s instead of the statement s=s*s, creating a fault. This fault may or may not be sensitized, depending on the input value.

- For $s_0 = 2$, the fault is not sensitized, since the expressions 2*s and s*s return the same value for s = 2.
- For $s_0 = 6$, the fault is sensitized, causing an error (s = 12 rather than s = 36 at label L), but the error is subsequently masked (since 12 mod 6 = 36 mod 6 at the end of the program).
- For $s_0 = 3$, the fault is sensitized, leading to an error (s =6 instead of s = 9 at label L); the error is subsequently propagated, causing a failure (s = 0 instead of s = 3 in the final state); in this instance, program g fails to behave according to its intended function G, but does not fail with respect to its specification R, since $s_0^2 \text{ mod } 3 = 9 \text{ mod } 3 = 0 = 0^2 \text{ mod } 3$; hence, strictly speaking, it satisfies its specification for $s_0 = 3$.

- Finally, for $s_0 = 4$, the fault is sensitized, leading to an error (the state at label L is s = 8 rather than s = 16); this error is propagated, leading to a final state that is distinct from the expected final state (the output is s = 2 rather than s = 4); this final state violates the specification, since 2 mod 3 ≠4 mod 3; in this case, the program failed to compute the expected final state, and also failed to satisfy the specification of the program.

The same fault may cause different chains of events, depending on the input. In order to be fault tolerant, a program must make provisions for error detection (to recognize when the potential of a failure may arise), error masking (to limit cases when recovery is necessary), and error recovery (to map a recoverable state into a maskable state, and let the computation proceed).

CHAPTER 3

SEMANTIC METRICS

3.1 Redundancy Metrics

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 this metric has an impact on the rate of equivalent mutants.
- How we compute this metric in practice.

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:

$$\frac{H(S) - H(\sigma)}{H(S)} \tag{3.1}$$

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.

$$SR_I = \frac{H(S) - H(\sigma_I)}{H(S)}, \tag{3.2}$$

$$SR_F = \frac{H(S) - H(\sigma_F)}{H(S)}, \tag{3.3}$$

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 (σI) and the entropy of the actual final state H(σ_F). For the entropy of the declared state, we simply add the entropies of the individual variable declarations, according to the Table 3.1 (for Java):

Table 3.1 Entropies of Basic Variable Declarations

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

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. We rely on preassert statement that the programmer may have included to specify the program's post-condition; we 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: We find:

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

- 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 when 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:

$$NI = \frac{H(\sigma_I|\sigma_F)}{H(\sigma_I)}. \qquad (3.4)$$

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:

$$NI = \frac{H(\sigma_I) - H(\sigma_F)}{H(\sigma_I)}. \tag{3.5}$$

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:

$$FR = \frac{H(X) - H(Y)}{H(X)} \tag{3.6}$$

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,

$$FR = \frac{63 - 32}{32} = 0.96875$$

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; x=1; x=2; y=3; z=x; x=y; y=z;}

P2: {int x,y,z; x=11;y=13; z=14; z=y; y=x; x=z;}

P3: {int x,y,z; x=10; y=20; z=20; 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 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:

$$ND = \frac{H(\sigma^M|\sigma^P)}{H(\sigma^M)} \tag{3.7}$$

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. Table 3.2 shows a few examples of equivalent oracles for the program.

Table 3.2 Non-Determinacy of Sample Oracles

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

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.

Table 3.3 Non-Determinacy of Sample Integer Oracles

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

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.

3.2 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 expectation is to reveal significant statistical relationships between the metrics (as independent variables) and the ratio of equivalent mutants (as a dependent variable).

Because we start computing the redundancy metrics by hand, we limit ourselves to programs that are relatively small. 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.

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 discussed 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 3.3, 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 3.3, and we take note of their non-determinacy. Below, Table 3.4 shows the raw data for our experiments.

Table 3.4 Raw data, REM vs Redundancy Metrics

Functions	LOC	Oracle	SRI	SRF	FR	NI	ND	COV	S/T	REM	log(REM/1-REM)
gcd	56	Equal	0.888693	0.924545	0.50	0.491	0		16/103	0.10526316	-0.929418926
		Eq%2	0.888693	0.924545	0.50	0.491	0.98438		22/103	0.21359223	-0.566062338
		Eq%4	0.888693	0.924545	0.50	0.491	0.95313	90%	19/103	0.18446602	-0.645525685
		Eq%16	0.888693	0.924545	0.50	0.491	0.9375		16/103	0.15533981	-0.73539927
mulAndCheck	42	Equal	0.861667	0.930833	0.50	0.43	0		6/43	0.13953488	-0.790050474
		Eq%2	0.861667	0.930833	0.50	0.43	0.98438	95%	6/43	0.13953488	-0.790050474
Fraction	68	Equal	0.88	0.961	0.33	0.66	0		22/95	0.23157895	-0.520900179
		dEq	0.88	0.961	0.33	0.66	0.5		23/95	0.24210526	-0.49560466
		dEq%2	0.88	0.961	0.33	0.66	0.84	96%	26/95	0.273	-0.425371764
getReducedFraction	26	Equal	0.86	0.98	1.00	0.77	0		17/46	0.37	-0.231138825
		dEq	0.86	0.98	1.00	0.77	1	96%	19/46	0.413	-0.15268805
erfInv	88	Equal	0.62	0.63	1.00	0.031	0	99%	9/126	0.071	-1.116757365
ebeDivide	20	Equal	0.897738	0.9	0.50	0.1	0	97%	1/13	0.077	-1.078710976
getDist	19	Equal	0.890208	0.940347	0.05	0.32	0	97%	1/17	0.059	-1.202737612
ArRealVec	12	Equal	0.901458	0.950729	0.90	0.48	0	97%	2/10	0.02	-1.69019608
ToBlocks	42	Equal	0.895669	0.903898	1.00	0.07887	0	95%	3/31	0.097	-0.968916016
getRowM	27	Equal	0.876503	0.948932	0.98	0.58648	0	95%	7/23	0.304	-0.359735656
orthogM	87	Equal	0.907995	0.933467	0.75	0.27685	0	100%	20/151	0.132	-0.817945794
Equals	31	Equal	0.851625	0.934625	0.20	0.55939	0	90%	6/21	0.286	-0.397332179
Density	18	Equal	0.883385	0.956771	0.25	0.23	0	95%	5/30	0.167	-0.69792853
Abs	20	Equal	0.89625	0.930833	0.50	0.33333	0	96%	2/20	0.1	-0.954242509
Pow	55	Equal	0.510214	0.61	0.67	0.19855	0	97%	6/52	0.115	-0.88624543
setSeed	17	Equal	0.80495	0.90455	1.00	0.51064	0	100%	4/16	0.25	-0.477121255
Asinh	17	Equal	0.897917	0.913542	1.00	0.15306	0	97%	13/82	0.159	-0.723398871
Atan	143	Equal	0.9	0.92	0.40	0.075	0	97%	14/136	0.103	-0.939955218
nextPrime(int n)	35	Equal	0.7925	0.89625	0.40	0.5	0		3/58	0.05	-1.278753601
		Eq%2	0.7925	0.89625	0.40	0.5	0.96	94%	34/58	0.58	0.140178703
Correlations1_logREM			0.111559	0.31138	0.096	0.58187	0.37524			1	
Correlations2_REM			0.018562	0.274208	0.072	0.60616	0.46913				

A. Statistical Analysis

Figure 3.1 represents 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.

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 dropped a covariate, each time testing the smaller model (one covariate less) with the previous model. We continued until the smaller model was significantly different, i.e., worse than the previous model. Using the procedure described above, we found that the final model contains the metrics FR, NI and ND, with coefficient estimates and standard errors given in the Table 3.5 below:

Table 3.5 Regression Model

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

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

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.

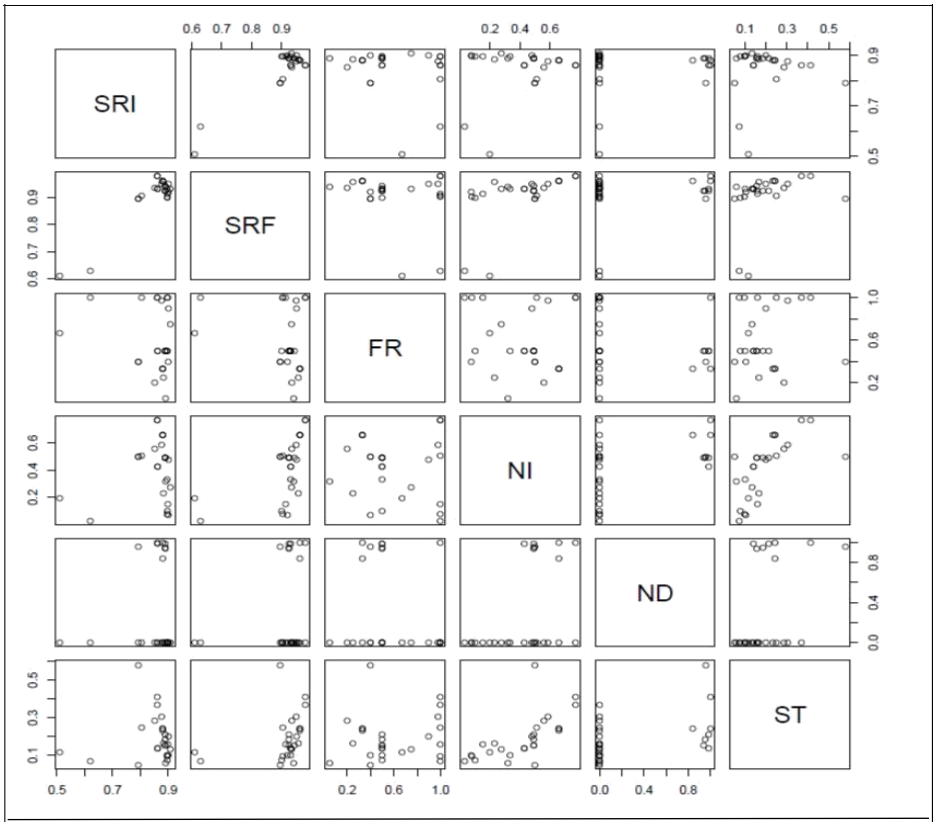

Figure 3.1 Scatter plot, redundancy metrics and ratio of equivalent mutants.

The sequence of models we tested, including their residual deviances, as well as the results of comparisons between them, are shown in the Table 3.6 below:

Table 3.6 Candidate Models

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 in Figure 3.2 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.

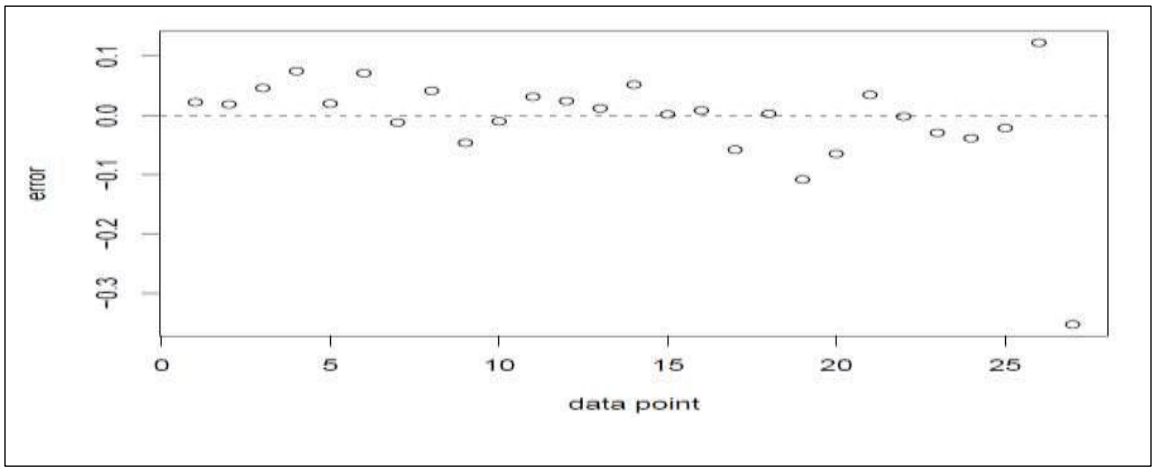

Figure 3.2 Residuals models.

CHAPTER 4

A JAVA COMPILER

4.1 Defining Elementary Metrics

In order to compute SRI, SRF, FR and NI, we have to derive these quantities for individual methods in Java classes. For each method, we must estimate the following quantities:

• The entropy of the declared space, H(S).

• The entropy of the initial actual space, $H(\sigma_I)$.

• The entropy of the final actual space, $H(\sigma_F)$.

• The entropy of the input space, H(X).

• The entropy of the output space, H(Y).

Therefore,

$$SR_I = \frac{H(S)-H(\sigma_I)}{H(S)} \quad (4.1)$$

$$SR_F = \frac{H(S)-H(\sigma_F)}{H(S)} \quad (4.2)$$

$$NI = \frac{H(\sigma_I)-H(\sigma_F)}{H(\sigma_I)} \quad (4.3)$$

$$FR = \frac{H(X)-H(Y)}{H(X)} \quad (4.4)$$

The entropies of the declared space, the input space, and output space are fairly straightforward; they consist in identifying the relevant variables and adding their respective entropies, depending on their data type, as per Table 3.1 For the entropy of the initial actual space, we are bound to rely on input from the source code, as we have no

other means to probe the intent of the programmer (re: how they use declared variables to represent the actual program state). To this effect, we introduce a special purpose assert statement, which the engineer may use to specify the precondition of the method whose REM we want to compute. We propose the following statement preassert(<precondition>) whose semantic definition is exactly the same as a normal assert statement, but this one is used specifically to analyze the entropy of the initial actual state. When the method has an exception call at the beginning as a guard for the method call, then it is straightforward to have a preassert() statement immediately after the exception statement, with the negation of the condition that triggers the exception.

The entropy of the initial actual state is computed as: $H(\sigma_I) = H(S) - \Delta H$, where ΔH is the reduction in entropy represented by the assertion of the preassert() statement. This quantity is defined inductively according to the structure of the assertion, as shown summarily below:

- $\Delta H(A \wedge B) = \Delta H(A) + \Delta H(B)$.
- $\Delta H(A \vee B) = \max(\Delta H(A), \Delta H(B))$.
- $\Delta H(X == Y)$, where X and Y are expressions of the same type, equals the entropy of the common type. For example, if x and y are integer variables, then $\Delta H(x+1 == y-1)$ is 32 bits.
- $\Delta H(X < Y) = \Delta H(X <= Y) = \Delta H(X > Y) = \Delta H(X >= Y) = 1$ bit. So for example $\Delta H(x+1 > 0) = 1$ bit, since this inequality reduces the range of possible values of x by half, whose $\log_2$ is then reduced by 1.

This is not a perfect solution, but it is adequate for our purposes. For the entropy of the final actual space, we must keep track of dependencies that the program creates between

its variables. We do so using a Boolean matrix (called D, for Dependency),which is initialized to the identity (T on the diagonal, F outside, to mean that initially each variable depends only on itself); whenever we encounter an assignment statement, of the form (x=E(y,z,u,v)), we replace the row of x in D with the logical OR of the rows of all the variables that appear in expression E. At the end of the program we add (i.e. take the logical OR) of all the rows of the matrix; this yields a vector that indicates which program variables affect the value of the final state of the program. The sum of the entropies of the selected variables is the entropy of the final actual state. If the assignment statement is embedded within an if-statement, an if-then-else statement or a while loop, then the variables that appear in the condition of the if or while are added to the variables that are on the right-hand side of the assignment, since they affect the value of the assigned variable. For example, consider the following example:

```
if (x>10)
y=z+g+3;
else
y=k+5;
```

We analysis if-part and else-part separately. We consider (y=F(x,z,g)) for if-part and (y=K(x,k)) for else-part. We replace the row of y in D with the logical OR of the rows of x, z, and g then we assign the effect to matrix D1. We replace the row of y in D with the logical OR of the rows of x and k, then we assign the effect to matrix D2. Then we find sum of all rows of D1 and D2, then the matrix that is given the minimum is assigned to the resulting matrix.

Consider the following example:

```
int x, y, z; z=10;
z=10;
x=y+z;
y=2*x+15*z;
```

Since we have three variables, we have 3 columns represent each variable in matrix D. When the variable is declared, a row is assigned to that variable. Therefore, the first row represents x, the second row represents y, the third row represents z.

The sequence of matrix D is shown below

Initial matrix is

T F F
F T F
F F T

When z=10, the matrix D becomes

T F F
F T F
F F F

When x = y+z, the matrix D becomes

F T F
F T F
F F F

When y = 2*x+15*z; the matrix D becomes

F T F
F T F
F F F

The sum of the rows is the vector F T F which means, 0*32 for x + 1* 32 for y + 0*32 for z. The total is 32 bits.

4.2 An ANTLR-Generated Compiler

In our research, we use ANTLR (**An**other **T**ool for **L**anguage **R**ecognition http://www.antlr.org/) to generate a compiler for different programming languages such as Java, C#, JavaScript, Python2, and Python3. The initial release of ANTLR was on February 1992 by Dr. Terence Parr at University of San Francisco.

ANTLR takes as input a grammar that specifies a language and generates output as source code for a recognizer for that language. It also automatically reports and recovers from syntax errors.

ANTLR is a recursive descent parser generator. It uses the top-down parsing strategy LL (*) for parsing. LL (*) is an LL-regular parser if it is not restricted to a finite k token of lookahead but can make parsing decisions by recognizing whether the following tokens belong to a regular language.

We use ANTLR v4 which is the latest version of ANTLR. ANTLR v4 automatically rewrites left-recursive rules such as expression into non left-recursive equivalents. it dramatically simplifies the grammar rules used to match syntactic structures.

How to set up ANTLR?

- We download https://www.antlr.org/download/antlr-4.7.2-complete.jar, add antlr4-complete.jar to CLASSPATH of our environment system variables.
- We create the following batch commands: *antlr4.bat* has the command *java org.antlr.v4.Tool %** and *grun.bat* has the command *java org.antlr.v4.gui.TestRig %**

ANTLR grammar

The grammar of ANTLR must be of extension g4. ANTLR provides grammar specification (https://github.com/antlr/grammars-v4) for some programming languages. We use java9.g4 to generate our compiler.

How does ANTLR work?

ANTLR first checks the specification of Java grammar, rules and actions and generates some of Java classes. Figure 4.1 shows flowchart of ANTLR run. If the rules are successfully built, the ANTLR generates the following:

- Java9Lexer.java
- Java9Parser.java
- Java9.tokens
- Java9Lexer.tokens
- Java9Listener.java
- Java9BaseListener.java

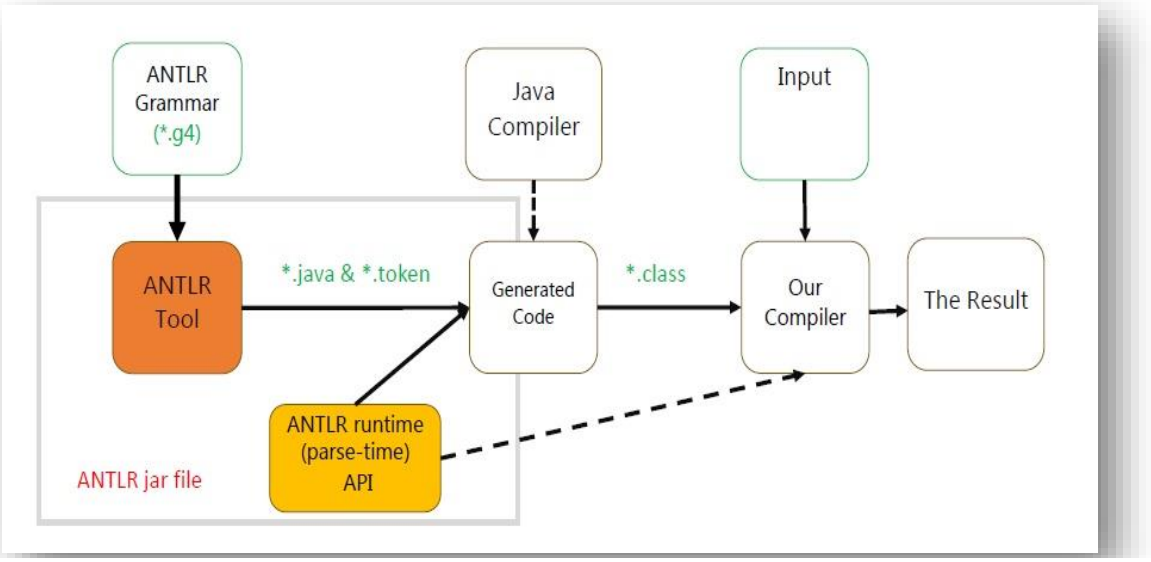

Figure 4.1 Flowchart of ANTLR run.

How to use ANTLR?

We have to create the object of lexer and parser, then we run the command *parser.compilationUnit();* this command runs our grammar starting from start symbol which is *compilationUnit* to the end of the grammar. Figure 4.2 shows our main method we consider data default size is 6. Each time we add the semantic actions to our grammar, we have to run our grammar then the example. Figure 4.3 illustrates the steps of our run.

```
public static void main (String[]args) throws IOException{

AntlrParse myInstance = new AntlrParse();
defaultArrSize=6;
strDefaultSize=8*6;
defaultIterator=2;
ANTLRInputStream input = new ANTLRFileStream(args[0]);
Java9Lexer  lexer = new Java9Lexer(input);
CommonTokenStream commonTokenStream=new CommonTokenStream(lexer);
Java9Parser parser = new Java9Parser(commonTokenStream);
ParseTree  parseTree = parser.compilationUnit();
```

Figure 4.2 Run of the main method in ANTLR.

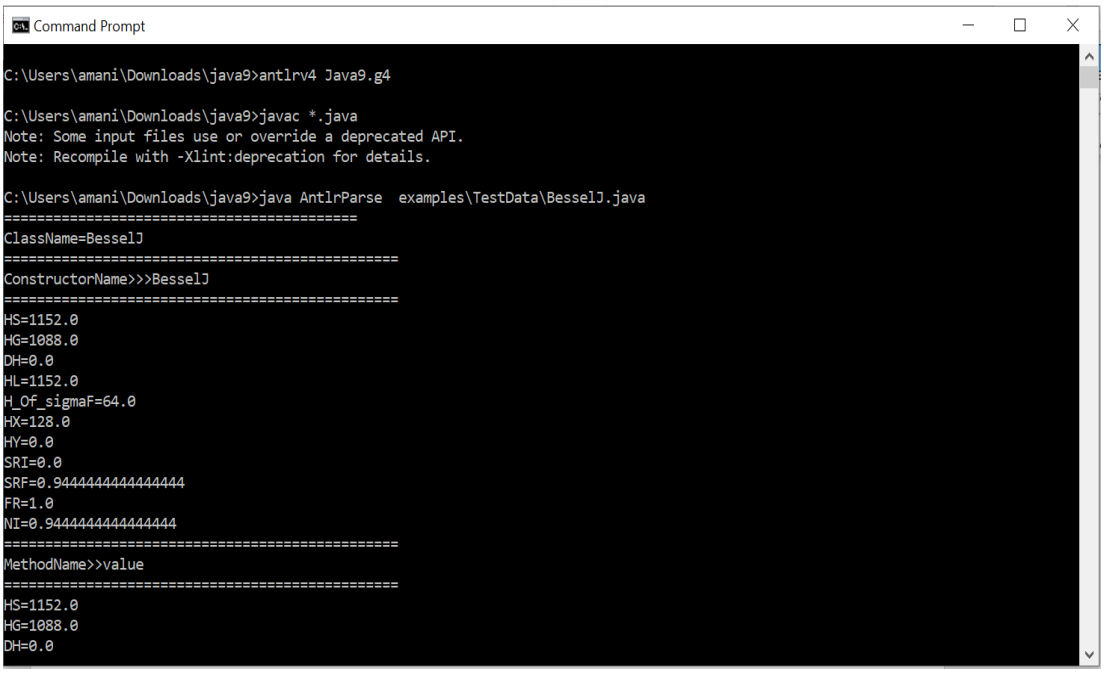

Figure 4.3 ANTLR run.

ANTLR can generate a parse tree that helps us to debug our semantic actions. Figure 4.4 shows our commands that generates the parse tree and Figure 4.5 shows parser tree inspector that the result of the run.

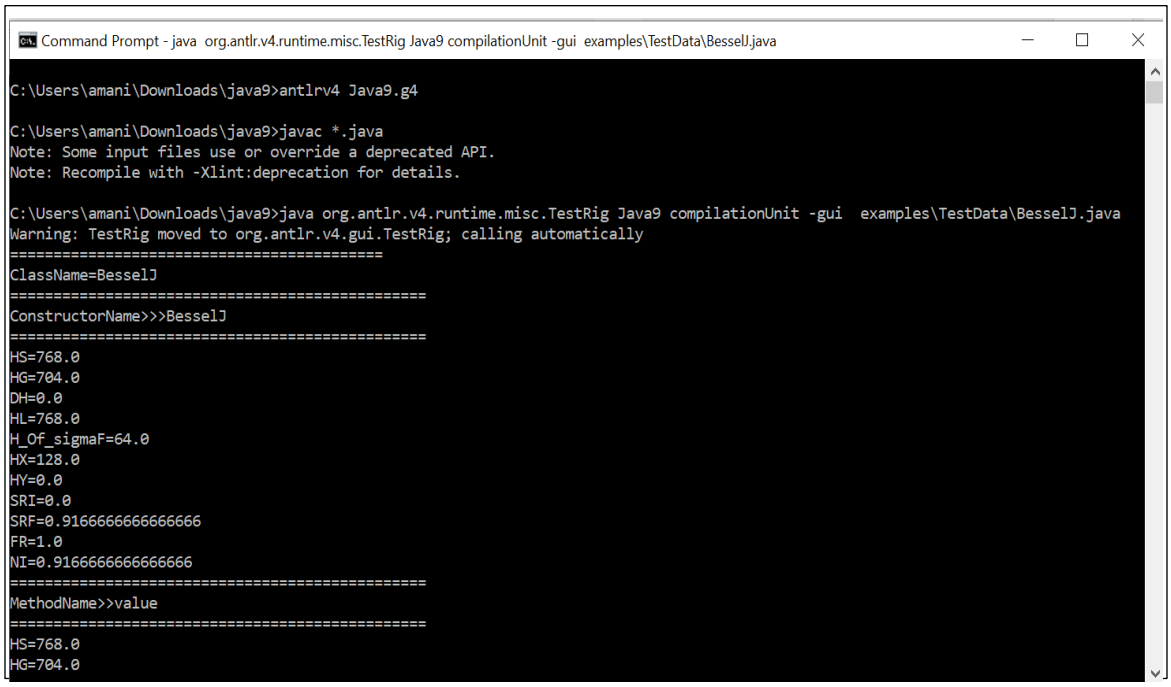

Figure 4.4 ANTLR run for parser tree inspector.

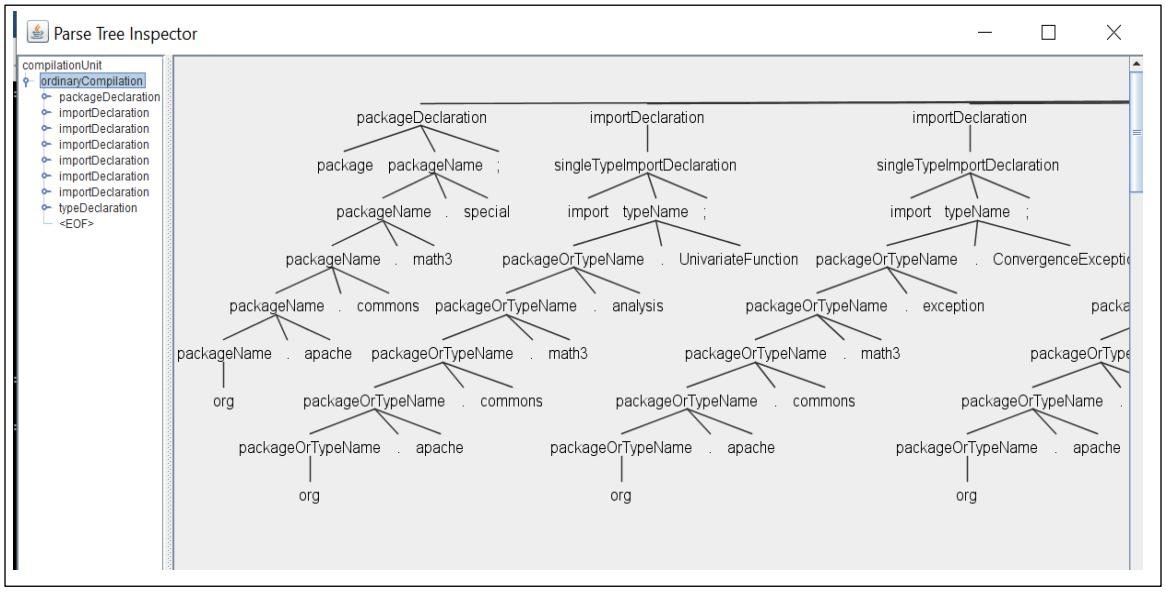

Figure 4.5 Parser tree inspector for ANTLR.

4.3 Computing the Elementary Metrics

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

Calculating H(S)

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

- H(G): Entropy of the global space, i.e. the space defined by the declared fields of the class; these are class wide variables that are accessible to all the methods of the class.
- H(P): Entropy of the space defined by the parameters that are passed to the method.
- H(I): Entropy of the local space, i.e. the space defined within the scope of the method.

Therefore, H(S) = H(G) + H(P) + H(I).

Calculating H(G)

H(G) is the entropy of global variables. It includes all variables that are declared within the class header and before the method header. We use a hashmap named *mapGlobalVar* to store all global variables. Global variables appear in the *fieldDeclaration* rule. The following is the *fieldDeclaration* rule before adding the semantic actions. Figures 4.6 and 4.7 show the rule after we add the semantic actions.

fieldDeclaration rule

fieldDeclaration :fieldDeclaration :fieldModifier unannType variableDeclaratorList ';' ;*

```
fieldDeclaration
   :  {variableList=new ArrayList<String>();}
   fieldModifier* unannType variableDeclaratorList ';' {
   if (unannArrayTypeflag==1  & dimvalue==0 & typeInterfaceFlag==0 ){
        arrayData =new ArrayList<String>();arrayData.add(arrayTypevalue);
        if (d==2){d=d*AntlrParse.defaultArrSize;arrayData.add(String.valueOf(d));
        else
            arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
            mapGlobalVar.put(arrayName,arrayData);
            IndMxALL=testMx.M_IND(arrayName,IndMxALL);
            ////////////////////////////////check if we have an assignment
            if ( ifElse==0 & ifOnly==0 & assF==1  ){
                    IndMxALL=testMx.MU(IndMxALL, leftElm,arrayDataInd01);
                    arrayDataInd01=new ArrayList<String>();assF=0;   }
                    unannArrayTypeflag=0;arrayTypevalue="";arrayName="";d=0;
                     arrayData =new ArrayList<String>();}
      /*   in case of arrays  decalred and outside any block  ex: double x[]
       and no created stat */
      else if (unannArrayTypeflag==0 & dimvalue==1 & typeInterfaceFlag==0){
      arrayData =new ArrayList<String>();
       arrayData.add($unannType.text);
      if (d==2)
      {


    d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
    arrayData.add(String.valueOf(d));
    }
    else arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
    mapGlobalVar.put(arrayName,arrayData);
    IndMxALL=testMx.M_IND(arrayName,IndMxALL);
 ////////////////////////////////check if we have an assignment
 if ( ifElse==0 & ifOnly==0& assF==1 ){
            IndMxALL=testMx.MU(IndMxALL, leftElm,arrayDataInd01);
            arrayDataInd01=new ArrayList<String>();
            assF=0;  }
     arrayName="";unannArrayTypeflag=0;dimvalue=0;arrayTypevalue="";d=0;
    arrayData =new ArrayList<String>();arraySize=AntlrParse.defaultArrSize;}
    ////////////////////////////////////////////////////////////////////////////
   else if (dimvalue==0 & unannArrayTypeflag==0 & typeInterfaceFlag==0) {
             arrayData= new ArrayList<String>();
             arrayData.add($unannType.text);
             for (int i=0;i<variableList.size();i++){
                          mapGlobalVar.put(variableList.get(i),arrayData);
                         IndMxALL=testMx.M_IND(variableList.get(i),IndMxALL);
                         }
               variableList=new ArrayList<String>();
 ////////////////////////////////check if we have an assignment
```

Figure 4.6 Sematic actions for fieldDeclaration rule.

```
 if ( ifElse==0 & ifOnly==0& assF==1 ){
        IndMxALL=testMx.MU(IndMxALL, leftElm,arrayDataInd01);
        arrayDataInd01=new ArrayList<String>(); assF=0;  }
        variableName="";}
else if (dimvalue==0 & unannArrayTypeflag==0 & typeInterfaceFlag==1) {
               arrayData= new ArrayList<String>();
               arrayData.add(typeInterface);
               mapGlobalVar.put(variableName,arrayData);
               IndMxALL=testMx.M_IND(variableName,IndMxALL);
               for (int i=0;i<variableList.size();i++){
                         mapGlobalVar.put(variableList.get(i),arrayData);
                         IndMxALL=testMx.M_IND(variableList.get(i),IndMxALL);
                         }
                       typeInterfaceFlag=0; variableName=""; typeInterface="";
                        }
             }
  ;
```

Figure 4.7 Sematic actions for fieldDeclaration rule.

Sometimes we have variables or arrays of type object class. So, we must store the class name and its entropy inside *mapGlobalVar*. We add our semantic actions into the *normalClassDeclaration* rule. The following is the rule without the semantic actions and Figure 4.8 shows the rule after we add the semantic actions.

normalClassDeclaration: classModifier 'class' Identifier typeParameters? superclass? superinterfaces? classBody ;*

The rule gives the structure of declaration class. *Class Modifier* can be one of the following keywords: annotation, public, protected, private, abstract, static, final, or strictfp.

Identifier represents the name of the class. *typeParameters* is type of class and it's optional.

superclass and *superinterfaces* are optional, they represent the inheritance feature. *classBody* is the body of the whole class.

```
normalClassDeclaration: {dimvalue=0;unannArrayTypeflag=0;localDec=0;
localArgument=0;blockfalg=0; elseCond=0;ifElse=0;ifName=0;ifMul=0;
AfterAssSign=0; ifOnly=0;}
classModifier* 'class' Identifier {
if (classF==0){classF=1;cNAME=$Identifier.text;}
else {if (!classEntopy.containsKey(cNAME)){
                                        classEntopy.put(cNAME,HG);
                                        AntlrParse.classEntopy1=classEntopy;
                                  }
      System.out.println("HG for class>>"+cNAME+">>"+HG);
      cNAME=$Identifier.text;}
      System.out.println("========================================");
      System.out.println("ClassName="+$Identifier.text);}
      typeParameters? superclass? superinterfaces?  classBody
      {if (!classEntopy.containsKey(cNAME)){
                      classEntopy.put(cNAME,HG);
                      AntlrParse.classEntopy1=classEntopy;}
        };
```

Figure 4.8 Sematic actions for normalClassDeclaration rule.

Also, our compiler processes the enum structure in Java. *mapGlobalVar* keeps the entropy of variables that appear in enum structure. The following is the rule without the semantic actions and Figure 4.9 illustrates the rule after we add the semantic actions.

classDeclaration : normalClassDeclaration

| enumDeclaration ;

```
      classDeclaration
         :  normalClassDeclaration
         |  enumDeclaration   {
         ansEnum=Math.log(enumConstantNum);

      if (!classEntopy.containsKey(enumConstantName1))
         classEntopy.put(enumConstantName1,ansEnum);

               enumFlag=1;
               myInstance.enumf=enumFlag;
               enumFlag=0;
               myInstance.enumf=enumFlag;
            }
      ;// end of rule
```

Figure 4.9 Sematic actions for normalClassDeclaration rule.

Calculating H(P)

H(P) includes all passing parameters in a method header such as variables or arrays. We create a hashmap named ele_HX. We calculate H(P) by adding our semantics actions into *formalParameter* rule and *lastFormalParameter* rule. The following is *formalParameter* rule and *lastFormalParameter* without adding the semantic actions. Figure 4.10 shows the *formalParameter* rule after we add our semantic actions. Figure 4.11 shows the *lastFormalParameter* rule after we add the semantic actions.

```
formalParameter
   :  variableModifier* unannType variableDeclaratorId  {
   /*   in case of arrays  decalred   ex: double []x  */
if (unannArrayTypeflag==1 & dimvalue==0 ){
       arrayData =new ArrayList<String>();
       arrayData.add(arrayTypevalue);
      if (d==2)
      {
      d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
      arrayData.add(String.valueOf(d));
      }
      else
       arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
       ele_HX.put(arrayName,arrayData);
       IndMxALL=testMx.M_IND(arrayName,IndMxALL);
       unannArrayTypeflag=0;arrayTypevalue="";arraySize=1;
       arrayName="";  arrayData =new ArrayList<String>(); d=0;}
      /*   in case of arrays  decalred   ex: double x[]   */
      else if (unannArrayTypeflag==0 & dimvalue==1){
      arrayData =new ArrayList<String>();


       arrayData.add($unannType.text);
      if (d==2)
      {
      d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
      arrayData.add(String.valueOf(d));
      }
      else
        arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
        ele_HX.put(arrayName,arrayData);
        IndMxALL=testMx.M_IND(arrayName,IndMxALL);
        dimvalue=0;arrayTypevalue="";d=0;
       arrayName="";  arrayData =new ArrayList<String>();}
    else if (dimvalue==0 & unannArrayTypeflag==0) {
           arrayData= new ArrayList<String>();
           arrayData.add($unannType.text);
           IndMxALL=testMx.M_IND(variableName,IndMxALL);
           ele_HX.put(variableName,arrayData);
           variableName="";arrayTypevalue="";}
   }
   ;// end of the rule
```

Figure 4.10 Sematic actions for formalParameter rule.

```
lastFormalParameter
    :  variableModifier* unannType annotation* '...' variableDeclaratorId
    {if (unannArrayTypeflag==1 & dimvalue==0 ){
       arrayData =new ArrayList<String>();
       arrayData.add(arrayTypevalue);
       if (d==2)
       {d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
        arrayData.add(String.valueOf(d));
        }
        else
             arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
        ele_HX.put(arrayName,arrayData);
        IndMxALL=testMx.M_IND(arrayName,IndMxALL);
        unannArrayTypeflag=0;arrayTypevalue="";arraySize=1;
        arrayName="";  arrayData =new ArrayList<String>(); d=0;
        }
       /*   in case of arrays  decalred   ex: double x[]   */
       else if (unannArrayTypeflag==0 & dimvalue==1){
       arrayData =new ArrayList<String>();
        arrayData.add($unannType.text);



        if (d==2)
        {
        d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
        arrayData.add(String.valueOf(d));
        }
        else arrayData.add(String.valueOf(AntlrParse.defaultArrSize));
           ele_HX.put(arrayName,arrayData);
           IndMxALL=testMx.M_IND(arrayName,IndMxALL);
           dimvalue=0;arrayTypevalue="";d=0;
          arrayName="";  arrayData =new ArrayList<String>();}
      else if (dimvalue==0 & unannArrayTypeflag==0) {
           arrayData= new ArrayList<String>();
           arrayData.add($unannType.text);
    IndMxALL=testMx.M_IND(variableName,IndMxALL);
    ele_HX.put(variableName,arrayData);
    variableName="";arrayTypevalue="";}
    }

    |  formalParameter
    ;
```

Figure 4.11 Sematic actions for lastFormalParameter rule.

Calculating H(X)

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

entropies of all these variables. We add the semantic rules into *assignment rule.* The following is the *assignment* rule before adding semantic actions. The semantic actions of *assignment* rule can be found in Figure 4.12.

assignment:leftHandSide assignmentOperator expression;

leftHandSide can be expressionName, fieldAccess or arrayAccess.

assignmentOperator can be one of the following keywords : '=', '*=', '/=', '%=', '+=', '-=', '<<=', '>>=', '>>>=', '&=', '^=', or '|='

```
assignment
   :  {assF=1;
    arrayData1=new ArrayList<String>();
    arrayDataInd01=new ArrayList<String>();
   }// end of action
   leftHandSide assignmentOperator  // rule
   {//start the action
   AfterAssSign=1;}
   expression
   {// start the action
   if (localArgument==1)
   {arrayDataInd01.addAll(arrayData1); arrayData1=new ArrayList<String>();
   localArgument=0;
   }
  /// condition for for and while
if (!CondForWhilestack.empty()){for(int i=CondForWhilestack.size()-1; i>=0;i--)
                               arrayDataInd01.addAll(CondForWhilestack.get(i));
                               }
    // condition for switch
   if (!CondSwitch.empty()){for(int i=CondSwitch.size()-1; i>=0;i--)
                                  arrayDataInd01.addAll(CondSwitch.get(i));
                                }
      // we adding the conditions in stack
    if(ifOnly==1 | ifElse==1 )
    {// get all variabls in stack conditions
      if (!stack.empty()){
              for(int i=stack.size()-1; i>=0;i--)
                                  arrayDataInd01.addAll(stack.get(i));
                               }
                           }
logic1= ifElse==0 ;
     if ( (logic1&ifOnly==1) | (logic1&ifOnly==0) ){
    // incase i have only if or  stat located outside of if
     IndMxALL=testMx.MU(IndMxALL, leftElm,arrayDataInd01);
   if (! HY_paraLeft.contains(leftElm))
         HY_paraLeft.add(leftElm);

for (int i=0;i<arrayDataInd01.size();i++)
        if (!hx.contains(arrayDataInd01.get(i)))
             hx.add(arrayDataInd01.get(i));
arrayDataInd01=new ArrayList<String>();assF=0;ifMul=1; }
 else if(ifElse==1 & elseCond==0  ){
 IndMxIf=testMx.MU(IndMxIf, leftElm,arrayDataInd01);
if (! HY_paraLeft.contains(leftElm))
         HY_paraLeft.add(leftElm);
for (int i=0;i<arrayDataInd01.size();i++)
        if (!hx.contains(arrayDataInd01.get(i)))
             hx.add(arrayDataInd01.get(i));
arrayDataInd01=new ArrayList<String>();
    assF=0;  ifMul=1;
 }
else  if (ifElse==1 & elseCond==1 ) {
IndMxElse=testMx.MU(IndMxElse, leftElm,arrayDataInd01);
   if (! HY_paraLeft.contains(leftElm))
         HY_paraLeft.add(leftElm);

for (int i=0;i<arrayDataInd01.size();i++)
        if (!hx.contains(arrayDataInd01.get(i)))
             hx.add(arrayDataInd01.get(i));
   arrayDataInd01=new ArrayList<String>();assF=0;ifMul=1;
}
AfterAssSign=0;
}

    ;

leftHandSide
   :  expressionName    {if (assF==1 ) leftElm=$expressionName.text; }
   |  fieldAccess
   |  arrayAccess
   ;
```

Figure 4.12 Sematic actions for assignment rule.

Calculating H(Y)

H(Y) is the output channel of a method and depends on whether the method is declared as void or has an explicitly declared return type:

- If the method has an explicitly declared return type, then the entropy of that type is the value for the output channel entropy. Figure 4.13 shows the *methodHeader* rule and the semantic actions added to the *methodHeader* rule.

- If the method is declared as a void method, then the output channel is made up of the following components: the parameters of type class (which are implicitly passed by reference); the global variables that appear on the left of an assignment statement.

To compute the entropy of the output channel, we use the dependency matrix D introduced in Section 4.1 whereas the entropy of the final state is computed by adding all the rows of D, the entropy of H (Y) is computed by adding the rows of D that correspond to the output variables cited above.

```
methodHeader
        :       result methodDeclarator throws_?
        |       typeParameters annotation* result methodDeclarator throws_?
        ;
result  : unannType
        |'void'
```

```
        ;
        methodHeader
           :  result methodDeclarator throws_?
           |  typeParameters annotation* result methodDeclarator throws_?
           ;
        result
           :  unannType   {
           if (unannArrayTypeflag==0) {
           returnTypeMethod.add($unannType.text);
        H_ReturnType=myInstance.findType(returnTypeMethod.get(0),AntlrParse.defaultArrS:
           }
           else{if (d==2)
              {d=AntlrParse.defaultArrSize*AntlrParse.defaultArrSize;
              H_ReturnType=d;
                    }else  H_ReturnType=AntlrParse.defaultArrSize;
              unannArrayTypeflag=0;
             arrayTypevalue="";}
            }
           |{MheaderV=1;} 'void'
           ;
```

Figure 4.13 Sematic actions for methodHeader rule.

CHAPTER 5

STATISTICAL MODELS

5.1 Benchmark

In our experiment, we use benchmark from the Apache Common Mathematics Library (http://apache.org/); each function comes with a test data file. We run our experiments on two different packages of Java classes. They are:

1. The Apache Commons Math project is a library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language or commons-lang. The version that we use is commons-math3-3.5-src. Table 5.1 shows the class name, number of the methods, and its directory.

Table 5.1 Classes Information of Commons-math3-3.5-src Library

Class Name	Number of the methods	Class Directory
SchurTransformer	10	org.apache.commons.math3.linear
BesselJ	7	org.apache.commons.math3.special
BlockRealMatrix	54	org.apache.commons.math3.linear
EigenDecomposition	27	org.apache.commons.math3.linear
Array2DRowRealMatrix	31	org.apache.commons.math3.linear
CholeskyDecomposition	11	org.apache.commons.math3.linear
BaseSecantSolver	7	org.apache.commons.analysis.solvers

2. Apache Commons Lang is a package of Java utility classes for the classes that are in java.lang's hierarchy, or are considered to be so standard as to justify existence in java.lang. The version that we use is commons-lang3-3.4-src. Table 5.2 shows the class name, number of the methods at that class, and its directory.

Table 5.2 Classes Information of Commons-lang3-3.4-src Library

Class Name	Number of the methods	Class Directory
Fraction	34	org.apache.commons.lang3.math
NumericEntityUnescaper	3	org.apache.commons.lang3.text.translate
WordUtils	13	org.apache.commons.lang3.text
NumberUtils	55	org.apache.commons.lang3.math
FastMath	26	org.apache.commons.lang3.math

5.2 Mutation Generators

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

- Either we select several well-known, widely used and / or widely researched generation policies, and generate a regression model for each. Then our tool offers the user a menu of policies and asks the user to select one; then the tool uses the corresponding regression formula.
- Or we select a number of well-known mutation operators, generate a regression for each mutator applied individually. Then our tool offers the user a menu of

operators and asks him / her to select all those she / he wishes to apply. Then the tool estimates the REM that stems from each mutator; but then it needs to combine the individual REM's to estimate the overall REM obtained by combining the mutation operators. This approach raises the question of how we combine individual REM's corresponding to single mutators to obtain the REM of the aggregate policy. In [51] we speculate that the following formula is a good approximation, and we provide some empirical evidence to this effect:

$$REM = 1 - \prod_{i=1}^{N}(1 - REM_i) \quad (5.1)$$

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

We use two different Mutation Generators: PITEST and Mujava. We divide the mutation operators into four classes. Class1, 2, and 4 have mutation operators are generated by PITEST. Class 3 has mutation operators are generated by Mujava.

A. PITEST

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.

Class 1:

a. Conditionals Boundary Mutator
b. Arithmetic Operator Replacement Mutator
c. Arithmetic Operator Deletion Mutator
d. Constant Replacement Mutator
e. Relational Operator Replacement Mutator

Class 2:

a. Constructor Call Mutator
b. Empty returns Mutator
c. False returns Mutator
d. Inline Constant Mutator
e. Null returns Mutator
f. Non-Void Method Call Mutator

g. Primitive returns Mutator
h. Remove Conditionals Mutator
i. Remove Increments Mutator
j. True returns Mutator
k. Experimental Argument Propagation
l. Experimental Big Integer
m. Experimental Naked Receiver
n. Experimental Member Variable Mutator
o. Experimental Switch Mutator
p. Negation Mutator
q. BitWise Operator
r. Unary Operator Insertion

Class 4:

a. Conditionals Boundary Mutator
b. Increments Mutator
c. Void Method Call Mutator
d. Return Values Mutator
e. Math Mutator
f. Negate Conditionals Mutator
g. Invert Negatives Mutator

B. MuJava

Mujava (https://cs.gmu.edu/~offutt/mujava/) which is a mutation system for Java programs. Class 3 has mutation operators that are generated by Mujava. Mujava provides six kinds of primitive operators: arithmetic, relational, conditional, shift, logical, and assignment. The Table 5.3 below shows the list of method-level mutation operators with its description.

Table 5.3 Method-level Mutation Operators in Mujava

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement
Deletion operator added in 2013	
SDL	Statement Deletion
VDL	Variable Deletion
CDL	Constant Deletion
ODL	Operator Deletion

5.3 Redundancy Metrices by the Compiler

The following is the data generated by the compiler when the data default size for array and string is equal to 10. Due to restricted space, we view only data that has LOC (lines of code) >=40.

Table 5.4 Raw data, independent variables, redundancy metrics, vs REM

Class Name	Method Name	LOC	HS	HG	DH	HL	H sigma_F	HX	HY	SRI	SRF	FR	REMP
BlockRealMatrix	preMultiply	41	9024.00	6624.00	66.00	8958.00	576.00	900.00	10	0.0073	0.9362	0.9889	0.0000
BlockRealMatrix	toBlocksLayout	42	20608.00	6624.00	4.00	20604.00	960.00	6760.00	100	0.0002	0.9534	0.9852	0.0968
CholeskyDecomposition/Solver	solve	43	10080.00	1408.00	64.00	10016.00	512.00	20.00	0.00	0.0063	0.9492	1.0000	0.0294
FastMath	cos	45	21985	21121	0	21985.00	64	64	64	0.0000	0.9971	0.0000	0.2500
BlockRealMatrix	setSubMatrix	49	14944.00	6624.00	4.00	14940.00	1344.00	6628.00	0	0.0003	0.9101	1.0000	0.0000
Fraction	getFraction	49	3072.00	2208.00	128.00	2944.00	192.00	64.00	64.00	0.0417	0.9375	0.0000	0.3000
BlockRealMatrix	multiply	51	8544.00	6624.00	32.00	8512.00	384.00	196.00	32	0.0037	0.9551	0.8367	0.0270
CholeskyDecomposition	CholeskyDecomposition	52	4512.00	1408.00	0.00	4512.00	128.00	148.00	0.00	0.0000	0.9716	1.0000	0.1071
FastMath	pow	55	22337	21121	0	22337.00	96	96	64	0.0000	0.9957	0.3333	0.1250
WordUtils	wrap	57	1409.00	0.00	0.00	1409.00	1313.00	1313.00	640.00	0.0000	0.0681	0.5126	0.1600
FastMath	sin	58	22050	21121	0	22050.00	64	64	64	0.0000	0.9971	0.0000	0.2105
BlockRealMatrix	multiply	59	9248.00	6624.00	66.00	9182.00	416.00	260.00	32	0.0071	0.9550	0.8769	0.0000
NumericEntityUnescaper	translate	61	339.10	1.10	2.00	337.10	33.10	32.00	32.00	0.0059	0.9024	0.0000	0.0417
FastMath	cosh	62	22465	21121	0	22465.00	0	192	64	0.0000	1.0000	0.6667	0.4118
FastMath	tan	68	22370	21121	0	22370.00	64	64	64	0.0000	0.9971	0.0000	0.2581
FastMath	asin	69	21953	21121	0	21953.00	64	64	64	0.0000	0.9971	0.0000	0.0000
FastMath	scalb	73	21537	21121	0	21537.00	96	96	64	0.0000	0.9955	0.3333	0.2000
FastMath	scalb	73	21377	21121	0	21377.00	64	64	32	0.0000	0.9970	0.5000	0.1774
FastMath	acos	75	21889	21121	0	21889.00	64	64	64	0.0000	0.9971	0.0000	0.0267
FastMath	cbrt	76	22082	21121	0	22082.00	0	74	64	0.0000	1.0000	0.1351	0.0462
BlockRealMatrix	getSubMatrix	92	7936.00	6624.00	9.00	7927.00	320.00	192.00	32	0.0011	0.9597	0.8333	0.0656
FastMath	exp	110	22657	21121	0	22657.00	192	768	64	0.0000	0.9915	0.9167	0.1515
FastMath	sinQ	114	22369	21121	2	22367.00	128	178	64	0.0001	0.9943	0.6404	0.1522
NumberUtils	isNumber	116	900.00	0.00	1.00	899.00	640.00	640.00	1.00	0.0011	0.2889	0.9984	0.0000
FastMath	tanh	117	22658	21121	0	22658.00	128	128	64	0.0000	0.9944	0.5000	0.0857
FastMath	sinh	118	22978	21121	0	22978.00	128	192	64	0.0000	0.9944	0.6667	0.5455
FastMath	tanQ	133	23074	21121	2	23072.00	129	179	64	0.0001	0.9944	0.6425	0.0816
FastMath	expm1	139	23522	21121	0	23522.00	0	768	64	0.0000	1.0000	0.9167	0.1308
FastMath	atan	144	22819	21121	0	22819.00	34	223	64	0.0000	0.9985	0.7130	0.1045

5.4 Preliminary Models

In order to test our assumption that our redundancy metrics are statistically correlated with the REM of a program, we have conducted an empirical experiment, whereby we select a set of Java classes from the Apache Common Mathematics Library and run our Java compiler to compute the redundancy metrics of each method of each class. On the other hand, we apply a mutant generator to these classes using a uniform set of standard mutation operators, then we execute the base program and the mutants on benchmark test data sets and record how many mutants are killed by the test.

Simultaneously, we keep track of coverage metrics, and exclude from consideration any method whose line coverage is below 90%. By keeping in our sample only those Java classes for which line coverage is high (in fact the vast majority reach 100%-line coverage) we maximize the likelihood that mutants that are found to survive after undergoing the test are equivalent to the base program. Under this assumption, we use the ratio of surviving mutants of each method over the total number of mutants as the REM of the method. Our data sample includes about 234 methods.

We perform a statistical regression using the REM as the dependent variable and the intrinsic redundancy metrics (i.e., those metrics that pertain to the program, not the equivalence oracle) as the independent variables. We use a logistic model, i.e., a model such that REM is a linear combination of the independent variables. The metric that pertains to the equivalence oracle (ND) is not part of the regression analysis, but is integrated in the equation in such a way that if ND = 0 we obtain the regression formula involving the intrinsic metrics, and if ND = 1 (extreme case when the oracle tests trivially for true , i.e., all the mutants are found to be equivalent) we want the REM to be 1.

The resulting formula is:

$$REM = 0.1275+0.2442*SRI+0.0254*SRF-0.0314*FR. \quad (5.5)$$

With this equation in place, we can now have a tool that automatically computes the redundancy metrics, then derives the REM using this formula.

In the following Chapter 6, we refine the model based on different parameters such as lines of code, default size of array and string, test data size and mutation policy.

CHAPTER 6

REFINING THE MODELS

In this chapter, we want to study different possible settings to improve the multiple linear regression model accuracy. Through the forward selection, in each setting, we find the statistical correlation between redundancy (as quantified by our metrics), which are SRI, SRF, FR, NI, and the ratio of equivalent mutants REM. We also note that SRI and NI appear to be highly correlated. Inclusion of both variables in a model can result in unstable estimates. Therefore, we build the models only for the following variables [SRI, SRF, FR and REM]. We assess each model based on standard error, predication error or residual plot (residuals versus fitted values). We select the best model and we move to the next selection. In some cases, the regression model is not meaningful, and the standard errors of the models are very similar, so we can select any model of our choice.

6.1 Fine Tuning Component Size

In this Section, we want to study the impact of lines of code (LOC) on accuracy of the statistical model. We build three models for the following cases:

We build three models for the following cases:

- Case 1: model 1, is for multiple linear regression for all methods.
- Case 2: model 2, is for multiple linear regression for all methods that have LOC>=20.

- Case 3: model 3, is for multiple linear regression for all methods that have LOC>=40.

- Our selection for data size default is not important in this section. Therefore, we conduct empirical experiments on the data that is generated by the compiler for data default size 10. Table 6.1 shows the results of the experiments. We find the model 3 has smallest value of standard error. As a result, the model is selected, and we consider only methods that have LOC>=40 and we move to the next step.

Table 6.1 Standard Error and Model Formula of the REM for Each Model

Model	Standard Error	Model Formula of REM
Model1	0.323043251122893	REM=0.1275+0.2442*SRI+0.0254*SRF-0.0314*FR
Model2	0.50518184729197	REM=0.1192-0.8648*SRI+0.1390*SRF-0.0788*FR
Model3	0.138295857014054	REM=0.1149+0.0331*SRI+0.0841*SRF-0.0779*FR

6.2 Fine Tuning Default Parameters

In this section, our setting is data default size which includes array and string sizes. We want to find out if data default size can improve the accuracy of the regression model or not. We run our compiler on different data default sizes for 1, 2, 4, 6, 10. We fit the five regression models, evaluate them, and select the significant model. Table 6.2 shows the results of the experiments.

Table 6.2 Model Formula of the REM and Standard Error for Each Data Default Setting

Classification	Standard Error	Model Formula of REM
Data Default Size=1	0.13248953676045	REM =-0.0551-0.4179*SRI+0.2247*SRF+0.0435*FR
Data Default Size=2	0.133687606356218	REM=-0.0684-0.3156*SRI+0.2437*SRF+0.0229*FR
Data Default Size=4	0.136640601594657	REM=-0.0063-0.1913*SRI+0.1939*SRF-0.0295*FR
Data Default Size=6	0.138385346683011	REM=0.0611-0.092*SRI+0.1312*SRF-0.055*FR
Data Default Size=10	0.138295857014054	REM=0.1149+0.0331*SRI+0.0841*SRF-0.07793*FR

We find that data default size doesn't improve the performance of the regression model, since there is no difference among standard error among the models. Therefore, we can select any model, so we select the model of data default size =10.

6.3 Fine Tuning Test Size

So far, we have data that LOC>=40 and data default size =10. We study the impact of test data size. Our setting is test data size for each method, so we build the regression model for all methods, methods that have test size >=20, and methods that have test size >=40. Table 6.3 shows the REM formula for each model and standard error. We conclude that test data size doesn't significantly improve the accuracy of the model. There is not much difference in standard error. We can select any model. We select mode of test size >=40.

Table 6.3 Model Formula of the REM and Standard Error for Each Classification

Classification	Standard Error	Model Formula of REM
All Test Size	0.138295857014054	REM= 0.1149 +0.0331*SRI+0.0841 *SRF-0.0779 *FR
Test Size>=20	0.148382153401973	REM=0.0405-0.5413*SRI+0.1350*SRF+0.0448*FR
Test Size>=40	0.152588301735323	REM=-0.1481-0.5209 *SRI+0.3173*SRF+0.0667 *FR

6.4 Fine Tuning Mutation Policy

Now we have data that LOC>=40, data default size =10, and test size =40. In this Section, our setting is the impact of mutation operators of each class on the model. Each class has different mutation operators. The details of each class are mentioned in the previous section 5.2

Table 6.4 Model Formula of the REM and Standard Error for Each Class

Classification	Standard Error	Model Formula of REM
Class 1	0.156514737076486	REM=0.3437+0.3310*SRI-0.1548*SRF-0.0559*FR
Class 2	0.0994563743180568	REM=0.2583-0.7434*SRI-0.0719*SRF-0.0306*FR
Class 3	0.0806778025553742	REM=-0.0271-1.2669*SRI+0.3434*SRF+0.0833*FR
Class 4	0.152588301735323	REM=-0.1481-0.5209*SRI+0.3173*SRF+0.0667*FR

We can understand from the result that the impact of mutant operators improves the regression model. Based on standard Error and residuals plots (residuals versus fitted

values) in Figure 6.1, we conclude that the models of class 2 and class 3 are the most significant. All values in class 2 and class 3 are within the range [-0.1-0.1] except outliers. Also, we calculate the prediction error for each class, and we find that only class 2 and class 3 have prediction errors >=0.10.

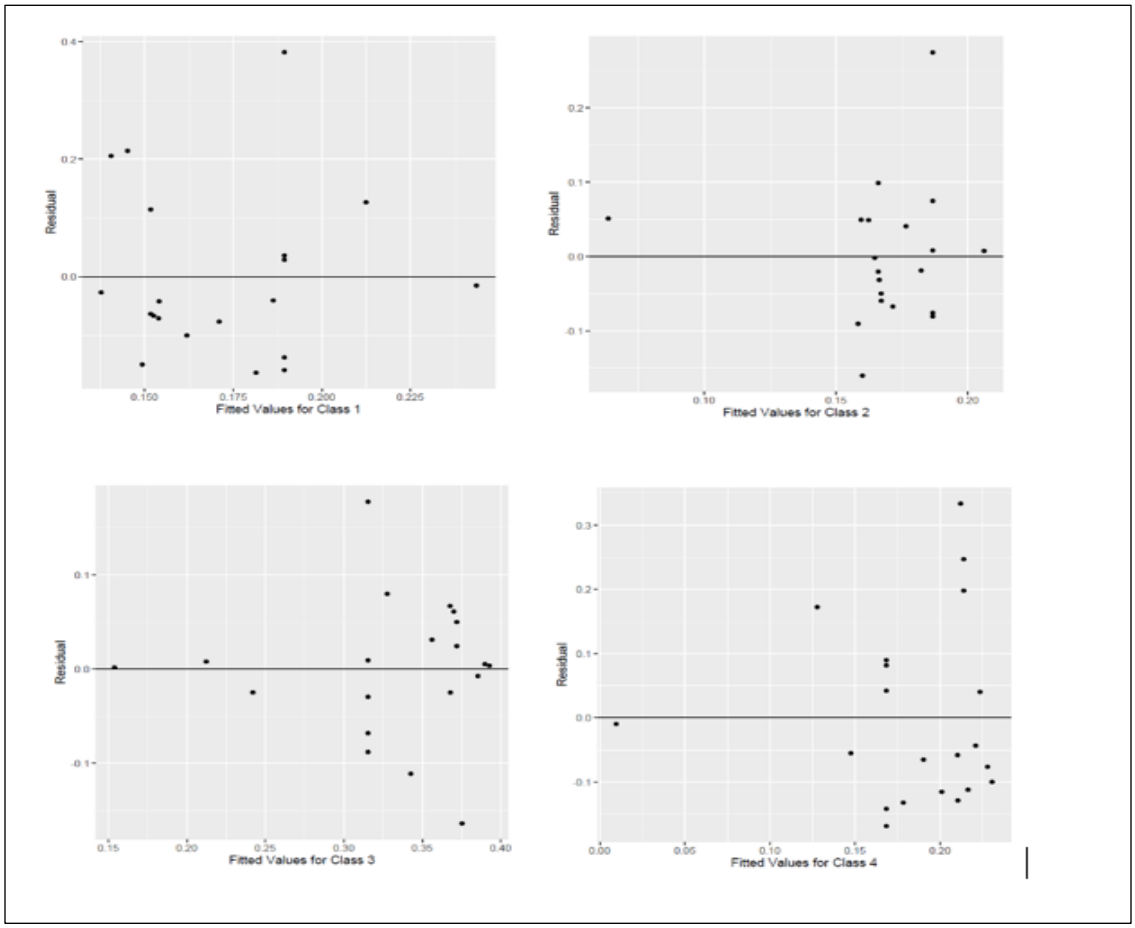

Figure 6.1 Residuals plot of each class.

CHAPTER 7

AUTOMATED ESTIMATION OF THE REM

We select class 2 and class 3 is the best meaningful models, and we show the correlation table, residuals, predication error, and residuals versus fitted values for each class. We plug in the values of SRI, SRF and FR variables into the REM formula. Then, we create the predicated REM column. We calculate predicated error which is equal to actual value of REM- predicated REM.

7.1 Class 2

Table 7.1 shows the correlation of each variable with the REM of class 2. The REM formula for class 2 is

$$REM= 0.2583-0.7434*SRI-0.0719*SRF-0.0306*FR \quad (7.1)$$

Table 7.1 Correlation Table for Class 2

	SRI	*SRF*	*FR*	*REM_CLASS2*
SRI	1			
SRF	-0.04377594	1		
FR	0.219976146	-0.333155345	1	
REM_CLASS2	-0.256383103	-0.067679541	-0.13508912	1

Standard Error is 0.0994563743180568. Table 7.2 shows the actual value of the REM, predicated REM, and predicated error. We find only two values that have predicated

error >=0.10. Figure 7.1 explains residuals plot versus fitted values. It's clear that all values are within [-0.1-0.1] except for two outliers.

Table 7.2 Residuals Table and Predicted Error for Class 2

REM_CLASS2	Predicated REM	Predicated Error
0.194968553	0.186609306	-0.008359247
0	0.15991875	0.15991875
0.217270195	0.176709012	-0.040561183
0.461139896	0.186608689	-0.274531207
0.145728643	0.1664	0.020671357
0.261538462	0.186605704	-0.074932757
0.110864745	0.186609611	0.075744866
0.106157113	0.186610224	0.080453112
0.163511188	0.182345946	0.018834758
0.107692308	0.167531476	0.059839169
0.21372549	0.206749764	-0.006975726
0.104347826	0.171806179	0.067458353
0.134993447	0.166800522	0.031807075
0.117414248	0.167463793	0.050049545
0.068027211	0.1589	0.090872789
0.163080408	0.165116995	0.002036588
0.211360634	0.162821682	-0.048538953
0.265135699	0.1664	-0.098735699
0.114772103	0.064242866	-0.050529237
0.208955224	0.160088253	-0.048866971

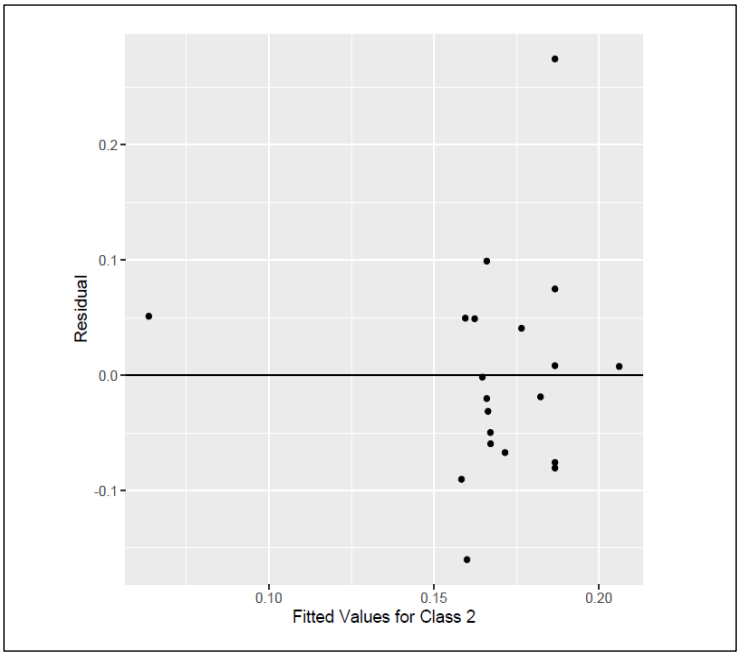

Figure 7.1 Residuals plot (residuals vs fitted values) of

7.2 Class 3

Table 7.3 shows the correlation of each variable with the REM of class 3. The REM formula for class 3 is

REM= -0.0271-1.2669*SRI+0.3434*SRF+0.0833*FR (7.2)

Table 7.3 Correlation Table for Class 3

	SRI	*SRF*	*FR*	*REM_CLASS3*
SRI	1			
SRF	-0.043534701	1		
FR	0.212056729	-0.325081884	1	
REM_CLASS3	-0.346097318	0.466440345	0.068956512	1

Standard Error is 0.0806778025553742. Table 7.4 shows the actual value of the REM, predicated REM, and predicated error. We find only three values that have predicated error >=0.10. Figure 7.2 represents residuals plot versus fitted values. All values are within [-0.1-0.1] except for three outliers.

Table 7.4 Residuals Table and Predicated Error for Class 3

REM_CLASS3	Predicated REM	Predicated Error
0.285714286	0.315300337	-0.029586051
0.217171717	0.24205	-0.024878283
0.231481481	0.342590802	-0.11110932
0.492890995	0.315303283	0.177587712
0.396296296	0.371833333	0.024462963
0.324590164	0.315317541	0.009272623
0.247191011	0.315298879	-0.068107868
0.227208976	0.315295952	-0.088086976
0.407407407	0.327556757	0.079850651
0.395061728	0.389748292	0.005313436
0.434607646	0.36757116	0.067036486
0.155555556	0.153866622	0.001688934
0.387211368	0.356010058	0.031201309
0.431034483	0.369920408	0.061114075
0.342913776	0.367787099	-0.024873323
0.396226415	0.392658333	0.003568082
0.211382114	0.375181612	-0.163799498
0.421768707	0.371833333	0.049935374
0.220198675	0.212198412	0.008000263
0.37791411	0.385293988	-0.007379878

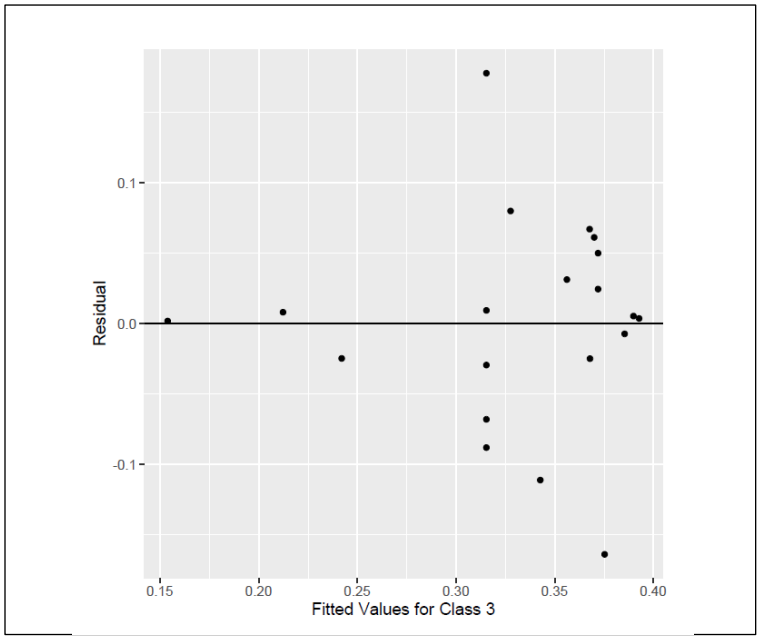

Figure 7.2 Residuals plot (residuals vs fitted values) of class 3.

CHAPTER 8

IMPLICATION

8.1 NEC: Number of Equivalence Classes

Mutant Equivalence

Given a set of M mutants of a base program P, and given a ratio of equivalent mutants REM, the number of equivalent mutants is estimated to be M ×REM. Hence, we cannot expect any test data set T to kill more than N = M×(1−REM) mutants (modulo the margin of error in the estimation of the REM).

Mutant Redundancy

In (Papadakis et al., 2019), Papadakis et al. raise the problem of mutant redundancy as the issue where many mutants may be equivalent among themselves, hence do not provide test coverage commensurate with their number. If we have sixty mutants divided into twelve classes where each class contains five equivalent mutants, then we have only twelve distinct mutants; and if some test data set T kills these sixty mutants, it should really get credit for twelve mutants (twelve casualties, so to speak), not sixty, since whenever it kills a mutant from one equivalence class, it automatically kills all the mutants of the same class. Of course, it is very difficult to determine, in a set of mutants, which mutants are equivalent, and which are not; but again, the REM enables us to draw some quantitative data about the level of redundancy in a pool of mutants.

The REM of the base program is computed using a regression formula whose independent variables are the redundancy metrics extracted from the source code of the program. Since the mutants are generated from the base program by means of elementary syntactic changes, it is reasonable to consider that the mutants have the same REM as the base program.

If we interpret the REM as the probability that any two mutants are semantically equivalent, then we can estimate the number of equivalence classes by answering the following question:

Given a set of size N, and given that any two elements of this set have a probability REM to be in the same equivalence class modulo some relation EQ, what is the expected number of equivalence classes of this set modulo EQ?

We denote this number by NEC (N, REM), and we write it as follows:

$$NEC(N, REM) = \sum_{k=1}^{N} K \times p(N, REM, K), \tag{8.1}$$

Where p (N, REM, K) is the probability that a set of N elements where each pair has probability REM to be equivalent has k equivalence classes. This probability satisfies the following inductive conditions.

Basis of Induction. We have two base conditions:

- One Equivalence Class.

$$p(N, REM, 1) = REM^{N-1} \tag{8.2}$$

This is the probability that all N elements are equivalent.

- As Many Equivalence Classes as Elements, or: All Equivalence Classes are Singletons.

$$p(N, REM, N) = (1 - REM)^{\frac{N \times (N-1)}{2}} \quad (8.3)$$

This is the probability that no two elements are equivalent: every two elements are not equivalent; there are N $\times$(N −1) pairs of distinct elements, but because equivalence is a symmetric relation, we divide this number by 2 (Mi ≠ Mj is the same event as Mj ≠ Mi).

Inductive Step

When we add one element to a set of N−1 elements, two possibilities may arise:

Either this adds 1 to the number of equivalence classes (if the new element is equivalent to no current element of the set); or it maintains the number of equivalence classes (if the new element is equivalent to one of the existing equivalence classes). Since these two events are disjoint, the probability of the disjunction is the sum of the probabilities of each event. Hence:

$$p(N, REM, K) = p(N-1, REM, K) \times (1 - (1 - REM)^K + p(N-1, REM, K - 1) \times (1 - REM)^{K-1} \quad (8.4)$$

The following recursive program, NEC, computes the number of equivalence classes of a set of size N whose elements have probability REM of being equivalent. Execution of this program with N = 65 and REM = 0.158 yields NEC(N,REM) = 14.64, i.e. , our 65 mutants represent only about 15 different mutants; the remaining 50 are redundant.

The following recursive program is used to find the number of equivalent classes given the REM and N, which is the number of mutants in any method.

```
#include <iostream>
#include <conio.h>
#include <map>
#include <vector>
#include "math.h"
using namespace std;
double p(int N, int k, double R);
std::map<vector<int>,double> resultsMap;
int main ()
{
  double R=0.0689; int N=116;
  double mean = 0.0;  double ps=0.0;
  for (int k=1; k<=N; k++)
    {double prob=p(N,k,R); ps = ps+prob;
     mean = mean + k*prob;
     double localVar=p(N,k,R);
     cout << k << " " << localVar << endl;}
    cout << "ps: " << ps << " mean: " << mean << endl;
     getch();
  }

 double p(int N, int k, double R)
  {
   vector<int> localVector;
   localVector.push_back(N);
   localVector.push_back(k);
   std::map<vector<int>,double>::iterator it=resultsMap.find(localVector);
   if(it!=resultsMap.end())
   {
      return it->second;
   };
   double result;
  if (k==1) {result=pow(R,N-1);}
   else
   if (N==k) {result=pow(1-R,(k*(k-1))/2);}
   else {result=p(N-1,k,R)*(1-pow(1-R,k))+p(N-1,k-1,R)*pow(1-R,k-1);}
   resultsMap.insert(pair<vector<int>,double>(localVector,result));
   //cout<<"Temporary result "<<N<<" "<<k<<" "<<R<<" - "<<result<<endl;
   return result;

   }
```

Verification of the number of equivalent classes

In this section, we want to verify the formula of finding the number of equivalent classes that is provided by the above program.

*Given a mutant **m** and test set **T**, what we refer to a vector is the array of all the outputs produced by m for all the elements of T, total number of mutants **N.** The algorithm finds the number of distinct mutants classes.*

We run our experiment on the Fibonacci class and we select the two following methods: *int_fib (int)* and void *power (int [][], int)*. We use *mujava* generation mutation policy to find the REM. For *int_fib (int), mujava* outputs 8 as surviving mutants and N, the total number of mutants, is equal to 32. Therefore, REM=8/32=0.25. NEC (N, REM) =NEC(32,0.25)= 8 distinct mutants classes.

For *void power (int [][], int) , mujava* outputs 0 as surviving mutants and N=7. Therefore, REM=0/7=0. NEC(N,REM)=NEC(7,0)= 7 distinct mutants classes. The question is, how good is our estimation? To answer the question, we do the following:

We write test class for each different mutant that runs 200 times. We store the mutants output into a text file. Then, we write mutant engine class that works out the comparison among all mutants' outputs and finds the distinct number of mutant classes. The mutant engine class finds 7 distinct mutants' classes for *int_fib (int)* and 4 for *void power (int [][], int)*. We plan to run more examples in future work.

The Validation of NEC

The proposed algorithm below shows the validation of NEC.

```
NEC(){
Given a set of M mutants of a base program P,
For each mutant m, run the mutant on nbtest test data.
Construct the output of m on testdata and save it to vector v.
vector=emptyvector; // vector of current mutant
for (t=1;t<=nbtest;t++){
        vector=vector+m(data(t)); //+ is append function, data is vector of input data
   if(vector not in vectorset){
     vector=vector Union {vector}; // add new output vector to set
     numDifferentClasses=numDifferentClasses Union {m};
     }// if
  }// for
   return numDifferentClasses;
}
```

8.2 Equivalence Based Mutation Score

The quantification of redundancy, discussed in the previous section, casts a shadow on the traditional way of measuring the mutation score of a test data set T: usually, if we execute a set of M mutants on some test data set T, and we find that X mutants have been killed (i.e., shown to be different from the base program P), we assign to T the mutation score X/M. This metrics ignores the possibility that several of M mutants may be equivalent, and several of the X killed mutants may be equivalent. We argue that this metric can be improved and made more meaningful, in three ways:

- Because of the possibility that mutants may be equivalent to the base program P, the baseline ought to be the number of non-equivalent mutants, i.e. N = (1−REM) ×M.

- Because of the possibility that those mutants that are not equivalent to P may be equivalent amongst themselves, we ought to focus not on the number of these mutants, but rather on the number of equivalence classes modulo semantic equivalence. This is defined in the previous section as NEC(N,REM).

- Because of the possibility that the X mutants killed by test data set T may be equivalent amongst themselves, we ought to give credit to T not for the cardinality of X, but rather for the number of equivalence classes that X may overlap. We refer to this number as COV(N,K,X), where K = NEC(N,REM) is the number of equivalence classes of the set of N mutants modulo equivalence.

  To compute COV(N,K,X), we designate by C1,C2, ...CK the K equivalence classes, we designate by fi, for ($1 \leq i \leq K$), the binary functions that take value 1 if and only if equivalence class Ci overlaps with (i.e., has a non-empty intersection with) set X, and value 0 otherwise. Then COV(N,K,X) = E($\sum_{i=1} Kf_i$). If we assume that all classes are the same size and that elements of X are uniformly distributed over the set of mutants, then this can be written as:

cov(N,K,X) = K × p(fi = 1) = K ×(1− p(fi = 0)), for an arbitrary i. For the first class to be considered, $p(f_i = 0) = \frac{K-1^X}{K}$, since each element of X has a probability $\frac{K-1}{K}$ of not being in class C1; for each subsequent element, the numerator and denominator each drops by 1. Hence, we have the following formula:

$$COV(N,K,X) = K\ x\ (1 - \frac{K-1^X}{K}\ x\ \prod_{i=0}^{X-1} N - \frac{i}{K-1}{N-i}), \qquad (8.5)$$

The following program computes this function, for N = 65, K = 15 and X = 50.

```
#include <iostream>
#include "math.h"
using namespace std;
double cov(int N, int K, int X);
int main ()
{
int N=65; int K=15; int X=50;
cout << "cov: " << cov(N,K,X) << endl;
}
double cov(int N, int K, int X)
{
float prod=1;
for (int i=0; i<K; i++)
{prod = prod *
(N-i/(float)(K-1))/(float)(N-i);}
return K*(1-prod*pow((K-1)/(float)K,X));
}
```

Execution of this program yields *COV*(65,15,50) = 12.55. We propose the following definition.

Definition1. *Given a base program P and M mutants of P, and given a test data set T that has killed X mutants, the* mutation score of *T is the ratio of equivalence classes covered by X over the total number of equivalence classes amongst the mutants that are not equivalent to P.*

We denote the mutation score by *EMS*(*M*, *X*).

The following proposition gives an explicit formula of the mutation score.

Proposition 1. *Given a program P and M mutants of P, and given a test data set T that has killed X mutants, the mutation score of T is given by the following formula:*

$$EMS(M, X) = \frac{COV(N, NEC(N, REM), X)}{NEC(N, REM)}, \tag{8.6}$$

where the REM is the ratio of equivalent mutants of P and N = M(1−REM) is the number of mutants that are not equivalent to P. In the example above, for N = 65, REM = 0.158, and X = 50 we find $EMS\,(77,50) = \frac{12.55}{15} = 0.84$,

8.3 MMS: Minimal Mutant Set

Now that we know how to estimate the redundancy of a set of mutants (by means of the NEC(N,REM)) function, we can derive a minimal set of mutants that is as good as the original set of mutants, but has no redundancy (i.e., all its elements are distinct). For example, imagine that we have 200 mutants and they are in 25 equivalent classes, how can we find 25 equivalent classes without having compare 200 mutants. The following program computes a minimal mutant set on the Fibonacci class of the method *int_fib (int)*. We have N, the number of total mutants, is equal to 32, and k, the number of different equivalent classes, is equal to 7. Figure 8.1 outputs the size of the minimal mutant set is equal to 18.

```
#include <iostream>
#include <map>
#include <vector>
#include "math.h"
using namespace std;

int main ()
{for (int k=1; k<=7; k++)
  {double bigoh=0;
  for (int i=1; i<=k; i++)
    {bigoh = bigoh + ((double)k/(double)i);
    }
  cout << "k= " << k << ". Big Oh()= " << bigoh << endl;}
}
```

Figure 8.1 Run of MMS.

Validation of MMS

To estimate how good the above estimation is, we propose the following algorithm.

*Given a mutant **m**, test set **T**, different equivalent classes **K.** The algorithm finds how many mutants we need to check before we get **K**.*

The proposed algorithm of validation of MMS can be found below. We use the same example as in the previous Section and the algorithm outputs 16 as the size of the minimal mutant set.

```
MMS()
{
vector=emptyset; // set of vectors obtained from distinct mutants
m=first(mutantset); // pick the first mutant in the set
while (card(vectorset)<NEC)  // while we have not found NEC distinct mutants
//card refers to cardinality
vector=emptyvector
for (t=1;t<=nbtest;t++){
vector=vector+m(data(t)); //+ is append function, data is vector of input data
   if(vector not in vectorset){
     vector=vector Union {vector}; // add new output vector to set
     minimalset=minimalset Union {m};
     }// if
     m=next(mutantset); // go to the next mutant
  }// while
  // now we have found NEC non-equivalent mutants; they constitute minimal set
   return minimalset;
}// end
```

CONCLUSION: SUMMARY AND PROSPECTS

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. Determination of mutant equivalence and mutant redundancy by inspection and analysis of individual mutants is very expensive and error-prone, at the same time that it is in fact unnecessary, for most purposes. As a substitute, we propose to analyze the amount of redundancy that a program has, in various forms, and we find that this enables us to extract a number of mutation-related metrics and attributes at negligible cost.

Specifically, we consider the following redundancy metrics: State Redundancy (*SRI* for the initial state, and *SRF* the final state of the program), Functional Redundancy (*FR*), Non-Injectivity of the program function (*NI*), and non-determinacy of the program specification (*ND*).

Central to this quantitative analysis is the concept of *ratio of equivalent mutants* (*REM*, for short), which measures the probability that any two mutants, or a mutant and the base program, are semantically equivalent. In this dissertation we proceed as follows:

- We highlight statistical relationships between the REM of a program and its redundancy metrics (SRI, SRF, FR, NI, ND) using experiments where the redundancy metrics are computed by hand.
- We develop a Java compiler that computes the redundancy metrics automatically, by analyzing the way execution of the program affects redundancy.

- We use the Java compiler to run a controlled experiment where the programs under consideration are of arbitrary size and complexity, and attempt to build four statistical models, which correspond to four different mutation policies.

All the steps executed so far are intended to estimate the REM of a program from a static analysis of its redundancy metrics. The next steps attempt to use the REM to support decision-making in mutation testing. These include:

- Estimating the number of equivalent mutants may multiplying the total number of mutants by (1-REM).

- Interpreting the REM as the probability that the original program and a mutant, or two distinct mutants, are equivalent, we estimate the number of equivalence classes in a set of mutants that are known to be distinct from the original; we call this function NEC(REM,N), where N is the number of mutants. This function reflects the amount of redundancy between the mutants; in other words, if N=100, and a test data T kills all of them, we want to distinguish between two situations: Did the test data set T kill 100 distinct mutants or 100 times the same mutant? NEC(REM,N) answers that question. For example, for the REM=0.15 and N=100, we find NEC=17.77; in other words, the 100 mutants we have killed amount to only 18 different mutants; the remaining 82 are redundant.

- Using the NEC() function, we turn our attention to the mutation score, and we argue that it requires a revision. Currently, when we have, say 100 mutants and a test data T kills 80 of them, we let the mutation score of T be 0.8, i.e. the ratio of killed mutants over the total number of mutants. We argue that the mutation score ought to count equivalence classes, not individual mutants, and we propose a new definition where the denominator is NEC(REM,N) and the numerator is the estimated number of equivalence classes that are covered by the set of killed mutants.

- The NEC function can also be used for another purpose: if (to cite the example above) 100 mutants are as good as 18, why are we using 100? Why can't we single out 18 distinct mutants and use only those? This is the well-known problem of minimal mutant set. Here again, knowledge of NEC via the REM helps a great deal. If we did not know how many distinct mutants to expect, we would have to compare each of the N mutants with the remaining (N-1) mutants, an O(N^2) operation. But if we know how many distinct mutants to expect, we can run an algorithm that finds distinct mutants until it reaches the count of NEC(); we have a program that estimates the number of iterations needed for this purpose. In the example above, with N=100 and NEC=17.77, we find that the expected number of mutants we need to consider is: 62.9.

B. Assessment

This work can be divided into two parts: the part that is geared towards estimating the REM, and the part that is geared towards using the REM to support decision-making. As far as estimating the REM, we make the following observations:

- We are able to show that the REM is statistically correlated to the redundancy metrics, using a sample of relatively small programs whose metrics are carefully computed by hand.

- When we use the compiler to tackle a sample of larger and more complex programs, we struggle to establish statistical relationships. Part of the difficulty is that the metrics are based on the assumption that we can readily compute the metrics of programs whose state entropy is easy to identify; most active benchmarks nowadays involve programs whose state space is ill-defined. It is not clear what variables are part of the state. This seems to have introduced biases into the metrics and precluded us from showing statistical relationships.

- We see two possible remedies to this situation, which can be used separately or jointly: one is to define broader metrics that take into account the case of programs whose entropy is not clearly identifiable; another is to consider a benchmark of programs where the entropy of the state space is more clearly defined. In the first case, the compiler's semantic rules have to be revised and adjusted.

C. Prospects

In the phase of estimating the REM, we envision to explore possible extensions to the Java compiler, as well as to apply the compiler to program samples that are better adapted to the proposed metrics.

In the phase of using the REM, we envision to validate our analytical results by means of empirical studies; we have started this process, as shown by the preliminary results presented in this thesis, but more remains to be done to conclude statistical significance.

APPENDIX A

THE PROGRAM NEC VALIDATES THE ALGORITHM OF COMPUTES NUMBER OF EQUIVALENT CLASSES (NEC).

```
import java.util.HashMap;
import AOIS_1.R1;
import AOIS_2.R2;
import AOIS_3.R3;
import AOIS_4.R4;
import AOIS_5.R5;
import AOIS_6.R6;
import AOIS_7.R7;
import AOIS_8.R8;
import AOIU_1.R9;
import AOIU_2.R10;
import AORB_1.R11;
import AORB_2.R12;
import AORB_3.R13;
import AORB_4.R14;
import CDL_2.R15;
import COI_1.R16;
import LOI_1.R17;
import LOI_2.R18;
import ODL_3.R19;
import ODL_4.R20;
import ROR_1.R21;
import ROR_2.R22;
import ROR_3.R23;
import ROR_4.R24;
import ROR_5.R25;
import ROR_6.R26;
import ROR_7.R27;
import SDL_1.R28;
import SDL_2.R29;
import SDL_3.R30;
import SDL_4.R31;
import VDL_2.R32;
import java.util.*;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.math.BigInteger;
```

```
import java.io.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class NEC {
public static void main(String[] args) throws IOException
 {
List<String> Mtemp  = new ArrayList();
NEC  test=new NEC();
Mtemp=test.FindEC();
int k=Mtemp.size();
System.out.println("the number of equivalence classes "+k);
}// main
public static List<String> mlist  = new ArrayList();

 //{
 // The engine of running the mutants
List<String> FindEC(){

// create objects of mutants // 88 mutants
R1  m1=new R1();
R2  m2=new R2();
R3  m3=new R3();
R4  m4=new R4();
R5  m5=new R5();
R6 m6=new R6();
R7 m7=new R7();
R8 m8=new R8();
R9 m9=new R9();
R10 m10=new R10();
R11 m11= new R11();
R12 m12 =new R12();
R13 m13 =new R13();
R14 m14 =new R14();
R15 m15=new R15();
R16  m16 =new R16();
R17 m17=new R17();
R18 m18 =new R18();
R19 m19=new R19();
R20 m20=new R20();
R21 m21=new R21();
R22 m22=new R22();
R23 m23=new R23();
```

```
R24 m24=new R24();
R25 m25=new R25();
R26 m26=new R26();
R27 m27=new R27();
R28 m28=new R28();
R29 m29=new R29();
R30 m30=new R30();
R31 m31=new R31();
R32 m32=new R32();
// call the mutants
m1.runfib();
m2.runfib();
m3.runfib();
m4.runfib();
m5.runfib();
m6.runfib();
m7.runfib();
m8.runfib();
m9.runfib();
m10.runfib();
m11.runfib();
m12.runfib();
m13.runfib();
m14 .runfib();
m15.runfib();
m16 .runfib();
m17.runfib();
m18.runfib();
m19.runfib();
m20.runfib();
m21.runfib();
m22.runfib();
m23.runfib();
m24.runfib();
m25.runfib();
m26.runfib();
m27.runfib();
m28.runfib();
m29.runfib();
m30.runfib();
m31.runfib();
m32.runfib();
List<String> Mutantslist  = new ArrayList();
```

```
String Filecomp="";
String Mname="AOIS_";
int m_num=1;
String MutantFile="m"+ Integer.toString(m_num);// source mutant
Filecomp=Mname+Integer.toString(1)+"//"+MutantFile+"."+"txt";

Mutantslist.add(Filecomp);
mlist.add(Filecomp);
Mutantslist=InitialStepComp(Filecomp,Mutantslist);
m_num++;
/// AOIS_1  8
MutantFile="m"+ Integer.toString(m_num);
for (int a=2;a<=8;a++){
Mname="AOIS_";
Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt"; // the path of mutant
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
m_num++;
MutantFile="m"+ Integer.toString(m_num);
}// for

//AOIU  1-2
for (int a=1;a<=2;a++){
Mname="AOIU_";
Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
m_num++;
MutantFile="m"+ Integer.toString(m_num);
}
//AORB_1 4
for (int a=1;a<=4;a++){
Mname="AORB_";
Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
m_num++;
MutantFile="m"+ Integer.toString(m_num);
}
 //CDL_2
Mname="CDL_";
Filecomp=Mname+Integer.toString(2)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
```

```
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
m_num++;
MutantFile="m"+ Integer.toString(m_num);

//COI 1
Mname="COI_";
Filecomp=Mname+Integer.toString(1)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);


m_num++;
MutantFile="m"+ Integer.toString(m_num);

//LOI 1  2
for (int a=1;a<=2;a++){
Mname="LOI_";
 Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);


m_num++;
MutantFile="m"+ Integer.toString(m_num);
}
// ODL   3  4
for (int a=3;a<=4;a++){
 Mname="ODL_";
 Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);


m_num++;
MutantFile="m"+ Integer.toString(m_num);
}
//ROR  1-7
for (int a=1;a<=7;a++){
Mname="ROR_";
Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
m_num++;
MutantFile="m"+ Integer.toString(m_num);
```

```
}
//SDL   1-4
for (int a=1;a<=4;a++){
Mname="SDL_";
Filecomp=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);

m_num++;
MutantFile="m"+ Integer.toString(m_num);
}
//VDL 2
Mname="VDL_";
Filecomp=Mname+Integer.toString(2)+"//"+MutantFile+"."+"txt";
mlist.add(Filecomp);
Mutantslist=FindEquiltyMutants(Filecomp,Mutantslist);
//System.out.println("m "+ m_num+"  "+Mutantslist.size());

m_num++;
MutantFile="m"+ Integer.toString(m_num);
return Mutantslist;
}

public static List<String> getmlist(){
return mlist;
}

public static List<String>   InitialStepComp(String mutantpath,List<String>  Mutantslist
){

int mindx=2;
String Filecomppath=" ";
/// AOIS_1-8
for (int a=2;a<=8;a++){
String Mname="AOIS_";
String MutantFile="m"+ Integer.toString(mindx);
Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}

//AOIU_1 2
```

```
for (int a=1;a<=2;a++){
String Mname="AOIU_";
String MutantFile="m"+ Integer.toString(mindx);
Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;

}/// for

//AORB_1-4

for (int a=1;a<=4;a++){
String Mname="AORB_";

String MutantFile="m"+ Integer.toString(mindx);
 Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}/// for
//////////////////////////////////////////
//CDL_2
 String Mname="CDL_";
String MutantFile="m"+ Integer.toString(mindx);
 Filecomppath=Mname+Integer.toString(2)+"//"+MutantFile+"."+"txt";

if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;

//COI 1
Mname="COI_";
MutantFile="m"+ Integer.toString(mindx);
Filecomppath=Mname+Integer.toString(1)+"//"+MutantFile+"."+"txt";

if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
//LOI  1  2
for (int a=1;a<=2;a++){
 Mname="LOI_";
 MutantFile="m"+ Integer.toString(mindx);
```

```
 Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";

if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}/// for

////////////////////////////////////
// ODL  3  4
for (int a=3;a<=4;a++){
 Mname="ODL_";
 MutantFile="m"+ Integer.toString(mindx);
Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";

if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}/// for
 ////////////////////////////////////
//ROR 1  7

for (int a=1;a<=7;a++){
 Mname="ROR_";
 MutantFile="m"+ Integer.toString(mindx);
 Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";

if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}/// for

//SDL   1 -4

for (int a=1;a<=4;a++){
 Mname="SDL_";
MutantFile="m"+ Integer.toString(mindx);
 Filecomppath=Mname+Integer.toString(a)+"//"+MutantFile+"."+"txt";
if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
}/// for

////////////
```

```
 //VDL 2
 Mname="VDL_";
MutantFile="m"+ Integer.toString(mindx);
 Filecomppath=Mname+Integer.toString(2)+"//"+MutantFile+"."+"txt";
if ( !Filecomppath.equals(mutantpath) && !findEquailty(Filecomppath,mutantpath) )
Mutantslist.add(Filecomppath);
mindx++;
return Mutantslist;
}
//////////////////////////////////////////
public static List<String>   FindEquiltyMutants(String mutantpath,List<String>
Mutantslist )
{
int s=Mutantslist.size();
List<String> Mutantslisttemp  = new ArrayList();
for (int i=0;i<s;i++)
{
if ( !findEquailty(mutantpath,Mutantslist.get(i)) )
 if(!Mutantslisttemp.contains(Mutantslist.get(i)))
  Mutantslisttemp.add(Mutantslist.get(i));
  }
   if(!Mutantslisttemp.contains(mutantpath))
    Mutantslisttemp.add(mutantpath);
return Mutantslisttemp;
}

 public static  boolean   findEquailty(String f1, String  f2)
          {
     boolean areEqual = true;
     int lineNum = 1;
     String line1="";
     String line2 ="";

         /// function to compare files
      try {

      BufferedReader reader1 = new BufferedReader(new FileReader(f1));
      BufferedReader reader2 = new BufferedReader(new FileReader(f2));
      line1 = reader1.readLine();
      line2 = reader2.readLine();
        while (line1 != null || line2 != null)
      {
         if(line1 == null || line2 == null)
```

```
            {
               areEqual = false;
               break;
            }
            else if(! line1.equalsIgnoreCase(line2))
            {
               areEqual = false;

               break;
            }

            line1 = reader1.readLine();
             line2 = reader2.readLine();
             lineNum++;
         }
          reader1.close();

         reader2.close();
      } catch ( IOException e ) {
            e.printStackTrace();
         }

         if(areEqual)
         {

            return true;
         }
         else
         {

            return false;
         }// else

           }// end of the funciton

}
```

APPENDIX B

THE PROGRAM VALIDATES THE ALGOTIRHM OF CALCULATING MINIMAL MUTANT SET(MMS)

```
import java.util.HashMap;
import AOIS_1.R1;
import AOIS_2.R2;
import AOIS_3.R3;
import AOIS_4.R4;
import AOIS_5.R5;
import AOIS_6.R6;
import AOIS_7.R7;
import AOIS_8.R8;
import AOIU_1.R9;
import AOIU_2.R10;
import AORB_1.R11;
import AORB_2.R12;
import AORB_3.R13;
import AORB_4.R14;
import CDL_2.R15;
import COI_1.R16;
import LOI_1.R17;
import LOI_2.R18;
import ODL_3.R19;
import ODL_4.R20;
import ROR_1.R21;
import ROR_2.R22;
import ROR_3.R23;
import ROR_4.R24;
import ROR_5.R25;
```

```
import ROR_6.R26;
import ROR_7.R27;
import SDL_1.R28;
import SDL_2.R29;
import SDL_3.R30;
import SDL_4.R31;
import VDL_2.R32;
import java.util.*;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.math.BigInteger;
import java.io.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class MMS {
public static void main(String[] args) throws IOException
 {
List<String> Mtemp  = new ArrayList();
NEC  test=new NEC();
Mtemp=test.FindEC();
int k=Mtemp.size();// number of equivalent classes extracted from previous program
List<String> Mutantslist=test.getmlist();// store all mutants
int minsetNumber=minset(k,Mutantslist);
System.out.println("minsetNumber="+minsetNumber);
}// main
public static int minset(int k,List<String>Mutantslist){
// k= 7. Big Oh()= 18.15
TestFib  test=new TestFib();
```

```
List<String> signatureset  = new ArrayList();
int nbit=0;
boolean check=false;
int s=Mutantslist.size();
for (int i=0;i<s;i++){
// m=nextMutants();
String m=Mutantslist.get(i);
signatureset.add(m);
if(!signatureset.contains(m)) signatureset.add(m);
for (int j=0;j<signatureset.size();j++){
if (!m.equals(signatureset.get(j))&&
//comparing  outputs of m on T;
test.findEquailty(m,signatureset.get(j)) )
check=true; // there is mutant is equal to it
}// for j
if( check==true){
 //(signature in signatureset)
signatureset.remove(m);
check=false;
}// if
nbit++;
if (signatureset.size()==k)    break;
}// for i// loop for all mutants
return nbit;
 }// end of the method
}// class
package AOIS_1;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.io.*;
```

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.BigInteger;

public class R1
{
  public R1(){}
  public void runfib(){
  BufferedWriter output = null;
   try {

         File file = new File("AOIS_1//m1.txt");
         output = new BufferedWriter(new FileWriter(file));
         for (int i=1;i<=200;i++){
         int result=Fibonacci.fib(i);
         output.write(Integer.toString(result));
         }// for

        output.close();

   } catch ( IOException e ) {
         e.printStackTrace();
      }


  }
```

APPENDIX C

FOR EACH MUTANT, THE TEST CLASS HAS TO BE ADDED TO RUN MUTANT OUTPUT

```
package AOIS_1;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.io.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigDecimal;
import java.math.BigInteger;

public class R1
{
  public R1(){}
  public void runfib(){
  BufferedWriter output = null;
   try {

          File file = new File("AOIS_1//m1.txt");
         output = new BufferedWriter(new FileWriter(file));
         for (int i=1;i<=200;i++){
         int result=Fibonacci.fib(i);
         output.write(Integer.toString(result));
         }// for

        output.close();

   } catch ( IOException e ) {
         e.printStackTrace();
      }


  }
```

BIBLIOGRAPHY

[1] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L.Traona, and M. Harmanb, "Mutation Testing Advances: An Analysis and Survey," In advances in computers, 1st ed.,vol.112, A.M.Memon, Ed. San Diego, CA: Elsevier Science Publishing Co Inc, 2018, ch.6, pp. 378. doi:10.1016/bs.adcom.2018.03.015

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," IEEE Transactions on Software Engineering, vol.3, no.4, pp.279-290, July.1977.doi: 10.1109/TSE.1977.231145

[3] R. A. DeMillo and R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, vol.11, no.4, pp.34-41, Apr.1978. doi:10.1109/C-M.1978.218136

[4] R. A. DeMillo, "Test Adequacy and program mutation," *in Proceedings of the 11th International Conference on Software Engineering*, pp.335-356, May.1989. doi:10.1145/74587.74634

[5] 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, June.2010. doi:10.1109/TSE.2010.62

[6] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol.40, no.1, pp. 23-42, Jan.2014. doi:10.1109/TSE.2013.44

[7] F. C. Souza, M. Papadakis, V. H. S. Durelli, and M. E. Delamaro, "Test data generation techniques for mutation testing: a systematic mapping," *Workshop on Experimental Software Engineering*, vol. XI, pp. 1– 14, Apr.2014. doi:10.13140/RG.2.1.3699.9209

[8] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing-approach and case studies," *Science of Computer Programming*, vol.120, pp. 25–48, May.2016. doi:10.1016/j.scico.2016.01.003

[9] R. A. Silva, S. do Rocio Senger de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information & software technology.*, vol.81, pp.19–35, Jan.2017. doi:10.1016/j.infsof.2016.01.017

[10] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to UML model-based conformance test generation," *1st International Conference on Software Testing, Verification, and Validation*, pp. 82-91, Apr. 2008. doi:10.1109/ICST.2008.48

[11] K. El-Fakih, A. Kolomeez, S. Prokopenko, and N. Yevtushenko, "Extended finite state machine based test derivation driven by user defined faults," 1st *International Conference on Software Testing, Verification, and Validation*, Apr.2008. doi: 10.1109/ICST.2008.16

[12] M. B. Trakhtenbrot, "Implementation-oriented mutation testing of statechart models*," 3rd International Conference on Software Testing, Verification and Validation*, pp.120-125, Apr.2010. doi:10.1109/ICSTW.2010.55

[13] S. F. Adra, and P. McMinn, "Mutation operators for agent-based models," *3rd International Conference on Software Testing, Verification and Validation*, pp.151-156, Apr.2010. doi:10.1109/ICSTW.2010.9

[14] F. Belli, M. Beyazit, T. Takagi, and Z. Furukawa, "Mutation testing of "go-back" functions based on pushdown automata," *4th IEEE International Conference on Software Testing, Verification and Validation,* pp.249-258, Mar.2011. doi: 10.1109/ICST.2011.30

[15] B. K. Aichernig, H. Brandl, E. J¨obstl, and W. Krenn, "Efficient mutation killers in action," *4th IEEE International Conference on Software Testing, Verification and Validation*, pp.120-129, Mar.2011. doi:10.1109/ICST.2011.57

[16] B. K. Aichernig, H. Brandl, E. J¨obstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Software Testing, Verification & Reliability*, vol.25, no.8, pp.716-748, Feb.2014.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Towards automated testing and fixing of re-engineered feature models," *35th International Conference on Software Engineering*, pp.1245-1248, May.2013. doi:10.1109/ICSE

[18] P. Arcaini, A. Gargantini, and E. Riccobene, "Using mutation to assess fault detection capability of model review," *Software Testing, Verification & Reliability*, vol.*25*, no.5-7, pp.629-652, Nov.2015. doi:10.1002/stvr.1530

[19] P. Arcaini, A. Gargantini, and P. Vavassori, "Generating tests for detecting faults in

feature models," *8th IEEE International Conference on Software Testing, Verification and Validation,* pp.1-10, April.2015. doi:10.1109/ICST.2015.7102591

[20] R. A. M. Filho, and S. R. Vergilio, "A multi-objective test data generation approach for mutation testing of feature models," *Journal of Software Engineering Research and Development*, vol. 4, Dec.2016. doi:10.1186/s40411-016-0030-9

[21] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, "Featured model-based mutation analysis," *in Proceedings of the 38th International Conference on Software Engineering*, pp.655-666, May.2016. doi:10.1145/2884781.2884821

[22] T. Su et al., "Guided, stochastic model-based GUI testing of android apps," *in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 245-256, Sep.2017. doi:10.1145/3106237.3106298

[23] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology,* vol.5, no.2, pp.99-118, Apr. 1996. doi:10.1145/227607.227610

[24] 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, Oct.2011. doi: 10.1109/TSE.2010.62

[25] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes and G. Fraser, "Are mutants a valid substitute for real faults in software testing," *in Proceedings of the 22nd ACM SIGSOFT International Symposiumon Foundations of Software Engineering*, China, Nov. 2014, pp.654-665. doi: 10.1145/2635868.2635929

[26] J. H. Andrews, L. C. Briand, and I. Labiche, "Is mutation an appropriate tool for testing experiments," *Proceedings. 27th International Conference on Software. Engineering*, May.2005. doi: 10.1109/ICSE.2005.1553583

[27] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," *in Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp.342-352, July.2011.doi: 10.1145/2001420.2001461

[28] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," *in Proceedings of the 36th International Conference on Software Engineering*, pp.919-930, June.2014. doi: 10.1145/2568225.2568265

[29] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353-374, Apr.2012.

[30] B. J. Gruen, D. Schuler, and A. Zeller, "The impact of equivalent mutants," *Software Testing, Verification and Validation Workshops*, May.2009. doi:10.1109/ICSTW.2009.37

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

[32] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," *International Symposium on Software Testing and Analysis*, July.2014. doi:10.1145/2610384.2610388

[33] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," *in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp.295-306, July.2017. doi:10.1145/3092703.3092714

[34] 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, Dec. 2014. doi: 10.1016/j.scico.2014.05.012

[35] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31-45, Mar.1982.

[36] J. A. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 164-192, 1997. Dec.1998.

[37] J. Voas and G. McGraw, "Software fault injection: inoculating programs against errors," *Journal of Software: Testing, Verification and Reliability*, vol.9, no.1, pp.75-76, Mar.1999.

[38] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Mutation Testing for the New Century*, vol.24, W.E.Wong, Eds. Springer, 2001, ch.10, pp.5-13. doi: https://doi.org/10.1007/978-1-4757-5939-6_4

[39] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233-262, Dec.1999.

[40] M. Ellims, D. C. Ince, and M. Petre, "The csaw C mutation tool: initial results," *Testing: Academic and Industrial Conference Practice and Research Techniques -MUTATION*, Oct.2007. doi: 10.1109/TAIC.PART.2007.28

[41] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371-379, July.1982. doi: 10.1109/TSE.1982.235571

[42] D. Schuler, V. Dallmaier and A. Zeller, "Efficient mutation testing by checking invariant violations," *in Proceedings of the 18th International Symposium on Software Testing and Analysis*, pp.69-80, July.2009. doi:10.1145/1572272.1572282

[43] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *in Proceedings of the 21st international conference on Software engineering*, vol. 27, no. 2, pp. 213-224, May.1999. doi:10.1145/302405.302467

[44] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," *in Proceedings WS-FMDS* , pp.1-8, July.2012.

[45] L. Carvalho et al., "Equivalent mutants in configurable systems: An empirical study," *in Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, pp.11-18, Feb.2018. doi:10.1145/3168365.3168379

[46] 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, Apr.2018. doi:10.1109/TSE.2017.2684805

[47] I. Csaszar, J. Koerner, *Information theory: coding theorems for discrete Memoryless Systems*, 2nd ed. Cambridge, UK: Cambridge University Press, 2011.

[48] R. M. Gray, *Entropy and information theory*, 1st ed., Springer, 2011.

[49] J.C. Laprie, "Dependability-its attributes, impairments and means," *in Predictably Dependable Computing Systems*, B. Randell, J.Laprie, H. Kopetz, and B.Littlewood, Ed. Springer,1995, pp 1–19. doi: https://doi.org/10.1007/978-3-642-79789-7_1

[50] A.Mili , S. Aharon, and C. Nadkarni, "Mathematics for reasoning about loop functions," *Science Computer Program*, vol.74, no.11-12, pp.989–1020, Nov.2009.

[51] I. Marsit, M. Nazih Omri, J. Loh, and A. Mili, “Impact of mutation operators on the ratio of equivalent mutants*,” International Conference on Intelligent Methodolgies, Tools and Technologies*, vol.17, Sep.2018.

[52] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379-423, July.1948.