5-31-2024

Empirical exploration of software testing

Samia Alblwi
New Jersey Institute of Technology, samia6mb@gmail.com

ABSTRACT

EMPIRICAL EXPLORATION OF SOFTWARE TESTING

by
Samia Alblwi

Despite several advances in software engineering research and development, the quality of software products remains a considerable challenge. For all its theoretical limitations, software testing remains the main method used in practice to control, enhance, and certify software quality. This doctoral work comprises several empirical studies aimed at analyzing and assessing common software testing approaches, methods, and assumptions. In particular, the concept of mutant subsumption is generalized by taking into account the possibility for a base program and its mutants to diverge for some inputs, demonstrating the impact of this generalization on how subsumption is defined. The problem of mutant set minimization is revisited and recast as an optimization problem by specifying under what condition the objective function is optimized. Empirical evidence shows that the mutation coverage of a test suite depends broadly on the mutant generation operators used with the same tool and varies even more broadly across tools. The effectiveness of a test suite is defined by its ability to reveal program failures, and the extent to which traditional syntactic coverage metrics correlate with this measure of effectiveness is considered.

EMPIRICAL EXPLORATION OF SOFTWARE TESTING

by
Samia Alblwi

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Information Systems

Department of Informatics

May 2024

APPROVAL PAGE

EMPIRICAL EXPLORATION OF SOFTWARE TESTING

Samia Alblwi

Ali Mili, PhD, Dissertation Co-advisor Professor of Computer Science, NJIT	Date
Vincent Oria, PhD, Dissertation Co-advisor Professor of Computer Science, NJIT	Date
Ji Meng Loh, PhD, Committee Member Professor of Mathematical Sciences, NJIT	Date
Michael Lee, PhD, Committee Member Associate Professor of Informatics, NJIT	Date
Hai Phan, PhD, Committee Member Associate Professor of Data Science, NJIT	Date
Ali Parsai, PhD, Committee Member Software Testing Researcher, IT Project Manager at ECO3, Mortsel, Belgium	Date

BIOGRAPHICAL SKETCH

Author: Samia Alblwi

Degree: Doctor of Philosophy

Date: May 2024

Undergraduate and Graduate Education:

- Doctor of Philosophy in Informatics,
  New Jersey Institute of Technology, Newark, NJ, 2024
- Master of Science in Computer Science,
  Clark Atlanta University, Atlanta, GA, 2017
- Bachelor of Science in Information System,
  Taibah University, Medina, Saudi Arabia, 2011

Major: Information Systems

Presentations and Publications:

S. Alblwi, A. Ayad, and A. Mili, "Mutation Coverage is Not Strongly Correlated with Mutation Coverage" *The 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024), 2024.*

S. Alblwi, and A. Ayad, "Minimizing Mutant Sets by Equivalence and Subsumption" *World Academy of Science, Engineering, and Technology, Open Science Index 205, International Journal of Computer and Systems Engineering, 18(1), Pages 21-27, 2024.*

S. Alblwi, A. Ayad, B. Khaireddine, I. Marsit, and A. Mili, "Semantic Coverage: Measuring Test Suite Effectiveness"*In Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023), pages 287-294, 2023 .*

S. Alblwi, A. Ayad, B. Khaireddine, I. Marsit, and A. Mili, "Quantifying the Effectiveness of Mutant Sets" *In 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), pp. 288-297. IEEE, 2022.*

S. Alblwi, I. Marsit, B. Khaireddine, A. Ayad, J. Loh, and A. Mili, "Three Forms of Mutant Subsumption: Basic, Strict and Broad"*International Conference on Software Technologies, pages 122-144, 2022.*

S. Alblwi, I. Marsit, B. Khaireddine, A. Ayad, J. Loh, and A. Mili, “Generalized Mutant Subsumption“ *In Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022), pages 46-56.*

S. Alblwi, I. Marsit, B. Khaireddine, A. Ayad, J. Loh, and A. Mili, “Subsumption, Correctness, and Relative Correctness“ *Submitted to journal SSRN 4598614.*

S. Alblwi, and K. Shujaee, “A survey on wireless security protocol WPA2“ *Proceedings of the International Conference on Security and Management (SAM 2017), pages 12-17, 2017.*

This dissertation is dedicated to my beloved mother, Nura Abdullah. Whose strength, wisdom, and unconditional support shaped the person I am today. Your memory will forever be cherished, and your legacy will live on through the knowledge and achievements I have attained. I know she is seeing me from heaven. For you alone, I think and plan.

ACKNOWLEDGMENTS

هَذَا مِنْ فَضْلِ رَبِّيْ

The success and accomplishments of my Ph.D. journey are a tremendous grace from God.

Prof.Ali Mili is my godfather, and this journey would have been impossible without him. I started to realize day by day that, as an advisee, what I really value in my Ph.D. journey is the advisor who enriches my academic experience with behavior, thinking, and kindness. I enormously appreciated his patience with me. After Allah, all this great credit to Prof.Ali Mili. My journey had many challenges, and I couldn't cope alone, but Prof. Ali's heroism and kindness made the journey possible. Also, Prof.Vincent Oria accepted me from day one and trusted me to work under his name while working with Prof.Ali.

I want to thank the rest of my dissertation committee, Drs. Ji Meng Loh, Micheal Lee, Hai Phan, and Ali Parsai. I am truly honored to have them as my committee members. Each member has provided extensive personal and professional guidance and taught me much about scientific research and life.

I want to extend my sincere thanks to Taibah University and the Higher Education Ministry of Saudi Arabia. Without their generous support, I would not have completed my Ph.D. degree.

From the bottom of my heart, I would like to thank all the friends and peers I have met at NJIT, especially my Friend Dr. Firas Gerges, who works at Princeton University, for his help and always being by my side whenever I needed it.

I wrote my heart out to reach my sister Asmaa, whom I love deeply and the only one in my family who supported me to see me succeed. My second family here in the USA, Prof.Ali Mili's family, his wife Dr.Amel, and his daughter with their hospitality. Also, I want to thank my friend Eng. Mohamed Ismael and his family for being my brother and safe side in the USA.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Software Testing

Software testing is the art of executing a program on some selected Test data will be used to check whether the program behaves as expected. There are four types of software testing, depending on the goal of the testing activity:

- *Unit Testing*, which is the activity where a programmer tests a small unit (method, routine, class, etc) against the specification of the unit to ensure that the unit performs the function it is intended to perform. Unit testing is usually carried out during the programming phase of a software lifecycle.
- *Integration Testing*, which is the activity where a software engineer tests a software system to check whether the individual units of the system work properly together, assuming that each unit works correctly by unit testing. Integration testing is usually carried out during the integration phase of a software project.
- *Acceptance Testing*, which is the activity where a team of representatives from the software developer organization and representatives from the software user organization test a completed software product to check that it meets the contractual obligations between the two parties. Acceptance testing is usually carried out during the phase of deployment of the product.
- *Regression Testing*, which is the activity where a software maintenance engineer tests a software product at the end of a maintenance update to check that the new version of the product did not regress with respect to the original version. Software maintenance can be applied to correct a fault (corrective maintenance) or to enhance performance (perfective maintenance) or to adapt to new requirements (adaptive maintenance). The purpose of regression testing is to ensure that the objective of the maintenance operation has been achieved without loss of functionality.

Software testing usually involves four phases [1]:

- *Generating a Test Environment.* Since many activities of testing take place in a development environment, rather than the environment where the software product is actually used, it is important to simulate the operating environment in which the software being tested is expected to run.

- *Generating Test Data.* This is the most important phase of the testing lifecycle and the phase that determines the success of failure of the test. This phase consists of deciding what input data to test the program on; the selection of test data depends on the goal of the test (finding faults, checking correctness, checking non-regression, estimating reliability) and on the scale (budget) of the test. This step can be carried out automatically or by detailed analysis.

- *Generating an Oracle.* The oracle is the agent that determines, for each execution, whether the outcome of the execution is correct. To generate the oracle for a test, we must have a specification of the requirements of the product; the most common form of an oracle is a Boolean function that checks whether a pair of (input-output) meets the requirements for correctness.

- *Generate a Test Driver.* A test can be carried out manually by executing the program on selected inputs and observing the corresponding outputs, or it can be carried out automatically by means of a test driver. The test driver is a program that runs the software product under test on the selected test data tests each execution with respect to the oracle, and reports on the results of each test.

- *Analyzing the Outcome of the Test.* The outcome of a test may be captured by a report that details what happened for each test data: which executions were found to satisfy the test oracle, which was not. The interpretation of this report depends on the type of test (unit testing, integration testing, acceptance testing, and regression testing).

1.2 Mutation Testing

Because test data selection is the most critical phase of the test lifecycle, it has received the most attention in software testing research. Mutation testing was introduced by DeMillo et al. [2] as a way to check whether a test suite is adequate, i.e., whether it tests the software product sufficiently thoroughly: it consists of generating several mutants of the program under test, then checking for each mutant, whether the test suite can detect the mutations, by exposing the difference between the base program and the mutant for at least one element of the test suite. The idea is that, to the extent that mutations are faithful representations of faults, a test suite that detects mutations can also detect faults [3–9].

This idea has given rise to a great deal of research on mutation testing, including the development of several tools to generate mutants [10–13]. The main obstacle to the widespread use of mutation testing in practice is that it is very costly because of the number of mutants of a program increases rapidly with the size of the program. Several attempts have been made to control the size of mutant sets: In [14] Marsit et al. attempt to estimate the number of equivalent mutants in a set by assessing the amount of redundancy in the base program in [15, 16] Kurtz et al. define the relation of *subsumption* between mutants, and argue that in a set of mutants, we can remove all the mutants that are subsumed by others without reducing the effectiveness of a mutant set.

Program mutations are also used in program repair [17]: To repair an incorrect program, we generate mutants of it and then test them against a test suite that represents the desired correct behavior; this research has led to the development of several prototypes that repair programs by a process called *generate-and-validate*.

1.3 Research Plan

In this doctoral research, our team carries out empirical studies about software testing, mutation testing, and test suite effectiveness.

In Chapter 2, I consider the original definition of subsumption [15, 16] and resolve to generalize it by considering the possibility that the base program or its mutants may fail to converge for some inputs. By considering different interpretations of what is the outcome of an execution when two outcomes are comparable, and under what condition are two comparable outcomes identical, we derive three possible definitions of subsumption. I have developed a Python script that can analyze the output of a base program and a mutant for some test suites, and derive the differentiator sets of the mutant with respect to the three interpretations of subsumption (delta0, delta1, delta2). I have also developed a Python script, which, given the differentiator, sets of a set of mutants with respect to a base program, derives the three corresponding subsumption graphs, by highlighting the inclusion relationships between the differentiator sets. This work is published in [18].

In Chapter 3, I analyze two approaches to the minimization of mutant sets:

- Either by partitioning the set of mutants into equivalence classes modulo semantic equivalence, then selecting one element per equivalence class; this is the approach proposed by [14].
- Or by ordering mutants by means of the subsumption relation and selecting the maximal mutants in the subsumption graph; this is the approach proposed by [15,16].

Two questions arise with respect to these approaches: First, subsumption should not be defined between individual mutants but rather between equivalence classes of mutants. Second, it is important to consider the question of how much we save with subsumption once we have partitioned the mutants into equivalence classes. If we start with 100 mutants and find that they are partitioned into 10 equivalence classes, how much more reduction do we achieve with subsumption? This work is published in [19].

Test suite effectiveness is usually measured by coverage metrics, such as statement coverage, branch coverage, line coverage, path coverage, etc. Chapter 4 focuses on defining a measure of effectiveness that is based on a test suite's ability to reveal program failures; this measure is called *semantic coverage*, and is published in [20].

When we try to minimize a set of mutants, be it by equivalence [14] or by subsumption [15, 16], we assume implicitly that we are reducing the number of mutants, but we are not reducing the effectiveness of the set of mutants. The problem of reducing the size of a set of mutants is an optimization problem where the objective function is the cardinality of the set and the constraint under which the objective function is minimized is that the effectiveness of the original mutant set is the same as the effectiveness of the reduced set. This raises the question: how do we define the effectiveness of a set of mutants. Chapter 5 proposes two definitions of mutant set effectiveness (assured effectiveness, potential effectiveness), and proves that the removal of subsumed mutants preserve potential effectiveness. This work is published in [21].

In Chapter 6, titled *Mutant Coverage is not Strongly Correlated with Mutation Coverage*, we run an empirical experiment where we consider a sample benchmark program, and its associated benchmark test suite, then we generate twenty subsets of this test suite of varying sizes (between 0.4 and 0.6 of the size of the original test suite). Then, we compute the mutation coverage of these twenty test suites under different mutant generation policies, and we find that the same test suite can have widely varying levels of mutation coverage depending on the mutant generation policy. Specifically, we find correlations near 0.5 between mutation coverage values obtained by different mutation operators of the same tool, and correlations near zero between mutation coverage values obtained by different tools. This work is published in [22].

In Chapter 7, we run an experiment in which we compare the values of traditional coverage metrics (statement coverage, branch coverage, condition coverage, line coverage, mutation coverage) with respect to semantic coverage, which reflects a test suite's ability to reveal the failures of an incorrect program. We find that most coverage metrics have a very low correlation with semantic coverage, and that mutation coverage has a higher correlation with semantic coverage than all the syntactic coverage metrics. This work is currently under review.

1.4 Mathematical Background

Because we use sets to represent program spaces, we represent sets by C-like variable declarations. If we declare a set S by the variable declarations:

```
xType x; yType y;
```

then S is the cartesian product of the sets of values that the types `xType` and `yType` take; elements of S are denoted by lower case s, and are referred to as *states*. Given an element s of S, we may refer to the x-component (resp. y-component) of s as $x(s)$ (resp. $y(s)$). But we may, for the sake of convenience, refer to the x component of states s, s', s'' (e.g.,) simply as x, x', x''.

A relation on set S is a subset of the cartesian product $S \times S$; special relations on set S include the *universal relation* $L = S \times S$, the *identity relation* $I = \{(s, s')|s' = s\}$ and the *empty relation* $\phi = \{\}$. Operations on relations include the set theoretic operations of union ($\cup$), intersection ($\cap$), difference($\setminus$) and complement ($\overline{R} = L \setminus R$). They also include the *product* of two relations, denoted by $R \circ R'$ (or RR', for short) and defined by

$$R \circ R' = \{(s, s')|\exists s\text{"} : (s, s\text{"}) \in R \wedge (s\text{"}, s') \in R'\}.$$

The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$. The *domain* of relation R is denoted by $dom(R)$ and defined by $dom(R) = \{s|\exists s' : (s, s') \in R\}$. The *pre-restriction* of relation R to set T is the relation denoted by ${}_{T\setminus}R = \{(s, s')|s \in T \wedge (s, s') \in R\}$.

A relation R is said to be *reflexive* if and only if $I \subseteq R$; relation R is said to be *symmetric* if and only if $R = \widehat{R}$; relation R is said to be *transitive* if and only if $RR \subseteq R$; relation R is said to be *asymmetric* if and only if $R \cap \widehat{R} = \phi$; relation R is said to be antisymmetric if and only if $R \cap \widehat{R} \subseteq I$. A relation R is said to be an *equivalence relation* if and only if it is reflexive, symmetric, and transitive. A relation R is said to be a *partial ordering* if and only if it is reflexive, transitive and antisymmetric. A relation R is said to be a *strict partial ordering* if and only if it is transitive and asymmetric.

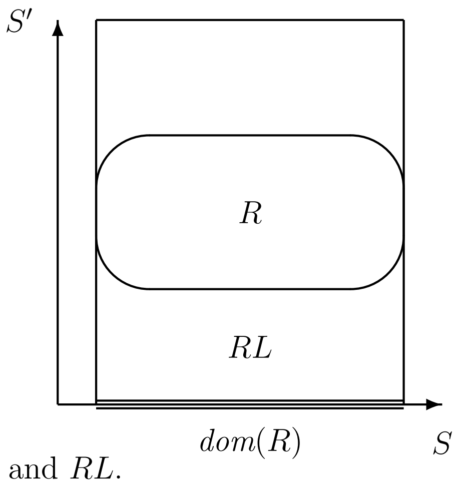


Figure 1.1 Relations R and RL.

A relation R is said to be *deterministic* (or: to be a *function*) if and only if $\widehat{R}R \subseteq I$. A relation R is said to be *total* if and only if $RL = L$. A relation R is said to be a *vector* if and only if $RL = R$; a vector V on set S is a relation of the form $V = A \times S$ for some non-empty subset A of S. We may use vectors to define pre-restrictions and post-restrictions: Given a relation R and a vector $V = A \times S$, the pre-restriction of R to A can be written as $V \cap R$ and the post-restriction of R to A can be written $R \cap \widehat{V}$. Note that given a relation R, the product of R by the universal relation L yields the rectangular relation $RL = dom(R) \times S$. See Figure 1.1; we use RL as a representation of the domain of R in relational form.

1.4.1 Program semantics

Definition 1. *Given two relations R and R' on space S, we say that R'* refines *R (abbreviation: $R' \sqsupseteq R$, or $R \sqsubseteq R'$) if and only if: $RL \subseteq R'L$ and $RL \cap R' \subseteq R$.*

Intuitive interpretation: this definition means that R' has a larger domain than R, and that R' assigns fewer images than R to the elements of the domain of R. This definition of refinement is the relational equivalent to the usual formula of refinement, which provides for weaker precondition ($RL \subseteq R'L$) and stronger postcondition ($RL \cap R' \subseteq R$) [23–26]. See Figure 1.2.

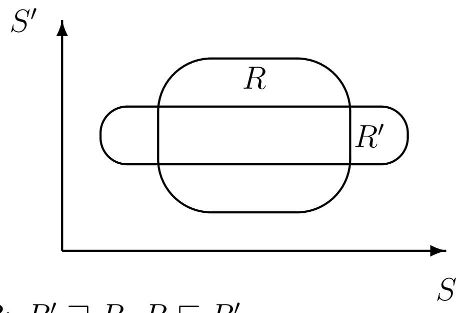


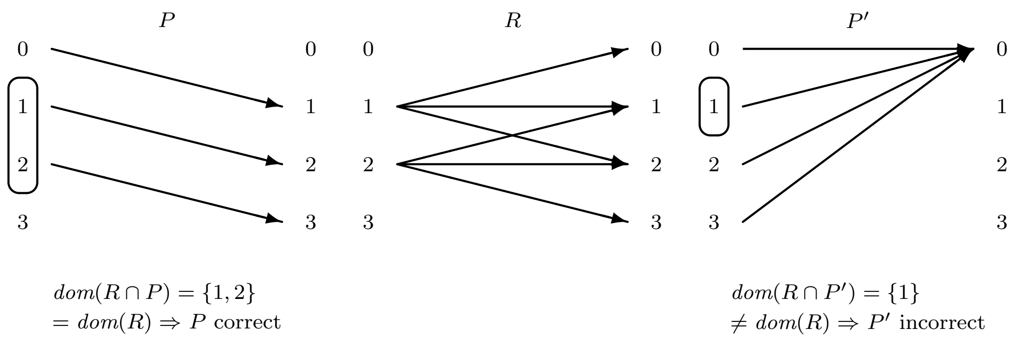
Figure 1.2 R' refines R: $R' \sqsupseteq R$, $R \sqsubseteq R'$.

Given a program P on space S, the function of program P (which, by abuse of notation, we also denote by P) is the set of pairs of states (s, s') such that if execution of program P starts in state s, it terminates normally in state s'; by *terminates normally* we mean that the execution terminates after a finite number of steps, without attempting any illegal operation such as a division by zero, an array reference out of bounds, a reference to a null pointer, a square root of a negative number, etc... As a consequence of this definition, the domain of P is the set of states (elements of S) such that execution of P on s terminates normally.

A specification on space S is a binary relation on S; it contains all the pairs of states (s, s') that the specifier considers correct. The correctness of a program P on space, S can be determined with respect to a specification R on S according to the following definition.

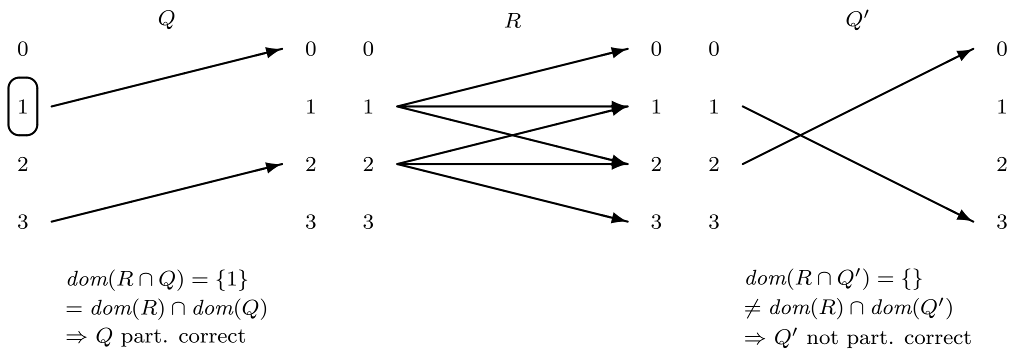
Definition 2. *Given a program P on state S and a specification R on S, we say that P is (totally) correct with respect to R if and only if P refines R. We say that P is* partially correct *with respect to R if and only if P refines $R \cap PL$.*

Except for the fact that they are formulated in relational terms, these definitions are equivalent to traditional definitions of total and partial correctness [25–28]. Figures 1.3 and 1.4 illustrate the properties of total and partial correctness; to be totally correct with respect to specification R, a program must obey the specification for all elements of $dom(R)$, whereas a partially correct program must obey the



P
R
P'
$dom(R \cap P) = \{1, 2\}$
$= dom(R) \Rightarrow P$ correct
$dom(R \cap P') = \{1\}$
$\neq dom(R) \Rightarrow P'$ incorrect

Figure 1.3 Total correctness.



Q
R
Q'
$dom(R \cap Q) = \{1\}$
$= dom(R) \cap dom(Q)$
$\Rightarrow Q$ part. correct
$dom(R \cap Q') = \{\}$
$\neq dom(R) \cap dom(Q')$
$\Rightarrow Q'$ not part. correct

Figure 1.4 Partial correctness.

specification only where it terminates. Figures 1.4 and 1.3 illustrate the priorities of Partial correctness and Total correctness concerning the specification.

CHAPTER 2

BASIC, STRICT AND BROAD SUBSUMPTION

2.1 Three Forms of Subsumption

In Kurtz et al. [15, 16]. introduce the concept of mutant subsumption, whereby a mutant M of program P is said to *subsume* a mutant M' of P if and only if M produces a different outcome from P for at least one input, and any input for which M produces a different outcome from P causes M' to produce a different outcome from P as well. This concept has given rise to much research aiming to minimize mutant sets using the criterion of subsumption [29–33]: if a mutant set μ includes M and M' such that M subsumes M', we can remove M' from μ without affecting the effectiveness of μ, provided we do keep M.

While it is defined in terms of execution outcomes, the concept of subsumption remains vague as to the exact definition of an execution outcome; also, while the definition is based on whether two executions are identical or distinct, and it remains to determine when two outcomes are comparable, and when two comparable outcomes can be considered correct. We discuss these questions below:

- *What is the Outcome of a Terminating Program?*. We argue that even when two programs terminate normally, it is not always clear whether they have the same outcome: Consider the following two programs whose purpose is to swap two integer variables x and y:
  ```
  P1:  {int x, y, z; z=x; x=y; y=z;}
  P2:  {int x, y, z; z=y; y=x; x=z;}
  ```
  Whether we consider these two programs to have the same outcome or two different outcomes depending on what we consider to be the outcome of each program: If we consider the final value of x and y to be the outcome of these programs, then they do have the same outcome if we consider the final state of these programs to be the outcome of their executions, then they have different outcomes, as the final value of z is not the same for P1 and P2.

- *Is Non-Termination an Outcome or the Absence of Outcome?* When we execute a program on some input (or starting at some initial state), then the program may terminate after a finite number of steps in some final state; we then say that the execution *converges*. But the execution may also lead to an exception, such as entering an infinite loop; attempting a division by zero; attempting to reference a nil pointer; attempting to access an array outside of its bounds; applying the log function to a negative number, etc. We then say that the execution *diverges*; the question that this situation raises whether we consider divergence to be an execution outcome, or do we consider that when a program diverges, it has no outcome (in which case we cannot compare its outcome to that of another program)?

The discussion of divergence may sound like a mundane academic question, but it is, in fact, very relevant to mutation: indeed, several mutation operators are prone to cause mutants to diverge, even when the base program converges. For example,

– If we consider the following loop that indexes an array `A[0..N-1]` using some index variable `i` in the base program P
```
P: while (i<N) {a[i]=0; ...  i=i+1;}
```
and a mutation operator changes condition `(i<N)` onto `(i<=N)`, then the mutant will diverge due to an array reference out of bounds.

– If the base program P has a variable x of type integer and a variable y of type float, and includes the following guarded assignment
```
P: if ((x!=0) && (x!=1)) {y=1.0/(x*(x-1));...  }
```
and a mutation operator changes the conjunction into a disjunction, then the resulting mutant will diverge for $x = 0$ and $x = 1$, due to a division by zero.

– If the base program has an integer variable x and includes a loop of the form
```
P: while (x>0) {x=x-2; ..;}
```
and a mutation operator changes the condition `(x>0)` into `(x!=0)` then the resulting mutant will diverge whenever the initial value of x is odd due to an infinite loop.

Not only do we need to make provisions for cases where mutants diverge, we must also consider the possibility that the base program itself may diverge for some inputs: indeed, test data is not determined by the domain of the program, but rather by the domain of the specification that the program is supposed to satisfy. For all these reasons, it is important to (re)define subsumption in a way that makes provisions for cases where the base program and/ or its mutants fail to converge.

- *Comparing Execution Outcomes.* Now that we recognize that not all executions converge, and not all converging executions have well-defined outcomes, we must decide on two questions:

  – When are two outcomes comparable?

  – When are two comparable outcomes identical?

  In this chapter, we will present three distinct definitions of mutant subsumption, which depends on how we answer these two questions.

Interestingly, we find that once we admit the possibility that the base program and its mutants may diverge, but there is no difference between *true subsumption* and *dynamic subsumption*, as defined originally: dynamic subsumption with respect to program P for some test suite T is the same as true subsumption with respect to program P', the pre-restriction of P to T. Also, we argue that there is no difference between *true subsumption* and *static subsumption*, since *static subsumption* refers to a static method to establish true subsumption, rather than to a different property between the base program and its mutants.

Hence, rather than the distinction between *true subsumption*, *static subsumption* and *dynamic subsumption*, we present an orthogonal classification: *basic subsumption*, *strict subsumption*, and *broad subsumption*; this classification is based on what one considers to be execution outcomes, what one considers to be comparable outcomes, and what one considers to be distinct execution outcomes. We also show, in passing, that the property of mutant subsumption is very similar to the property of *relative correctness*, which orders candidate programs for correctness with respect to a specification, and was introduced to define program faults [34]. Given the ongoing discussions about the relationships between faults and mutations [6, 7, 35, 36], it is hardly surprising that similar concepts are introduced to model mutations [15, 16] and faults [34]; but it is noteworthy nevertheless.

In Section 2.2, we introduce three definitions of differentiator sets, where a differentiator set between two programs is the set of inputs for which the programs return different outcomes; there are three different forms of differentiator sets, depending on how we interpret program outcomes and how we compare program outcomes. In Section 2.3, we use the three forms of differentiator sets to introduce three different forms of mutant subsumption. In Subsection 2.4.2, we use the definitions presented in Section 2.3, to derive a statistical model that predicts the shape of subsumption graphs; specifically, we estimate the likelihood of subsumption between any two mutants, the estimated number of subsumption relations between any two mutants (i.e., the number of arcs in a subsumption graph), and the number of maximal nodes in a subsumption graph. In Section 2.5, we consider a benchmark program, generate its mutants, weed out its equivalent mutants, then compute the three differentiator sets of each mutant with the base program; using these differentiator sets, we derive the subsumption relations between the mutants and draw the three subsumption graphs between the mutants; the fact that these subsumption graphs are different from each other proves that the three forms of subsumption are indeed meaningful. In Section 2.6, we summarize our findings, critique them, and then discuss venues for further research.

2.2 Three Forms of Differentiator Sets

2.2.1 Divergence and differentiator sets

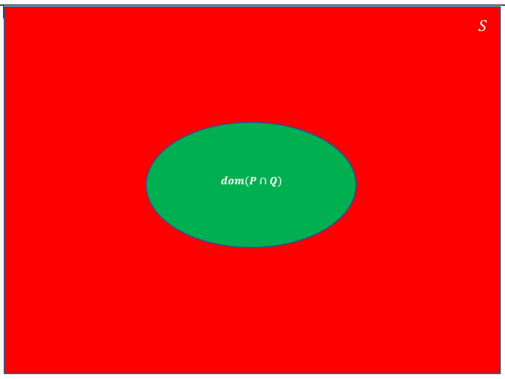
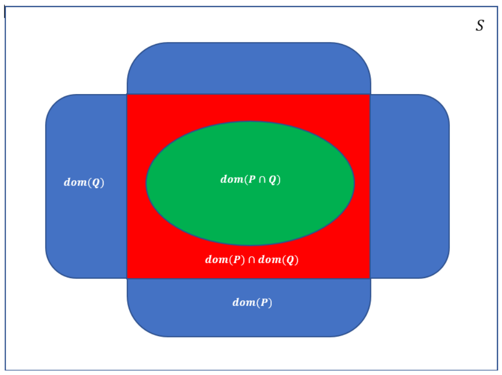
In Section 2.1, we argued that while the definitions of mutant subsumption refer to program outcomes and the condition under which two program outcomes are identical, they are not perfectly clear about what constitutes the outcome of a program, when two program outcomes are comparable, and if they are, when can we consider them to be identical? In this section, we address this ambiguity by introducing several definitions of *differentiator sets*, which reflect different interpretations of the questions above.

Given two programs, say P and Q, the *differentiator set* of P and Q is the set of initial states for which execution of P and Q yield different outcomes. For the purposes of this chapter, we adopt the three definitions of differentiator sets proposed by Mili in [37]:

- *Basic Differentiator Set.* The basic differentiator set of two programs P and Q on space S is defined only if P and Q converge for all s in S; it is the set of states s such that $P(s) \neq Q(s)$. This set is denoted by $\delta_B(P,Q)$ and defined by:
$$\delta_B(P,Q) = \overline{dom(P \cap Q)}.$$
- *Strict Differentiator Set.* The strict differentiator set of two programs P and Q on space S is defined regardless of whether P and Q converge for all initial states. It includes all the states for which executions of P and Q both converge and yield distinct outcomes. This set is denoted by $\delta_S(P,Q)$ and defined by:
$$\delta_S(P,Q) = dom(P) \cap dom(Q) \cap \overline{dom(P \cap Q)}.$$
- *Loose (or Broad) Differentiator Set.* The loose (broad) differentiator set of two programs P and Q on space S is defined regardless of whether P and Q converge for all initial states. It includes all the states for which executions of P and Q both converge and yield distinct outcomes, as well as the states for which only one of the programs converges and the other diverges. This set is denoted by $\delta_L(P,Q)$ and defined by:
$$\delta_L(P,Q) = (dom(P) \cup dom(Q)) \cap \overline{dom(P \cap Q)}.$$

Figure 2.1 illustrates the three definitions of differentiator sets (represented in red in each case). To gain an intuitive understanding of these definitions, it suffices to note the following details:

- The domain of program P ($dom(P)$) is the set of initial states on which execution of P converges (i.e. terminates normally after a finite number of steps without raising any exception or attempting any illegal operation). We assume that when a program enters an infinite loop, it gets timed out by the run-time environment so that non-termination is an observable outcome.
- The domain of $(P \cap Q)$ is the set of inputs for which programs P and Q converge and return the same outcome.
- The complement of the domain of $(P \cap Q)$ is the set of inputs for which program P and Q converge and return distinct outcomes. In other words,
$$\overline{dom(P \cap Q)} = \{s : s \in dom(P) \wedge s \in dom(Q) \wedge P(s) \neq Q(s)\}.$$

	
$\delta_B(P,Q)$	$\delta_S(P,Q)$

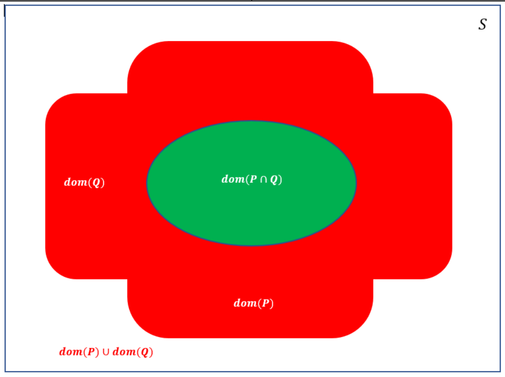

$\delta_L(P,Q)$

Figure 2.1 Due to Samia et al.,(2022): Three definitions of differentiator sets.

For an illustration of differentiator sets under the strict and broad interpretation, we consider the following programs P and Q on space S defined by an integer variable s.

```
P:  {if (s<0) {while (s!=0) {s=s-1;}}
     else {s=pow(s,4)+35*s*s+24;}}
Q:  {if (s>5) {while (s!=5) {s=s+1;}}
     else {s=10*pow(s,3)+50*s;}}
```

Note that P fails to converge for all s less than zero (since it enters an infinite loop) and Q fails to converge for all s greater than 5 (for the same reason). The functions of these programs are:

$P = \{(s, s')|s \geq 0 \wedge s' = s^4 + 35s^2 + 24\}$.

$Q = \{(s, s')|s \leq 5 \wedge s' = 10s^3 + 50s\}$.

From these definitions, we compute the following parameters:

$dom(P) = \{s|s \geq 0\}$.

$dom(Q) = \{s|s \leq 5\}$.

$P \cap Q = \{(s, s')|0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s \wedge s' = 10s^3 + 50s\}$.

$dom(P \cap Q) = \{s|0 \leq s \leq 5 \wedge s^4 + 35s^2 + 24 = 10s^3 + 50s\}$.

By solving the equation $(s^4 + 35s^2 + 24 = 10s^3 + 50s)$, we can simplify the formula of $dom(P \cap Q)$ as:

$dom(P \cap Q) = \{s|1 \leq s \leq 4\}$.

Whence we find the following results for the strict differentiator set and the broad differentiator set of programs P and Q:

$\delta_1(P, Q) = \{0, 5\}$.

$\delta_2(P, Q) = \{s|s \leq 0 \vee s \geq 5\}$.

Interpretation:

- *Strict Differentiator Set.* The set of initial states that expose the difference between P and Q is $\{0, 5\}$ because the interval $[0..5]$ includes all the initial states where both P and Q are defined $(dom(P) \cap dom(Q))$, and programs P and Q return the same results for initial states in the interval $[1..4]$ $(dom(P \cap Q))$.

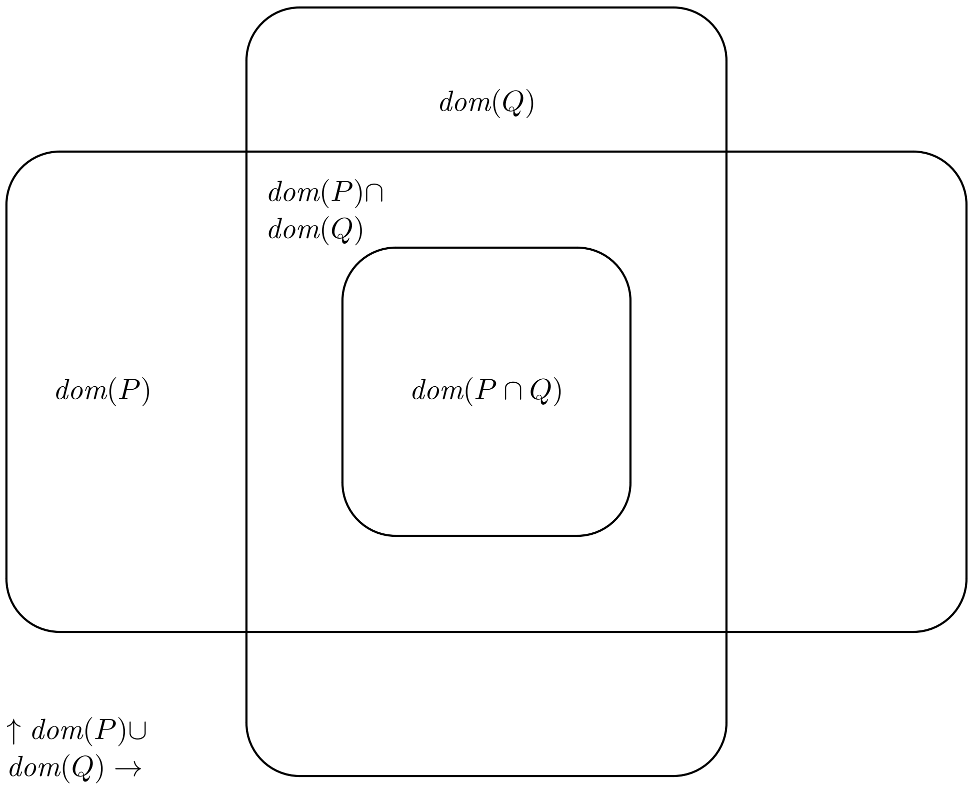


Figure 2.2 Test data to expose behavior difference.

- *Broad Differentiator Set.* Any initial state outside the interval [1..4] exposes the difference between P and Q, either because they are both defined but give different results (if the initial state is 0 or 5) or because one of them terminates normally while the other diverges (for s greater than 5, P terminates normally but Q does not; for s negative, Q terminates normally but P does not).

2.3 Three Forms of Subsumption

2.3.1 Definitions and properties

In Kurtz et al. [15, 16] , define the concept of *true subsumption* as follows:

Definition 3. *Given a program P on S and two mutants M and M', we say that M* subsumes *M' with respect to P if and only if:*

P1 There exists an initial state s for which P and M produce different outcomes.

P2 For all s in S such that P and M produce different outcomes, so do P and M'.

Since this definition makes no mention of P, M or M' failing to converge, we assume that P, M and M' are considered to converge for all initial states. The following Proposition formulates subsumption by means of the basic differentiator sets.

Proposition 1. *Given a program P on space S and two mutants M and M' of P, M subsumes M' if and only if:*

$$\emptyset \subset \delta_B(P, M) \subseteq \delta_B(P, M').$$

The proof of this proposition is due to [38]. Proposition 1 provides an alternative formula to define mutant subsumption in the case where we assume that all programs and mutants terminate for all initial states. This proposition is formulated in terms of the basic differentiator sets, which are defined when the program and its mutants are assumed to converge for all initial states, but in Section 2.2, we have introduced two more definitions of differentiator sets, which do not assume universal convergence of programs and mutants, and take a liberal interpretation of program outcomes and when to consider outcomes as identical or distinct. The following definitions generalize the concept of subsumption to the case when programs and their mutants do not necessarily converge for all initial states.

Definition 4. Strict Subsumption. *Given a program P on space S and two mutants M and M' of P, we say that M* strictly subsumes *M' if and only if:*

$$\emptyset \subset \delta_S(P, M) \subseteq \delta_S(P, M').$$

Definition 5. Loose Subsumption. *Given a program P on space S and two mutants M and M' of P, we say that M* loosely subsumes *M' if and only if:*

$$\emptyset \subset \delta_L(P, M) \subseteq \delta_L(P, M').$$

In Section 4.4, we see that the distinction between the basic definition of subsumption (Definition 3, [15, 16]), strict subsumption, and loose subsumption is not a mere academic exercise. These definitions yield vastly different subsumption graphs.

2.3.2 Subsumption and dynamic subsumption

The following definition is due to Kurtz et al. [15, 16].

Definition 6. *Given a program P, a test suite T and two mutants of P, say M and M', we say that M* dynamically subsumes *M' with respect to P for test suite T if and only if:*

D1 There exists a test $t \in T$ such that P and M compute different outcomes on t.

D2 For every possible test $t \in T$, if M computes a different outcome from P, then so does M'.

From Definitions 3 and 6, it is easy to see that true subsumption is a special case of dynamic subsumption, namely the case when T is the set of all possible tests. In the following proposition, we show that dynamic subsumption is also a special case of true subsumption.

Proposition 2. *Given a program P, a test suite T and two mutants of P, say M and M', M dynamically subsumes M' with respect to P for test suite T if and only if M subsumes M' (in the sense of true subsumption) with respect to $P' =_{T\backslash} P$, where $_{T\backslash}P$ is the pre-restriction of P to T.*

The proof of this proposition is due to [38]. The program that computes function ${}_{T\backslash}P$ can be written as:

```
p':  {if (s in T) {p;} else {abort();}}
```

Dynamic subsumption of M over M' for base program P with respect to test suite T is the same as true subsumption of M over M' for base program ${}_{T\backslash}P$.

2.4 Statistical Modeling

2.4.1 Probability of subsumption

In [17], Gazzola et al. present a comprehensive survey of program repair; this survey highlights the predominance of search space size as the most critical concern in program repair. At its core, program repair is the act of making a program more-correct than it is [39]; when the program has only one fault (which is what many program repair experiments assume), then making the program absolutely correct is indistinguishable from making it relatively correct (i.e. more correct than it is). Most program repair methods rely on common mutation operators to generate repair candidates, essentially the same kind of mutation operators that are used in mutation experiments; on the other hand, according to [38], the search for candidate repairs is based on the same criterion (relative correctness) as the determination of subsumption relations. This raises the following question:

- *If mutants are generated by the same operators, and pairs of mutants are compared using the same criterion (relative correctness $\Leftrightarrow$ subsumption), why is it so difficult to find program repairs (i.e., to reveal relative correctness relationships) yet so easy to reveal subsumption relations, as most subsumption graphs published in the literature are very dense (in terms of number of arcs over number of nodes)? Is this perhaps the result of loss of recall in program repair, or loss of precision in mutant subsumption?*

We are not going to answer this question in this chapter (as that requires an empirical study well beyond the scope of this chapter, but we will discuss the mathematics that enable us to analyze this matter.

Relative correctness is determined by checking an inclusion relationship between competence domains, and subsumption is determined by checking an inclusion relationship between differentiator sets. Hence, both criteria can be modeled statistically by considering the following question: If we choose K **non-empty** subsets of a set of size T, what is the probability that any two subsets be in an inclusion relationship? Once we estimate this probability, we can answer two related questions:

- What is the expected number of inclusion relationships between these K subsets? This would be the expected number of arcs in a subsumption graph of K nodes.
- What is the expected number of maximal subsets among the K subsets? This would be the size of the minimal set of mutants, as determined by subsumption.

These two questions are addressed in the next two subsections. It is important to note that by modeling subsumption and relative correctness with set inclusion, we are assuming that the competence domains of two mutants with respect to a specification are statistically independent. Our statistical analysis is sound only to the extent that this assumption is valid.

2.4.2 Graph density

By abuse of notation, we use the same symbol to denote a set and its cardinality. Given a set T and K non-empty subsets thereof, we ponder the question: what is the probability that any two subsets among K are in an inclusion relationship? The Following proposition is due to [38].

Proposition 3. *Under the assumption of statistical independence of differentiator set the expected number of arcs in a subsumption graph of K nodes can be approximated* *by:*

$$(\frac{3}{4})^T \times K(K-1).$$

In practice, even with moderate values of T, this expected number is very small.

2.4.3 Number of maximal nodes

Using the probability estimate $p = \frac{3}{4}^T$, we can estimate the probability that any subset of T is maximal: A given subset is maximal if and only if all $(K-1)$ other subsets are not supersets thereof; hence,

$$prob(maximality) = (1 - (\frac{3}{4})^T)^{K-1}.$$

Whence we derive the expected number of maximal mutants in a subsumption (/ relative correctness) graph that stems from K mutants and a test suite of size T:

$$K \times (1 - (\frac{3}{4})^T)^{K-1}.$$

2.5 Illustration: Three Subsumption Graphs

We consider the Java benchmark program of *jTerminal* [1], an open-source software product routinely used in mutation testing experiments [30]. We apply the mutant generation tool *LittleDarwin* in conjunction with a test generation and deployment class that includes 35 test cases [30]; we augmented the benchmark test suite with two additional tests, intended specifically to *trip* the base program *jTerminal*, by causing it to diverge. We let T designate the augmented test suite codified in this test class; all our analysis of mutant equivalence, mutant redundancy, mutant

[1]available online at `http://www.grahamedgecombe.com /projects /jterminal`

survival, etc is based on the outcomes of programs and mutants on this test suite (and carefully selected subsets thereof). Execution of LittleDarwin on jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants; for the sake of documentation, we list them below:

```
m1, m2, m7, m8, m9, m10, m11, m12, m13,
m14, m15, m16, m17, m18, m19, m21, m22,
m23, m24, m25, m26, m27, m28, m44, m45,
m46, m48, m49, m50, m51, m52, m53, m54,
m55, m56, m57, m58, m59, m60, m61, m62,
m63, m83, m88, m89, m90, m92, m93.
```

The remaining 46 mutants are semantically equivalent to the pre-restriction of jTerminal to T. The first order of business is to partition these 48 mutants into equivalence classes modulo semantic equivalence; we find that these 48 mutants are partitioned into 31 equivalence classes, and we select a member from each class; we let μ be the set of selected mutants: $\mu =$

```
m1, m2, m7, m11, m13, m15, m19, m21, m22,
m23, m24, m25, m27, m28, m44, m45, m46, m48,
m49, m50, m51, m52, m53, m55, m56, m57, m60,
m63, m92, m93.
```

We resolve to draw the subsumption graphs of these mutants according to the three definitions:

- Basic/ True Subsumption:

$$\emptyset \subset \delta_B(jTerminal, M) \subseteq \delta_B(jTerminal, M').$$

- Strict Subsumption:

$$\emptyset \subset \delta_S(jTerminal, M) \subseteq \delta_S(jTerminal, M').$$

- Loose Subsumption:

$$\emptyset \subset \delta_L(jTerminal, M) \subseteq \delta_L(jTerminal, M').$$

To this effect, we must compute the differentiator sets

$\delta_B(jTerminal, M)$,

$\delta_S(jTerminal, M)$,

$\delta_L(jTerminal, M)$

for all 31 mutants selected above, with respect to *jTerminal*.

Note that this experiment is artificial in the sense that whereas the strict and loose definitions of differentiator sets can be applied to the same combination of program and test suite, the basic definition can only be applied when we know or assume, that the base program and all the mutants converge for all the elements of the test suite. In the case of *jTerminal* and its mutants, this assumption does not hold, as virtually all of them fail to converge on at least some elements of T. We obviate this difficulty by considering that divergence is itself an execution outcome, but this is merely a convenient assumption for the sake of the experiment.

By computing the basic, strict, and loose differentiator sets of all the mutants with respect to *jTerminal* and comparing them for inclusion, we derive the subsumption relations between the mutants, which we can represent by graphs; these graphs are given in, respectively, Figures 2.3, 2.4 and 2.5. Nodes in these graphs represent mutants, and arrows represent subsumption relations: whenever there is an arrow from a mutant M to mutant M', it means that M subsumes M' (hence, M' can be eliminated from the mutant set without affecting its effectiveness). When two mutants subsume each other (for example $M27$ and $M28$ in 2.4), this means that though these mutants are distinct from each other (they compute functions functions), they have the same differentiator set with respect to *jTerminal*.

From these graphs, we derive minimal mutant sets by selecting the maximal nodes in the subsumption ordering. Once we have the minimal mutant sets, we derive minimal test suites that kill all the mutants in these sets. We verify, in each case, that the test suites that kill all the mutants of the minimal mutant sets actually, kill all the 48 non-equivalent mutants derived from our experiment; this comes as no surprise since this is precisely the rationale for deleting subsumed mutants.

For strict subsumption, for example, we find the following minimal mutant set: `m22, m23, m27, m28, m44, m45, m48, m50, m51, m54, m56, m61, m83, m92, m93`.

Using this mutant set, we derive minimal test suites that kill all these mutants, we find 6 minimal test suites of size 7:

```
Suite 1:  {t7,t16,t18,t20,t21,t22,t25}
Suite 2:  {t7,t16,t18,t20,t21,t22,t26}
Suite 3:  {t16,t18,t20,t21,t22,t23,t25}
Suite 4:  {t16,t18,t20,t21,t22,t25,t27}
Suite 5:  {t16,t18,t20,t21,t22,t23,t26}
Suite 6:  {t16,t18,t20,t21,t22,t26,t27}
```

By virtue of subsumption, these test suites kill all 31 mutants selected above; by virtue of equivalence, they necessarily kill all 48 killable mutants of *jTerminal*.

Using the basic interpretation of subsumption, we find 96 minimal test suites, all of them of size 12; for the loose interpretation of subsumption, we find 48 minimal test suites, all of them of size 11. Due to space limitations, we do not include these test suites. Suffice it to say that their number and their size are vastly different from those found under the strict interpretation.

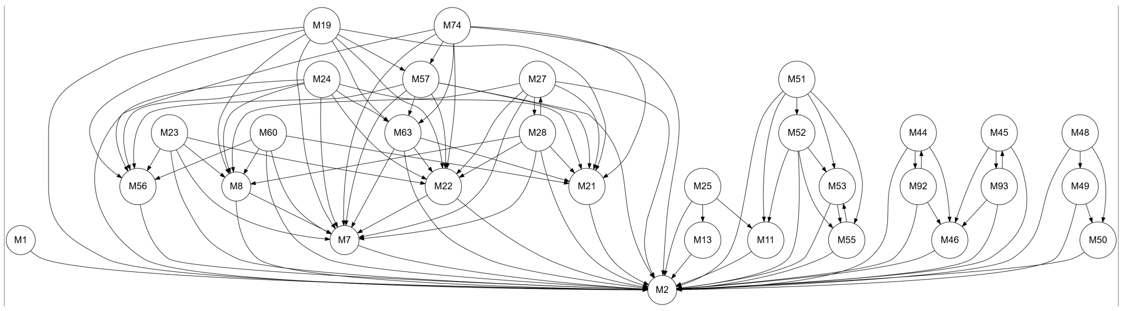


Figure 2.3 Due to Samia et al.,(2022): Basic subsumption graph, jTerminal mutants.

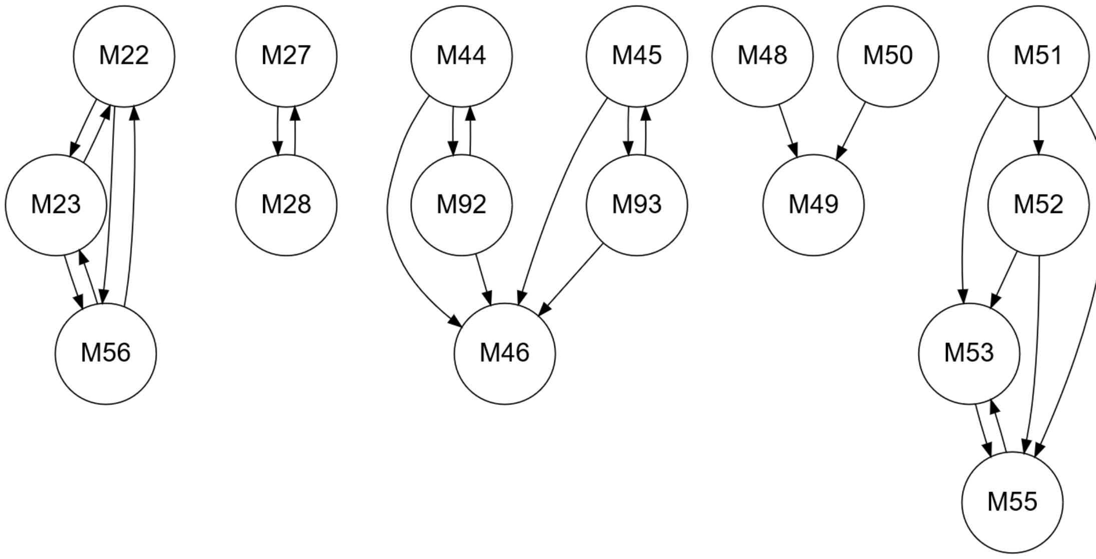


Figure 2.4 Strict subsumption graph, jTerminal mutants.

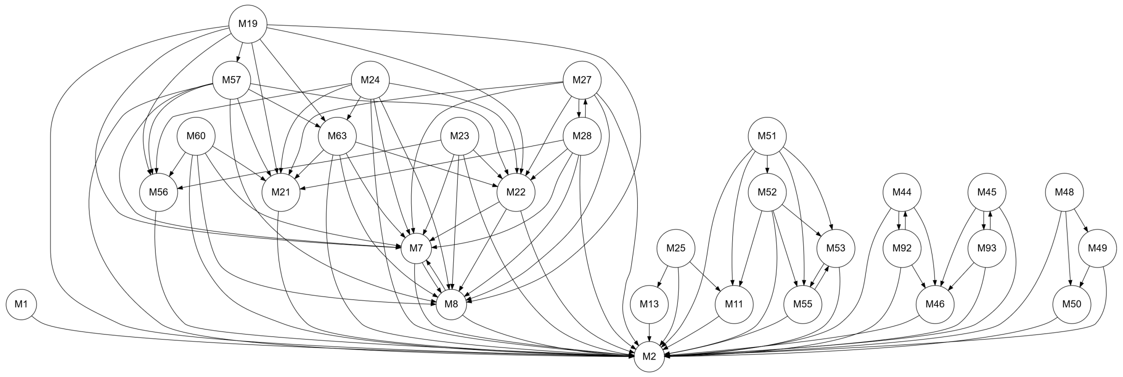


Figure 2.5 Loose subsumption graph, jTerminal mutants.

2.6 Concluding Remarks

2.6.1 Summary

we consider the definition of mutant subsumption, and we resolve to generalize it by taking into consideration the possibility that the execution of programs or their mutants may diverge. The possibility of divergence raises the question of what is the outcome of an execution (is divergence an outcome or the absence of outcome), under what conditions can we compare two outcomes (can we compare the outcome of a program that converges with that of a program that diverges? can we compare the outcome of two programs that diverge?), and under what condition can we consider that two comparable outcomes are identical or distinct (when two programs diverge, do they have the same outcome?). We argue that the definition of subsumption varies according to how we answer these questions, and we identify three possible formulas for subsumption based on three sensible interpretations of these questions.

We further argue that considering the possibility of divergence is not a mundane academic exercise, but an important consideration in mutation testing, as several mutation operators are prone to trigger divergence, even when the base program makes careful provisions to avoid it. Also, we find that once we admit the possibility that programs may define a partial function (i.e., that they may diverge for some initial states), then there is no difference between true subsumption and dynamic subsumption: dynamic subsumption with respect to a program P and test suite T is the equivalent to true subsumption with respect to the pre-restriction of P to T.

We also find, interestingly, that the property of mutant subsumption with respect to a base program P is equivalent to the property of relative correctness with respect to the function of program P interpreted as a specification. The main difference is that relative correctness culminates in absolute correctness, which characterizes the candidate programs at the top of the relative correctness graph; these are the absolutely correct programs. By contrast, due to condition P1, mutant subsumption culminates at the layer immediately below the top of the graph; these are the maximally stubborn mutants (whose differentiator sets are singletons).

We generalize the definition of mutant subsumption by modeling the subsumption of mutant M over mutant M' with respect to program P by an equation of the form:

$$\emptyset \subset \delta(P, M) \subseteq \delta(P, M'),$$

where $\delta(,)$ is the differentiator set function, for three possible definitions of $\delta(,)$. Both $\delta(P, M)$ and $\delta(P, M')$ are subsets of T. This model enables us to reason about the probability of occurrence of subsumption relations: Given that differentiator sets are subsets of T, there are $2^{|T|} - 1$ possible non-empty differentiator sets; using combinatorial formulas, we can estimate the probability that two random subsets of T have an inclusion relation; then we can use this probability to estimate the number of arcs in a subsumption graph and the number of maximal nodes in a subsumption graph. These quantities can be estimated in terms of the cardinality of T and the number of mutants; this is the subject of Subsection 2.4.2.

In Section 4.4, we show empirical evidence of the effect that the distinctions we make between basic subsumption, strict subsumption and broad (aka loose) subsumption are meaningful. In the benchmark example we consider in this section, the three definitions of differentiator sets yield three distinct definitions of what it means to kill a mutant, three distinct definition of mutant subsumption, and three distinct subsumption graphs.

2.6.2 Critique

It is important to acknowledge that when we talk about execution outcomes, we leave much room for interpretation: not only does divergence raises a host of issues about what constitutes the outcome of an execution, but even when an execution converges, it is not always clear what constitutes the outcome of the execution. To clarify this matter, it helps to adopt a homogeneous model, based on state spaces and a mapping from initial states to final states, or a heterogeneous model based on a mapping from an input stream to an output stream. It is also important to specify which definition of differentiator set one adopts, as that determines many important parameters of the analysis.

The analogy between mutant subsumption and relative correctness is interesting, but to ensure that it is useful beyond academic curiosity, we need to explore how advances in each branch can benefit the other branch.

The statistical study of Subsection 2.4.2 is interesting but it raises a paradox: intuitively if we take two random subsets of a set T, the probability that one of them is a subset of the other decreases very quickly with the cardinality of T. This seems to suggest that subsumption graphs ought to have very few arcs between mutants, but most published subsumption graphs are very dense, i.e., have many arcs for the number of nodes they have [15, 16, 29–33]. The only possible explanation for this discrepancy is that the differentiator sets of mutants generated from a base program are not random, but it is difficult to imagine what statistical relation may hold between the differentiator sets of two mutants of the same program.

CHAPTER 3

MINIMIZING MUTANT SETS BY EQUIVALENCE AND SUBSUMPTION

3.1 Minimizing Mutant Sets

Mutation testing is a reliable way to assess the effectiveness of test suites, but it is also an expensive proposition. As a consequence, it is sensible to try to reduce the size of the mutant sets, without loss of effectiveness. Two broad families of criteria are used for the purpose of minimizing mutant sets:

- *Subsumption* [15,16,29–33,40]. A mutant M is said to *subsume* a mutant M' if and only if any test that kills M also kills M', and there exists a test that kills M. The subsumption criterion provides that if M subsumes M' then M' can be removed from the set of mutants.
- *Equivalence* [14]. In [14] Marsit et al. consider the equivalence relation of *semantic equivalence* between mutants, and resolve to derive a minimal set of mutants as a set that includes one element from each equivalence class.

In this chapter, we consider these two policies of mutant set minimization and compare them analytically and empirically. While subsumption is defined as an ordering relation between individual mutants, we argue that it is best viewed as an ordering relation between *equivalence classes* of mutants (modulo semantic equivalence): Indeed, if M subsumes M' and there are ten mutants that are equivalent to M and ten that are equivalent to M', we are still looking at a single instance of the ordering relation, not 100 instances.

The implication of this remark is that subsumption ought not be applied as an alternative to equivalence, but rather alongside equivalence: We must first identify equivalence classes of mutants modulo semantic equivalence, then identify which equivalence classes are maximal by subsumption, and select a representative from each maximal equivalence class. This raises the question: if we have reduced a set of mutants to one representative per equivalence class, how much more reduction do we achieve by applying the criterion of subsumption? As a corollary of this question, it is also legitimate to ask: is the extra reduction in the set of Are mutants commensurate with the effort and risk of subsumption?

The criteria of mutant set minimization by equivalence and by subsumption suffers from another flaw: they both fail to *explicitly* specify a constraint under which the minimization is attempted. Indeed, all optimization problems aim to maximize or minimize an objective function under some constraints: The *Knapsack Problem* aims to maximize some benefit function under the constraint that the capacity of the knapsack is bounded; the *Linear Programming* problem aims to maximize some linear objective function under some affine constraints on the system parameters; the *Maximum Flow* problem aims to maximize the flow through a flow network subject to the topology of the network and the constraint is that each arc has a limited capacity, etc. Of course, we consider that the minimization of mutant sets assumes implicitly that discarded mutants do not reduce the effectiveness of the mutant set, but in the absence of an explicit definition of what is the effectiveness of a mutant set and how to quantify it, it is difficult to make the case that the minimization algorithms are sound. This question is discussed in Section 3.3.

Another question that is raised by the use of subsumption as a criterion for mutant set minimization is the fact that the definition of subsumption is based on the outcome of programs and mutants being different. In order to give a precise meaning to this definition, we must agree on what is the outcome of a program and when we say that two outcomes are identical or distinct. This is less clear-cut than it may appear:

- *What is a program's outcome*? If a program or a mutant fails to terminate due to an infinite loop, a division by zero, or an array reference out of bound, do we consider these to be legitimate outcomes? Or do we define the outcome of a program only when the Program's execution terminates normally?

Also, even when a program does terminate normally, it is not always clear what we consider to be its outcome: is it its final state or the output that the program delivers as a projection of the final state? For example, what is the program's outcome if a program permutes two variables x and y using an auxiliary variable z? Is it the final values of x, y, and z, or just the final values of x and y?

- *When do we say that two outcomes are identical*? If two programs terminate normally for some common input, then (assuming we agree on what variables represent the program's outcome) we can tell whether they have the same outcome. But what about if one of them converges and the other fails to converge? Do we assume that they have distinct outcomes or that their outcomes cannot be compared? What about the case when two programs fail to converge? Do we consider that they have the same outcome (failure to converge) or that their outcomes are incomparable?

This matter will be discussed in this chapter, and subsumption will be (re) defined accordingly.

Using these concepts, we revisit the definition of subsumption and generalize it in Section 3.2 in several ways: first by revisiting the concept of program outcome and the criterion of outcome equality (or diversity) in light of the possibility that programs and mutants may fail to converge for a given input; and second, by redefining the subsumption between mutant equivalence classes as an inequality between differentiator sets.In Section 3.3, we consider metrics for mutant effectiveness that may be used as constraints under which mutant set minimization is attempted. The models introduced in Section 3.2, can also be used to estimate statistically, the frequency of subsumption relations between equivalence classes, and the expected number of maximal equivalence classes in the subsumption graph; this is done in Subsection 2.4.2. In Section 3.4, we consider a sample example of a mutation experiment to which we apply minimization by subsumption and equivalence, and assess their results. We conclude in Section 5.4, with some observations and draw some preliminary lessons.

3.1.1 Detector sets

An ideal test suite is one that we can rely on to prove correctness: If program P runs successfully on test suite T, we want to be able to infer that P is correct; equivalently, we want that if P is incorrect, then testing P on test suite T ought to expose a failure of P. This leads us to the concept of *detector set*, i.e., the set of all the initial states on which program P violates its specification. This set is important because it enables us to characterize ideal test suites: ideal test suites are supersets of the program's detector set.

But before we define detector sets, we must consider that there are two definitions of correctness, and these yield two distinct interpretations of what it means for a program to fall short of the standard of correctness; therefore, there are two possible definitions of detector sets, depending on what standard of correctness we adopt. We consider two definitions of program correctness: *total correctness* [25, 28] and *partial correctness* [27]. These are given in Chapter 1.

Definition 7. *Due to [41] Given a program P on space S and a specification (relation) R on S, P is said to be* totally correct *with respect to R if and only if:*

$$dom(R) = dom(R \cap P).$$

Due to [1] Given a program P on space S and a specification (relation) R on S, P is said to be partially correct *with respect to R if and only if:*

$$dom(R) \cap dom(P) = dom(R \cap P).$$

These definitions are equivalent, modulo differences of notation, to the traditional definitions of total and partial correctness [25, 27, 28]. The domain of $(R \cap P)$

is the set of initial states on which P satisfies R; we refer to it as the *competence domain of P with respect to R*. Since total correctness is a stronger property than (logically implies) partial correctness, we expect the set of tests that disprove the former to be a superset of the set of tests that disprove the latter. We adopt the definitions of detector sets given in [37], hence, we content ourselves in this chapter with introducing these definitions and briefly commenting on them.

Definition 8. *Due to [37] Given a program P on space S and a specification R on S, the* total detector set *of P with respect to R is the set denoted by $\Theta_T(P,R)$ and defined as the set of initial states on which execution of P produces an outcome that disproves the total correctness of P with respect to R (either the execution fails to converge or it does converge but produces a final state s' such that $(s,s') \notin R$).*

Given a program P on space S and a specification R on S, the partial detector set *of P with respect to R is the set denoted by $\Theta_P(P,R)$ and defined as the set of initial states on which execution of P produces an outcome that disproves the partial correctness of P with respect to R (the execution converges but produces a final state s' such that $(s,s') \notin R$).*

When we want to refer to a detector set and do not wish to specify to which one we refer, we use the notation $\Theta(P,R)$. The following proposition, due to [37] gives explicit expressions of the detector sets.

Proposition 4. *Given a program P on space S and a specification R on S, the total detector set and the partial detector set of P with respect to R are given by the following formulas:*

$$\Theta_T(P,R) = dom(R) \cap \overline{dom(P \cap R)}.$$

$$\Theta_P(P,R) = dom(P) \cap dom(R) \cap \overline{dom(P \cap R)}.$$

See Figure 3.1. A test suite T disproves the total (respectively, partial) correctness of P with respect to R if and only if (respectively):

$$T \cap \Theta_T(R, P) \neq \emptyset.$$

$$T \cap \Theta_P(R, P) \neq \emptyset.$$

If a test suite T disproves a correctness property, then so does any superset thereof. See Figure 3.2; test suites $T0$, $T1$ and $T2$ do not disprove correctness; TP disproves partial correctness; hence, also total correctness; TT disproves total correctness but not partial correctness. According to the definitions of total and partial correctness (Definition 7) we can easily prove the following Proposition.

Proposition 5. *A program P is totally (resp. partially) correct with respect to a specification R if and only if its total (resp. partial) detector set is empty.*

3.2 Revisiting Subsumption

In this section, we use differentiator sets to characterize the subsumption relation, then to generalize this relation into two ways:

- First, by considering the case where the base program or the mutants fail to converge for some initial states.
- Second, by defining subsumption not between individual equivalence classes of mutants, but between sets of equivalence classes.

3.2.1 Subsumption of convergent programs

We consider a program P on space S and two mutants M and M' of P, and we assume that P, M and M' converge for all s in S. According to [15, 16], mutant subsumption is defined as follows.

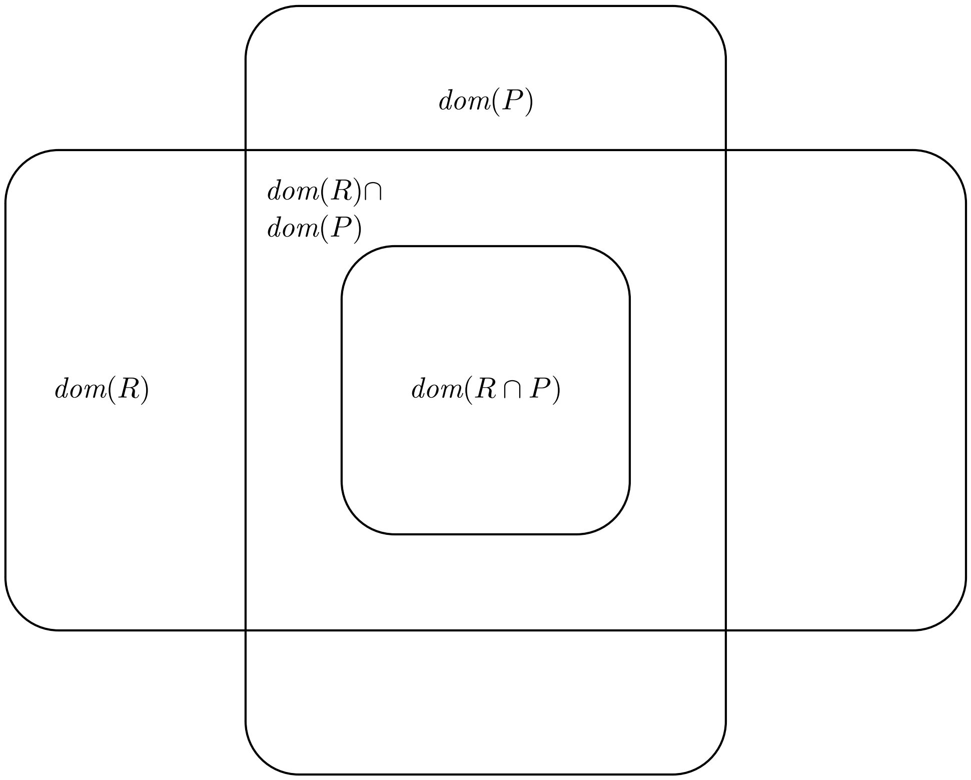


Figure 3.1 Test data to disprove correctness.

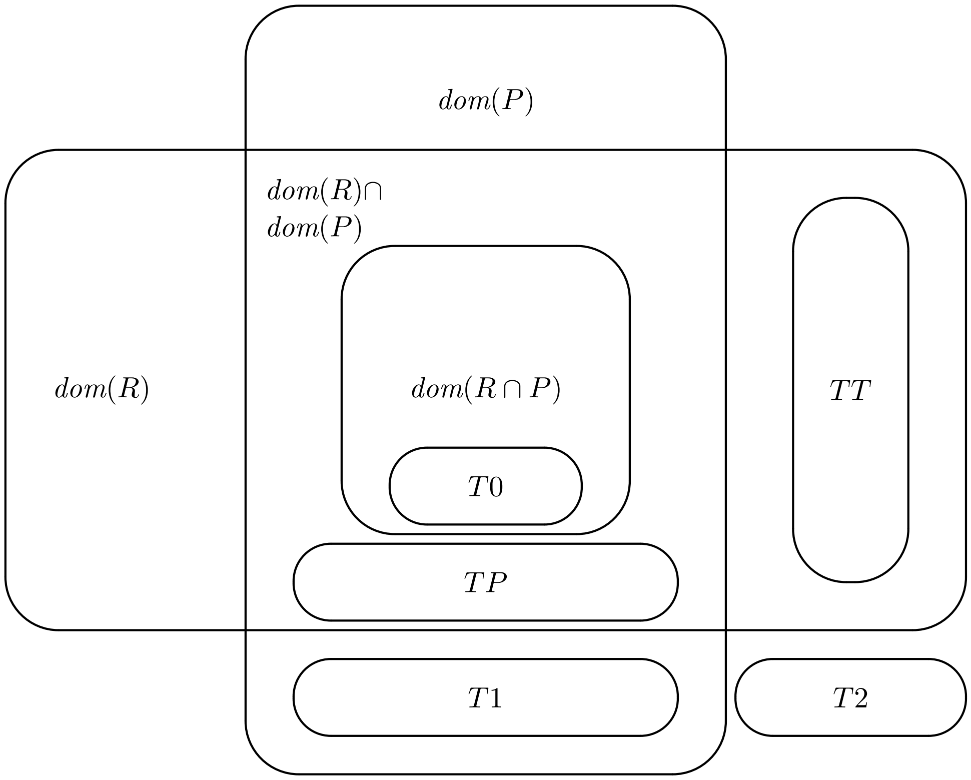


Figure 3.2 Sample test suites.

Definition 9. *Given a program P on space S and two mutants M and M' of P, we say that M* subsumes *M' with respect to P if and only if:*

- *There exists an initial state s in S such that P and M computes different outcomes.*
- *For all initial states s in S, if M computes a different outcomes from P on s, then so does M'.*

The following Proposition gives a simple characterization of mutant subsumption, using differentiator sets.

Proposition 6. *Given a program P on space S and two mutants M and M', M subsumes M' with respect to P if and only if:*

$$\emptyset \subset \delta_0(P, M) \subseteq \delta_0(P, M').$$

The proof of this Proposition is due to [38].

3.2.2 Considering divergence

Failure to converge is a condition that arises often, not least in mutation testing; for example, if we have a loop that visits all the cells of an array between indices 0 and $N-1$ using the condition `while (i<N)`, and we change the condition of the loop from $<$ to $\leq$ (a common mutation operator), the mutant we generate will raise an exception (array reference out of bound) and fail to terminate normally. Hence, it is sensible to (re-) define subsumption in a way that makes provisions for the possibility that the program or its mutants may diverge for some initial states. To this effect, we use the more general definitions of differentiator sets, namely $\delta_1(P, M)$ and $\delta_2(P, M)$.

Definition 10. *Given a program P on space S and two mutants M and M' of P, we say that M* subsumes *M' with respect to P if and only if:*

$$\emptyset \subset \delta(P, M) \subseteq \delta(P, M').$$

We use $\delta(,)$ as a surrogate for any of the differentiator sets we have introduced in Section 3.1: $\delta_0(,)$, $\delta_1(,)$, $\delta_2(,)$. A user may select a function among these, depending on their interpretation of program outcomes, and when they consider that two outcomes are comparables, and if they are when they consider that two outcomes are identical. Note that even though we define subsumption as if it were a relation between individual mutants, we refer to classes of equivalence of mutants modulo semantic equivalence. Since we are taking a semantic approach, we do not distinguish between mutants that compute the same function on S, even if they are syntactically distinct.

3.3 Mutant Set Effectiveness

The problem of minimizing a set of mutants is, at its core, an optimization problem, except that the way it is formulated in the literature does not lend itself to an optimization model. Indeed, an authentic optimization includes an objective function to optimize (minimize or maximize) along with a set of constraints under which this optimization is attempted; if it were not for the constraints, the minimal mutant set would be the empty set. But, of course, that is not the intent: the intent is to find a minimal set of mutants that has the same effectiveness as the original set of killable mutants. This raises the question: How do we define (or measure or characterize) the effectiveness of a set of mutants?

As a first cut, we can consider that the effectiveness of a set of mutants is measured by its ability to vet/ identify compelling test suites.

If we assume that the function of a set of mutants is to assess the quality of test suites, then its effectiveness can be measured by the quality/ effectiveness of test suites that it vets. This leads us to ponder how to measure the quality of a test suite. To this effect, we introduce the concept of semantic coverage of a test suite.

Definition 11. *Given a program P on space S, a specification R, and a test suite T (a subset of S), the* semantic coverage *of test suite T of program P with respect to R is denoted by $\Gamma(T,P,R)$ and defined as:*

$$\Gamma(T,P,R) = T \cup \overline{\Theta(P,R)},$$

where $\Theta(P,R)$ is the (total or partial) detector set of P with respect to R.

The semantic coverage of a test suite T ranges from the empty set to all of S:

- It equals the empty set when T is empty and the detector set $\Theta(P,R)$ equals S; in other words, even though any initial state exposes the program's failure, T does not expose any failure because it is empty; that is the worst kind of test suite.
- Conversely, this quantity takes the maximal value of S when $\Theta(P,R)$ is a subset of T; in other words, T detects all the failures of P with respect to R (failure to be correct or failure to be partially correct, depending on which version of $\Theta(P,R)$ we choose). That is the best kind of test suite, the smallest test suite that has a maximal semantic coverage ($\Gamma(T,P,R) = S$) is the detector set of P with respect to R: it is $T = \Theta(P,R)$; it includes all the initial states that detect failure, and nothing else.

As a simple explanation of this formula, consider that the complement of $\Gamma(T,P,R)$ can be written as:

$$\Theta(P,R) \cap \overline{T}.$$

This set represents the elements of $\Theta(P,R)$ that are not in T, in other words, the initial states that expose faults in P but are not in T; clearly, the fewer such elements

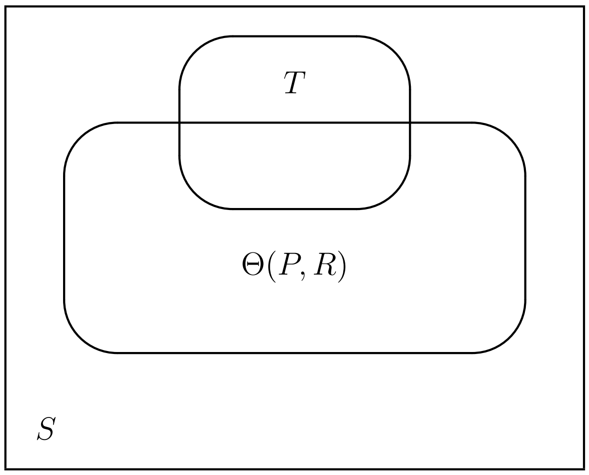


Figure 3.3 Semantic coverage of T with respect to R.

we have, the better (hence, the larger their complement, the better). See Figure 4.5. The semantic coverage of test T with respect to specification, R is the complement of the intersection of $\Theta(P,R)$ with the complement of T.

In [42], Binksma et al. critique syntactic measures of test coverage and define an alternative measure of test coverage that is based on program semantics; we know of no other work that attempts to define semantic measures of test coverage.

Now that we know how to measure the quality of test suites, we use this metric to measure the quality of a set of mutants. We want to consider that the quality of a set of mutants is defined by the quality of the test suites that it vets/ selects; also, we consider that a test suite is selected by a set of killable mutants if and only if the test suite kills all the mutants of the set. With this in mind, we introduce below two measures of the quality of a mutant set.

Definition 12. *Given a program P on space S and a specification R on S, and given a set of mutants $\mu = \{M_1, M_2, ..., M_k\}$, we define two measures of quality of the mutant set μ:*

- Assured Quality.

  $Q_A(\mu) = \bigcap_{M \in \mu \wedge T \cap \Theta(P,M) \neq \emptyset} \Gamma(T,P,R).$

- Potential Quality.

  $Q_P(\mu) = \bigcup_{M \in \mu \wedge T \cap \Theta(P,M) \neq \emptyset} \Gamma(T, P, R)$.

The condition $T \cap \Theta(P, M) \neq \emptyset$ means that test suite T kills mutant M. So that $Q_A(\mu)$ is the minimal semantic test coverage of a test suite T that kills all the mutants of μ, and $Q_P(\mu)$ is the maximal semantic test coverage of a test suite T that kills all the mutants of μ. We use metric $Q_A(\mu)$ when we are interested to minimize risk and we use metric $Q_P(\mu)$ when we are interested to maximize potential reward. When we want to refer to the quality of a set of mutants without specifying which metric we intend, we use the notation $Q(\mu)$.

Given the function $Q(\mu)$, we can now (re) formulate the problem of minimizing a set of mutants as follows.

Definition 13. *Given a program P on space S, a specification R on S and a set of mutants $\mu = \{M_1, M_2, ..., M_k\}$ that are killable concerning R. Minimizing the set of mutants μ consists in finding a subset μ' of μ that satisfies the following conditions:*

- $Q(\mu') = Q(\mu)$.
- *μ' is minimal concerning the above property, i.e. any proper subset of μ' violates this property.*

It is easy to prove that if μ contains two mutants M_i and M_j that are semantically equivalent, then one of them can be removed without affecting the quality metric $Q(\mu)$. Whether the same can be said of the subsumption criterion, we can neither claim nor (much less) prove; we leave this for further research.

3.4 Minimizing a Mutant Set for jTerminal

We consider the Java benchmark program of *jTerminal*[1], an open-source software product routinely used in mutation testing experiments [30]. We apply the mutant generation tool *LittleDarwin* in conjunction with a test generation and deployment

[1]available online at `http://www.grahamedgecombe.com/projects/jterminal`.

class that includes 35 test cases [30]. All our analysis of mutant equivalence, mutant redundancy, mutant survival, etc is based on the outcomes of programs and mutants on this test suite (and carefully selected subsets thereof). For differentiator sets, we adopt the broad definition $\delta_2(P, M)$; hence, we consider that failure to converge is a legitimate execution outcome and that failure to converge is comparable to a normal outcome and is distinct from that place. Execution of LittleDarwin on jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants; for the sake of documentation, we list them below:

```
m1, m2, m7, m8, m9, m10, m11, m12, m13,
m14, m15, m16, m17, m18, m19, m21, m22,
m23, m24, m25, m26, m27, m28, m44, m45,
m46, m48, m49, m50, m51, m52, m53, m54,
m55, m56, m57, m58, m59, m60, m61, m62,
m63, m83, m88, m89, m90, m92, m93.
```

The remaining 46 mutants are semantically equivalent to the pre-restriction of jTerminal to the selected test suite. In this section, we generate a minimal mutant set out of the 48 mutants using respectively the criterion of equivalence and the criterion of subsumption.

3.4.1 Minimal mutant set by equivalence

The procedure for generating a minimal mutant set is outlined in [14] provides for executing the following steps:

- Parse the source code of jTerminal to compute its redundancy metrics [43]: State redundancy (SR_I, SR_F); functional redundancy (FR); Non Injectivity (NI).
- Use the redundancy metrics to estimate the REM (Rate of Equivalent Mutants) of jTerminal: $REM = f(SR_I, SR_F, FR, NI)$.

- Use the REM of jTerminal to estimate the number of equivalence classes of the set of mutants modulo semantic equivalence: $K = NEC(N, REM)$, where N is the number of (killed) mutants.
- Using K and N, estimate the expected number of mutants that we must inspect (among N) before we encounter K distinct mutants: $H = NOI(N, K)$.
- Inspect the mutants one by one, comparing them against previously inspected mutants, until we find K distinct mutants or we inspect H mutants in total.
- We adopt the resulting set of distinct mutants as a minimal set of mutants that preserves (approximately) the same functionality as the original set of N mutants.

Because, in this case, the number of mutants is not very large, and because we want to obviate the uncertainties that stem from estimating the redundancy metrics, then REM, then K, then H, we resolve to inspect all 48 mutants and compare them to each other to find K distinct mutants. We find that out of the 48 mutants under consideration, the following 30 are distinct from each other. We find $\mu_E =$

`m1, m2, m7, m11, m13, m15, m19, m21, m22,`

`m23, m24, m25, m27, m28, m44, m45, m46, m48,`

`m49, m50, m51, m52, m53, m55, m56, m57, m60,`

`m63, m92, m93.`

We compute the differentiator set of each of these 30 mutants, which we use to derive minimal test suites that kill all the mutants. To this effect, we record the differentiator sets on a two-dimensional array where the mutants are represented in columns, and the test data are represented in rows. We iterate through the following two steps until the array is empty.

- We select the data that kills the most mutants.
- , we remove the row corresponding to the selected data and the columns of all the mutants that the data kills.

Because there are several instances where more than one row has the same maximal number of mutants, we may (and typically do) generate several minimal test suites.

We list ten minimal test suites generated according to this procedure; the numbers refer to the line of code where the data is generated in the original test class; for our purposes, these numbers uniquely identify the test data. Interestingly, all these sets have exactly 11 elements. We find:

TE1= $\{t90, t118, t133, t168, t185, t189, t191, t209, t215, t239, t280\}$.

TE2= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.

TE3= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t284\}$.

TE4= $\{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.

TE5= $\{t90, t118, t133, t168, t185, t189, t191, t203, t209, t239, t280\}$.

TE6= $\{t90, t118, t133, t168, t185, t189, t191, t209, t215, t239, t280\}$.

TE7= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t284\}$.

TE8= $\{t90, t118, t133, t168, t185, t189, t191, t203, t207, t239, t284\}$.

TE9= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t241, t280\}$.

TE10= $\{t90, t118, t133, t168, t185, t189, t191, t209, t215, t241, t284\}$.

By construction, this test suite kills the 30 mutants of the minimal mutant set. Because the mutants outside the minimal mutant set are semantically equivalent to mutants of the set, the test suites above also kill the 48 killable mutants.

In order to facilitate comparisons with the set of minimal test suites derived by subsumption, we isolate the elements that are common to all the test suites, namely:

$$T0 = \{t90, t118, t133, t168, t185, t189, t191\},$$

then we can rewrite these test suites as:

TE1= $T0 \cup \{t209, t215, t239, t280\}$.

TE2= $T0 \cup \{t114, t209, t239, t280\}$.

TE3= $T0 \cup \{t114, t209, t241, t284\}$.

TE4= $T0 \cup \{t207, t215, t239, t280\}$.

TE5= $T0 \cup \{t203, t209, t239, t280\}$.

TE6= $T0 \cup \{t209, t215, t239, t280\}$.

TE7= $T0 \cup \{t114, t209, t241, t284\}$.

TE8= $T0 \cup \{t203, t207, t239, t284\}$.

TE9= $T0 \cup \{t114, t209, t241, t280\}$.

TE10= $T0 \cup \{t209, t215, t241, t284\}$.

We notice that TE1 and TE6 are identical; different selections made when two or more test data kill the same number of mutants may ultimately yield the same minimal test suite.

3.4.2 Minimal mutant set by subsumption

To apply the subsumption criterion, we consider a representative from each of the 30 equivalence classes of the 48 killable mutants and test them pairwise by comparing their broad differentiator sets ($\delta_2(P, M)$). Then, we isolate the maximal mutants, i.e. those that are not subsumed by any other mutants. We find the following minimal set of mutants: μ_S ={`m1, ,19, ,23, m24, m25, m27, m44, m45, m48, m51, m60.`}

We compute the broad differentiator sets of these mutants, which are (by construction) much smaller than those of the mutants selected by equivalence; We apply the same procedure above to derive minimal test suites that kill all these mutants. We find:

TS1= $\{t90, t114, t118, t133, t168, t185, t189, t191, t207, t239, t284\}$.

TS2= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.

TS3= $\{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.

TS4= $\{t90, t118, t133, t168, t185, t189, t191, t209, t215, t241, t280\}$.

TS5= $\{t90, t118, t133, t168, t185, t189, t191, t207, t215, t239, t280\}$.

TS6= $\{t90, t118, t133, t168, t185, t189, t191, t203, t209, t241, t284\}$.

TS7= $\{t90, t118, t133, t168, t185, t189, t191, t203, t207, t239, t280\}$.

TS8= $\{t90, t114, t118, t133, t168, t185, t189, t191, t207, t241, t280\}$.

TS9= $\{t90, t114, t118, t133, t168, t185, t189, t191, t209, t239, t280\}$.

TS10= $\{t90, t118, t133, t168, t185, t189, t191, t203, t207, t241, t284\}$.

We remove the common elements so as to facilitate comparisons if we let $T0$ be the following set:

$$T0 = \{t90, t118, t133, t168, t185, t189, t191\},$$

Then, these sets can be written as:

TS1= $T0 \cup \{t114, t207, t239, t284\}$.

TS2= $T0 \cup \{t114, t209, t239, t280\}$.

TS3= $T0 \cup \{t207, t215, t239, t280\}$.

TS4= $T0 \cup \{t209, t215, t241, t280\}$.

TS5= $T0 \cup \{t207, t215, t239, t280\}$.

TS6= $T0 \cup \{t203, t209, t241, t284\}$.

TS7= $T0 \cup \{t203, t207, t239, t280\}$.

TS8= $T0 \cup \{t114, t207, t241, t280\}$.

TS9= $T0 \cup \{t114, t209, t239, t280\}$.

TS10= $T0 \cup \{t203, t207, t241, t284\}$.

A cursory inspection of the minimal test suites generated from the minimal mutant set derived by equivalence and the minimal mutant set derived by subsumption reveals that some of the test suites are identical. For example, $TE2$ is identical to $TS2$; and $TE4$ is identical to $TS3$. It is possible, even likely, that the set of all the minimal test suites that kill all the mutants of μ_E is the same as the set of all the minimal test suites that kill all the mutants of μ_S. To be certain, we need to generate all the minimal test suites for each mutant set. Nevertheless, the fact that the sets of minimal test suites have several elements in common is noteworthy.

3.5 Concluding Remarks

In this chapter, we have considered two policies for minimizing a set of mutants and tried to analyze them and compare them using analytical and empirical arguments. Some of the premises of our comparative study includes the following:

- Subsumption is not a relation between individual mutants; rather, it is a relation between equivalence classes of mutants, modulo semantic equivalence.
- As a consequence of this premise, the subsumption policy is not orthogonal to the equivalence policy; rather, it must be mindful /cognizant of the equivalence relation,

and must identify equivalence classes prior to identifying subsumption relations between classes.

- Minimizing a set of mutants is, at its core, an optimization problem; as such, it must be formulated in such a way as to clearly specify an objective function that we want to optimize along with the set of constraints under which we want to optimize the objective function. In mutant set minimization, the objective function is clear (the size of the mutant set), but the constraints under which the optimization is carried out have not been clearly specified. Implicitly, we want to minimize the cardinality of the mutant set while preserving the utility of the original set of mutants. This, in turn, requires that we define/ quantify the utility of a mutant set.
- The theoretical formulas of the Section 3.3 notwithstanding, we can characterize the utility/ quality of a set of mutants by the minimal test suites that kill all the killable mutants.
- If we quantify the effectiveness of a mutant set by the minimal test suites that it vets, then the empirical study of the Section 3.4 is a resounding endorsement of subsumption, since it appears to vet the same test suites with one-third of the size (nine mutants vs thirty), and the test suites it vets kill all the (48) killable mutants of the base program.
- The results of Section 3.4, to the extent that they are valid (a tenuous stretch, given their tentative/ partial/ incomplete nature) may also be interpreted to mean that if the set μ_E vets the same minimal test suites as μ_S, then it may be sufficient to generate μ_E.
- The premise that the effectiveness of a test suite is not an attribute of the program and its mutants alone, but also involves the specification with respect to which the program is supposed to be correct. This is the perennial question of whether mutants are or are not a good representation of actual faults [7, 35, 36]; our discussions of Section 3.3 formalize the relation between mutants and faults by a formula that links the differentiator sets of mutants (as proxies for mutations) with the detector sets of programs (as proxies for faults).

Among the contributions of this chapter, we mention:

- A semantic measure of test effectiveness that adheres to the same rationale as Brinksma et al. [42] but uses an innovative formula involving the detector set of a program with respect to a specification R.
- Tentative measures of the effectiveness of a mutant set ($Q_A()$, $Q_P()$), using the semantic measure of test effectiveness of the test suites that are vetted by the mutant set.
- A reformulation of subsumption using differentiator sets and a generalization of subsumption to take into consideration the possibility that the base program or the mutants fail to converge.
- The observation that maximal mutants by subsumption feature minimal differentiator sets and are, in fact, what Yao et al. [44] refer to as *stubborn mutants*.

Future research prospects include completing the experiment of Section 3.4 by computing all the minimal test suites of μ_E and μ_S and comparing them. Also, we envision to apply the generalized definition of subsumption that ranks sets of mutants rather than individual (equivalence classes of) mutants; it would be interesting to see whether this generalized formula enables us to reduce further the minimal set of mutants while preserving its effectiveness.

CHAPTER 4

TEST SUITES EFFECTIVENESS

4.1 On the Effectiveness of a Test Suite

4.1.1 Motivation

If the purpose of test suites is to reveal the presence of faults in programs, then it is legitimate to measure the quality of a test suite by its ability to reveal faults. A necessary condition to reveal a fault is to exercise the code that contains the fault; hence, many metrics of test suite effectiveness focus on the ability of a test suite to exercise the syntactic attributes of the program, such as statements, conditions, branches, paths, etc [45]; but while achieving syntactic coverage is necessary, and it is far from sufficient, and not always possible. Indeed, not all faults cause errors, and not all errors lead to observable failures [46]; also, not all syntactic paths are feasible (re: infeasible paths, unreachable code, etc.). A better measure of test suite quality is mutation coverage, where the quality of a test suite is measured by the ratio of mutants that it kills out of a set of generated mutants. But while mutation coverage is often used as a baseline for assessing the value of other coverage metrics [35,47], it has issues of its own: the same mutation score may mean vastly different things depending on whether the killed mutants are all distinct, all equivalent, or something in between (partitioned into several equivalence classes); if a test suite T kills N mutants, what we can infer about T depends to a vast extent on whether T killed N different mutants or N times the same mutant (i.e., N equivalent versions of the same mutant). Also, the same test suite T may yield different mutation scores for different sets of mutants; hence, it cannot be considered as an intrinsic attribute of the test suite.

In this chapter, we present a measure of test suite effectiveness which depends only on the program under test, the correctness property we are testing it for, and the specification against which correctness is defined. In the next section, we present and justify a number of criteria that measure of test suite effectiveness ought to satisfy, and in Section 5.1.2, we present and justify some design decisions that we resolve to adopt in the process of defining our measure.

In Section 4.2 we introduce detector sets, and discuss their significance for the purposes of the program testing and program correctness. In Section 4.3 we use detector sets to define the concept of *semantic coverage*, and in Subsection 4.3.2 we validate our definition by showing, analytically, that it meets all the requirements outlined in Subsection 4.1.2. In section 4.4 we illustrate the derivation of semantic coverage on a sample benchmark example, and show its empirical relationship to mutation scores. We conclude in Section 4.5 by summarizing our findings, critiquing them, comparing them to related work, and sketching directions of further research.

4.1.2 Requirements of semantic coverage

We consider a program P that we want to test for correctness against a specification R and we wish to assess the fitness of a test suite T for this purpose. We argue that the effectiveness of the test suite T to achieve the purpose of the test ought to be defined as a function of three parameters:

- Program P.
- Specification R.
- The standard of correctness that we are testing P for: partial correctness or total correctness [25, 27, 28].

Hence, whereas most traditional measures of the test suite coverage depends exclusively on the program under test, our definition depends also on the standard of correctness that we are testing the program for, as well as the specification against which correctness is tested.

The requirements we present below dictate how semantic coverage ought to vary as a function of each of these three parameters: the standard of correctness, the specification, and the program. To understand the discussions below, it is helpful to reason by analogy: the effectiveness of a tool to perform a task increases with the intrinsic power of the tool, but it also increases as the task becomes easier.

- *The Program.* Relative correctness is the property of a program to be more correct than another with respect to a specification [48]; each fault repair makes the program more-correct, culminating in absolute correctness when all faults are repaired [49]. The same test suite ought to have greater and greater semantic coverage as the program becomes increasingly more correct with respect to the same specification, as there are fewer and fewer faults left to discover.
- *The Specification.* Specifications are naturally ordered by refinement, whereby more refined specifications represent stronger/ harder to satisfy requirements [23, 24, 50–52]; it is harder to test a program against a more refined specification than against a less-refined specification since a more-refined specification represents a stronger requirement to test against. Hence, the same test suite ought to have greater semantic coverage for less-refined specifications.
- *The Standard of Correctness.* There is a difference between testing a program for total correctness and testing a program for partial correctness: under total correctness, if we select a test data t and the program fails to terminate on t, we conclude that the program fails the test, i.e., is deemed incorrect; under partial correctness, if we select a test data t and the program fails to terminate on t, we conclude that the test data selection is wrong (and we chose another test data). Total correctness is a stronger property than partial correctness; hence, it is more difficult to test a program for total correctness than for partial correctness. Therefore, the same test suite T must have greater semantic coverage if it is applied to partial correctness than if applied to total correctness.

Of course, we also want the effectiveness of a test suite T to increase with T, i.e. if T' is a superset of T then the effectiveness of T' is greater than that of T. In Subsection 4.3.2, we present propositions to the effect that the formula of semantic coverage we present in the Section 4.3 satisfies all these requirements.

4.1.3 Design principles

We resolve to adopt the following design decisions in defining a measure of semantic coverage:

- *Focus on Failure.* We adopt the definitions of fault, error, and failure proposed by Avizienis et al. [46]: a *fault* is a feature of the program that precludes it from being correct; an *error* is the impact of a fault on the state of the program for a particular execution (that sensitizes the fault); a *failure* is the event where the program violates its specification because an error has propagated to the program's output. We quantify the effectiveness of a test suite, not by its ability to reveal faults, but by its ability to reveal failures. The reason is that failures are objectively verifiable observation: by contrast, resolving that some features in the source code of the program is the cause of the observed failure is a subjective assumption, as there is no one-to-one mapping between observed failures and faults.

  Hence, we resolve to define the semantic coverage of a test suite T in terms of the scale of program failures that T can reveal.

- *Partial Ordering.* It is common to think of measurement as the assignment of a numeric value to an artifact to reflect a particular attribute thereof. But assigning numeric values to an attribute that is not totally ordered causes a loss of precision: it is easy to imagine two test suites that cannot be compared (e.g., they expose unrelated sets of failures), yet if we assign them numbers, we will always find that one test suite is assigned a greater number than the other.

  Hence, we resolve to define semantic coverage not as a number, but as an element of a partially ordered set, our goal is to ensure that whenever two test suites have comparable semantic coverages, it is because the test suite with the more excellent coverage is superior (in a sense to be defined) to the other.

- *Analytical Validation.* There are several reasons why we prefer to rely on analytical argument to validate our definition of semantic coverage: First and foremost, we do not know of a widely accepted ground truth of test suite effectiveness against which we can validate our definition. Second, most existing coverage metrics reflect program attributes only, while our semantic coverage definition depends on the correctness standard and the specification, in addition to the program. Hence, we resolve to validate our definition of semantic coverage on the basis of analytical arguments, by arguing that it captures the right attributes and that it satisfies all the properties that we mandate in Subsection 4.1.2.

  Still, we do include an empirical validation step: In Section 4.4, we compute the semantic coverage of a set of (20) test suites of a benchmark program for two distinct specifications and two standards of correctness, and we compare the four graphs so derived against the graph that ranks these test suites by mutation coverage, but we do so without the expectation that the graphs be identical because semantic coverage depends on the program, the specification, and the correctness standard, whereas mutation coverage depends on the program and the mutation generator. As we discuss in the Section 4.4, the similarity we observe empirically between the graphs exceeds our expectation.

4.2 Detector Sets

4.2.1 Requirements on a measure of effectiveness

We present some requirements that we expect our definition of mutant set effectiveness (as well as competing definitions) to satisfy. For the sake of generality, we do not assume that effectiveness is defined as a number; we only assume that it takes values in an ordered set; hence, when we talk about *greater than*, we refer to the unspecified ordering relation of the target set.

RQ1 *Monotonicity with Respect to the Standard of Correctness.* Total correctness is a stronger property than partial correctness; hence, it is easier to test a program for partial correctness than for total correctness. Since the effectiveness of a mutant set is defined in terms of the test suites that it vets, the same dependency holds: a mutant set ought to have higher effectiveness to test for partial correctness rather than total correctness.

RQ2 *Monotonicity with Respect to the Specification.* Specifications are naturally ordered by refinement, whereby a specification R *refines* a specification R' if and only if any program that is correct with respect to R is necessarily correct with respect to R' [23, 24]. It is easier to test a program against a less-refined specification, hence, a mutant set ought to have higher effectiveness when testing a program against a less-refined specification.

RQ3 *Effectiveness and Redundancy.* Whereas adding mutants to a mutant set ought to increase (or at least not decrease) its effectiveness, we require that adding *redundant* mutants does not change the effectiveness of the set; a mutant M is considered to be redundant with respect to a mutant set μ if and only if any test suite T that is vetted by μ kills M.

RQ4 *Effectiveness and Subsumption.* A mutant M is said to subsume a mutant M' if and only if any test (hence, a fortiori, any test suite) that kills M kills M' [15, 16]. Subsumption has been used as a criterion for mutant set minimization and can be viewed as a special case of mutant redundancy [10, 15, 16, 29–33, 40]. We require that removing a subsumed mutant from a mutant set preserves the effectiveness of the set.

The following definition gives criteria of *relative* correctness: a program P' may be more-correct than a program P, which both are incorrect.

Definition 14. *Given programs P and P' on space S, and specification R on S, we say that P' is* more-totally-correct *than P with respect to R if and only if:*

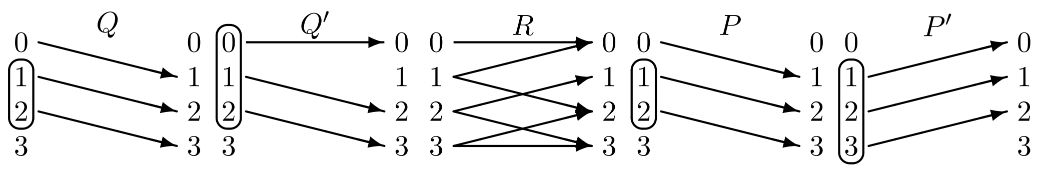
$$dom(R \cap P') \supseteq dom(R \cap P).$$

We say that P' is more-partially-correct *than P with respect to R if and only if:*

$$dom(R \cap P') \cup \overline{dom(P')} \supseteq dom(R \cap P) \cup \overline{dom(P)}.$$

The definition of relative total correctness is due to [48]; the definition of relative partial correctness is derived herein by analogy for the sake of completeness. To contrast the definitions given in the definition 2 with those given herein for relative correctness, we may refer to the former as *absolute correctness.*

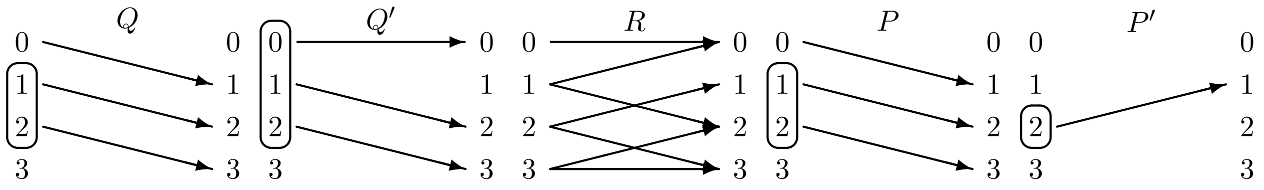
A program P' is more-totally-correct than a program P with respect to R if and only if it has a larger competence domain with respect to R, i.e. it obeys specification R over



Preserving Correct Behavior
$(R \cap Q) \subseteq (R \cap Q')$

Preserving Correctness
$(R \cap P)L \subseteq (R \cap P')L$

Figure 4.1 Relative total correctness.



Q' is more-partially-correct than Q
by virtue of having a larger competence domain
$dom(R \cap Q') \cup \overline{dom(Q')} = \{0, 1, 2, 3\}$
$dom(R \cap Q) \cup \overline{dom(Q)} = \{1, 2, 3\}$

P' is more-partially-correct than P
by virtue of having a smaller domain
$dom(R \cap P') \cup \overline{dom(P')} = \{0, 1, 2, 3\}$
$dom(R \cap P) \cup \overline{dom(P)} = \{1, 2, 3\}$

Figure 4.2 Relative partial correctness.

a larger set of initial states; see Figure 4.1. A program P' is more-partially-correct than a program P with respect to R if and only if it has a larger competence domain with respect to R, or it fails to terminate normally over a larger set of initial states (i.e., has a smaller domain); see Figure 4.2. From the standpoint of partial correctness, a program can evade accountability by failing to terminate.

4.2.2 Detector sets

The detector set of a program P on space S for some standard of correctness (partial or total) with respect to specification R is the set of states that disprove the correctness of P with respect to R (i.e. execution of P on these states yields a final state s' that violates specification R).

Definition 15. *Given a program P on space S and a specification R on S:*

- *The* detector set for total correctness *of program P with respect to R is denoted by $\Theta_T(R, P)$ and defined by:*

$$\Theta_T(R, P) = dom(R) \setminus dom(R \cap P).$$

- *The* detector set for partial correctness *of program P with respect to R is denoted by $\Theta_P(R, P)$ and defined by:*

$$\Theta_P(R, P) = (dom(R) \cap dom(P)) \setminus dom(R \cap P).$$

When we want to refer to a detector set without specifying a particular standard of correctness (partial, total), we simply say *detector set*, and we use the notation $\Theta(R, P)$. This definition generalizes the concept of *detector* set introduced in [53] by taking into consideration the definition of correctness and the specification against which correctness is tested. Given that detector sets are intended to expose incorrectness, they are empty whenever there is no incorrectness to expose; this is formulated in the following proposition.

Proposition 7. *Given a specification R on space S and a program P on S. Program P is totally correct with respect to specification R if and only if the detector set for total correctness of R and P is empty. Program P is partially correct with respect to specification R if and only if the detector set for partial correctness of R and P is empty.*

4.2.3 Properties

Since total correctness logically implies partial correctness, any test that disproves partial correctness necessarily disproves total correctness. Whence the following proposition.

Proposition 8. *The detector set for partial correctness of a program P with respect to specification R is a subset of the detector set for total correctness of P with respect to R.*

Proof. This stems readily from the observation that $\Theta_P(R, P) = \Theta_T(R, P) \cap dom(P)$. Ω**qed!**

Table 4.1 Definitions of Correctness

	Partial Correctness	Total Correctness
Absolute Correctness	$\Theta_P(R, P) = \emptyset$	$\Theta_T(R, P) = \emptyset$
Relative Correctness	$\Theta_P(R, P') \subseteq \Theta_P(R, P)$	$\Theta_T(R, P') \subseteq \Theta_T(R, P)$

In addition to the standard of correctness, detector sets depend on the program under and the specification against which the program is being tested. The following two propositions highlight how detector sets vary according to the relative correctness of the program, and the refinement of the specification.

Proposition 9. *Due to [38] Given a specification R on space S and two programs P and P' on S. P' is more-totally-correct than P with respect to specification R if and only if:*

$$\Theta_T(R, P') \subseteq \Theta_T(R, P).$$

If P' is more-partially-correct than P with respect to specification R then:

$$\Theta_P(R, P') \subseteq \Theta_P(R, P).$$

The intuitive interpretation of proposition 9 is straightforward: If P' is more correct (totally or partially) than P, then any test that reveals the failure of P' reveals necessarily the the failure of P (if the more-correct program fails, then so does the less-correct program).

Table 4.1 summarizes the results of Propositions 7 and 9 (though for relative partial correctness, we have proven necessity but not sufficiency of the provided condition).

Whereas Proposition 9 stipulates how the detector set of a program P with respect to R varies according to the program P, we now ponder the question of how it varies according to the specification R. We consider a program P on space S and two specifications R and R' on S such that R' refines R; R' refines R means that R' represents a stronger requirement than R; any test that disproves the correctness of P with respect to the less-refined (weaker) specification R disproves (a fortiori) the correctness of P against the (stronger) more-refined specification R'. Whence the the following proposition.

Proposition 10. *Given a program P on space S and two specifications R and R' on S. If R' refines R then the detector set of P with respect to R is a subset of the detector set of P with respect to R'.*

The proof of this proposition is due to [20].

4.3 Semantic Coverage

4.3.1 Definition

In this section, we propose a formula that measures the effectiveness of a test suite T for a program P to be tested for total and partial correctness with respect to specification R. As a first step, we ponder the following question: What is an ideal test suite? Then, we define the semantic coverage of an arbitrary (not necessarily ideal) test suite to reflect the extent to which the test suite comes close to the ideal case.

Given a program P on space S and a specification R on S, an ideal test suite T is one that is a superset of the detector set of P with respect to R for the selected correctness standard, since such a test suite exposes all the failures of P with respect to R. Let $\Theta(R, P)$ be the detector set of program P with respect to R; what precludes T from being a superset of $\Theta(R, P)$ is the set of elements of $\Theta(R, P)$ that are outside T, i.e.

$\Theta(R,P) \cap \overline{T}$.

The smaller this set, the better the test suite T; since we want a quantity that increases with the effectiveness of T rather than decreases, we take the complement of this quantity, whence the following definition.

Definition 16. *The semantic coverage of test suite T for the total correctness of program P with respect to specification R on space S is denoted by $\Gamma^{TOT}_{[R,P]}(T)$ and defined by:*

$$\Gamma^{TOT}_{[R,P]}(T) = T \cup \overline{\Theta_T(R,P)}.$$

The semantic coverage of test suite T for the partial correctness of program P with respect to specification R on space S is denoted by $\Gamma^{PAR}_{[R,P]}(T)$ and defined by:

$$\Gamma^{PAR}_{[R,P]}(T) = T \cup \overline{\Theta_P(R,P)}.$$

If we want to talk about semantic coverage without specifying the standard of correctness, we use the notation $\Gamma_{[R,P]}(T)$ defined by:

$$\Gamma_{[R,P]}(T) = T \cup \overline{\Theta(R,P)}.$$

4.3.2 Analytical validation

In this section, we revisit the requirements put forth in section 4.1.2 and prove that the formula of semantic coverage proposed above does satisfy all these requirements. For the most part, the propositions presented in this section stem immediately from the properties of detector sets discussed in the Subsection 4.2.3.

Proposition 11. *Due to [20]. Given a program P on space S, a specification R on S, and test suite T (subset of S), the semantic coverage of T for partial correctness of P with respect to R is greater than or equal to the semantic coverage for total correctness of P with respect to R.*

Proposition 12. *Due to [20]. Given a specification R on space S and two programs P and P' on S, and a subset T of S. If P' is more-totally-correct than P with respect to R then:*

$$\Gamma^{TOT}_{[R,P']}(T) \supseteq \Gamma^{TOT}_{[R,P]}(T).$$

Proposition 13. *Due to [20]. Given a specification R on space S and two programs P and P' on S, and a subset T of S. If P' is more-partially-correct than P with respect to R then:*

$$\Gamma^{PAR}_{[R,P']}(T) \supseteq \Gamma^{PAR}_{[R,P]}(T).$$

Proposition 14. *Due to [20]. Given a program P on space S and two specifications R and R' on S, and a subset T of S. If R' refines R then:*

$$\Gamma^{TOT}_{[R',P]}(T) \subseteq \Gamma^{TOT}_{[R,P]}(T).$$

Proposition 15. *Due to [20]. Given a program P on space S and two specifications R and R' on S, and a subset T of S. If R' refines R then:*

$$\Gamma^{PAR}_{[R',P]}(T) \subseteq \Gamma^{PAR}_{[R,P]}(T).$$

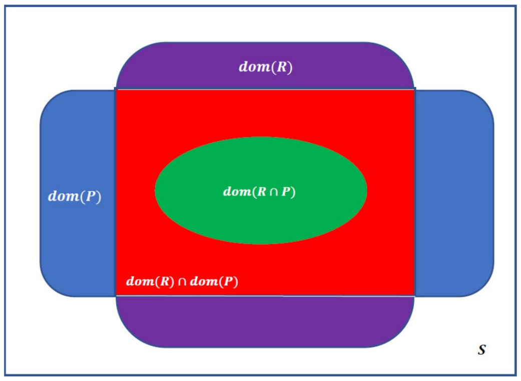


Figure 4.3 Detector sets for partial correctness.

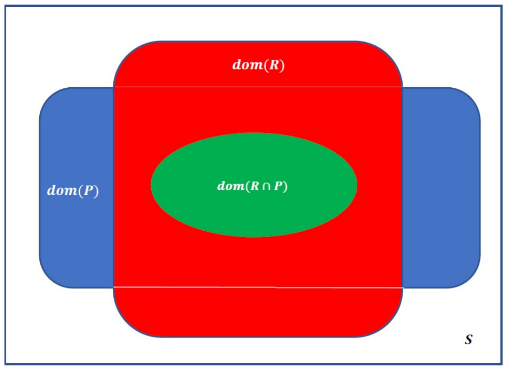


Figure 4.4 Detector sets for total correctness.

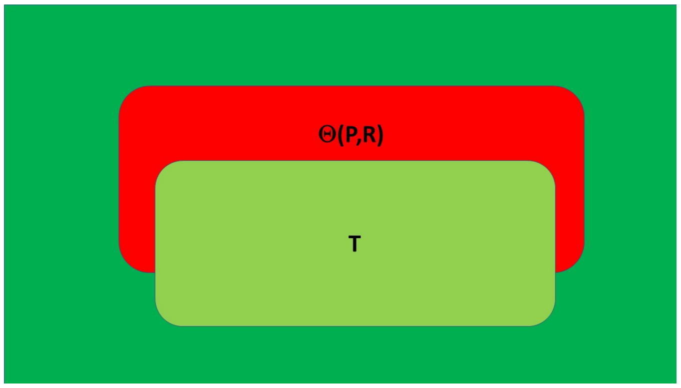


Figure 4.5 Semantic coverage of test T for program P with respect to R (shades of green).

4.4 Illustration

In this section, we report on an experiment in which we evaluate the semantic coverage of a set of test suites, and compare our findings to an existing measure of coverage, namely *mutation coverage*. To this effect, we consider the Java benchmark program of *jTerminal*[1], an open-source software product routinely used in mutation testing experiments [30]. We apply the mutant generation tool *LittleDarwin* in conjunction with a test generation and deployment class that includes 35 test cases [30]; we augment the benchmark test suite with two additional tests, intended specifically to *trip* the base program *jTerminal*, by causing it to diverge (i.e., fail to terminate normally). The purpose of these tests is to enable us to distinguish between partial correctness and total correctness. We let T designate the augmented test suite codified in this test class. Execution of LittleDarwin on jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants:

```
m1, m2, m7, m8, m9, m10, m11, m12, m13,
m14, m15, m16, m17, m18, m19, m21, m22,
m23, m24, m25, m26, m27, m28, m44, m45,
m46, m48, m49, m50, m51, m52, m53, m54,
m55, m56, m57, m58, m59, m60, m61, m62,
m63, m83, m88, m89, m90, m92, m93.
```

Some of these mutants are equivalent to each other, i.e., they produce the same output for each of the 37 elements of T; when we partition these 48 mutants by equivalence, we find 31 equivalence classes, and we select a mutant from each class:

$\mu =$

[1]Available online at `http://www.grahamedgecombe.com /projects /jterminal`

`m1, m2, m7, m11, m13, m15, m19, m21, m22,`

`m23, m24, m25, m27, m28, m44, m45, m46, m48,`

`m49, m50, m51, m52, m53, m55, m56, m57, m60,`

`m63, m92, m93`.

Orthogonally, we consider set T, and we select twenty subsets thereof, derived as follows:

- *T1, T2, T3, T4, T5*: Five distinct test suites obtained from T by removing 15 elements at random.
- *T6, T7, T8, T9, T10*: Five distinct test suites obtained from T by removing 10 elements at random.
- *T11, T12, T13, T14, T15*: Five distinct test suites obtained from T by removing 5 elements at random.
- *T16, T17, T18, T19, T20*: Five distinct test suites obtained from T by removing one element at random.

To have a baseline against which we compare our measures of semantic coverage, we rank the test suites T1... T20 by mutation coverage [31]; but we do not equate mutation coverage with the ratio of killed mutants over generated mutants; rather, we rank test suites by comparing, in terms of set inclusion, the sets of mutants they kill. The result is shown in Figure 4.6.

To compute the semantic coverage of these test suites, we need to define specifications against which the test is carried out; for the sake of this experiment, we choose mutants M25 and M50 as sample specifications, and we compute the semantic coverage of test suites T1... T20 for total correctness and partial correctness with respect to M25 and M50. For each specification and standard of correctness, we compute the semantic coverage of each of the twenty test suites, which we compare by inclusion. The result is shown in Figures 4.7, 4.8, 4.9 and 4.10 for, respectively, the partial correctness with respect to M25 and M50 then the total correctness with respect to M25 and M50.

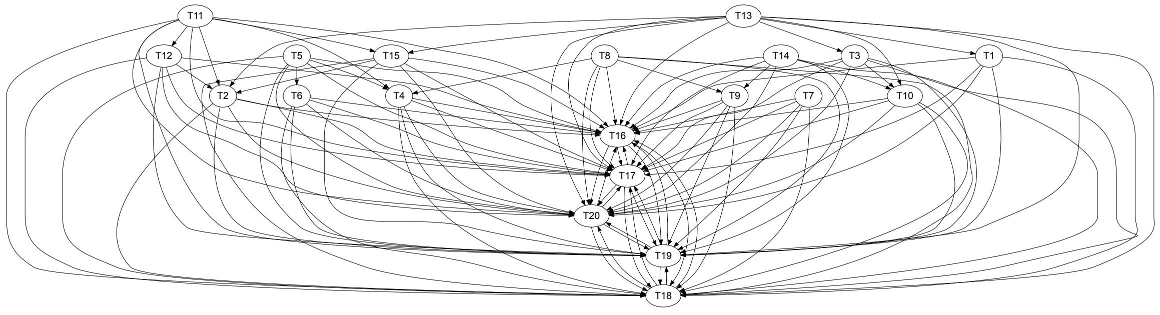


Figure 4.6 Ordering test suites T_i by mutation coverage (inclusion relations of killed mutant sets).

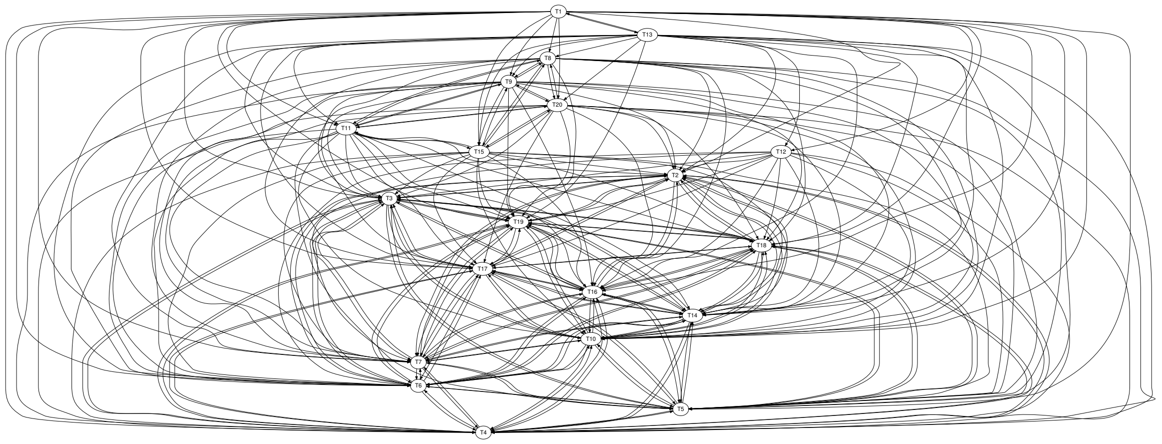


Figure 4.7 Ordering test suites by inclusion relations of $\Gamma^{PAR}_{[M25,P]}(T_i)$.

While we do not expect that the graph of mutation coverage (Figure 4.6) and the graphs of semantic coverage be the same, since the former is intrinsic to the program, whereas the latter also depends on the specification and the standard of correctness, we are interested in considering the similarity between the mutation coverage graph and the graphs of semantic coverage. To assess the similarity between graphs, we use the imperfect but simple metric of the ratio between the number of common arcs over the total number of arcs. The results are shown in Table 4.2. Interestingly, the graphs of semantic coverage have greater similarity between themselves than they have with mutation coverage.

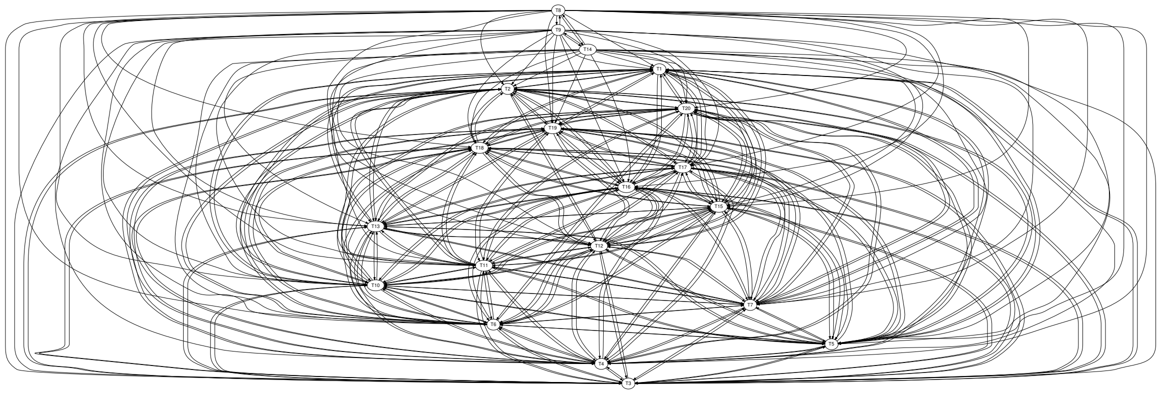


Figure 4.8 Ordering test suites by inclusion relations of $\Gamma^{PAR}_{[M50,P]}(T_i)$.

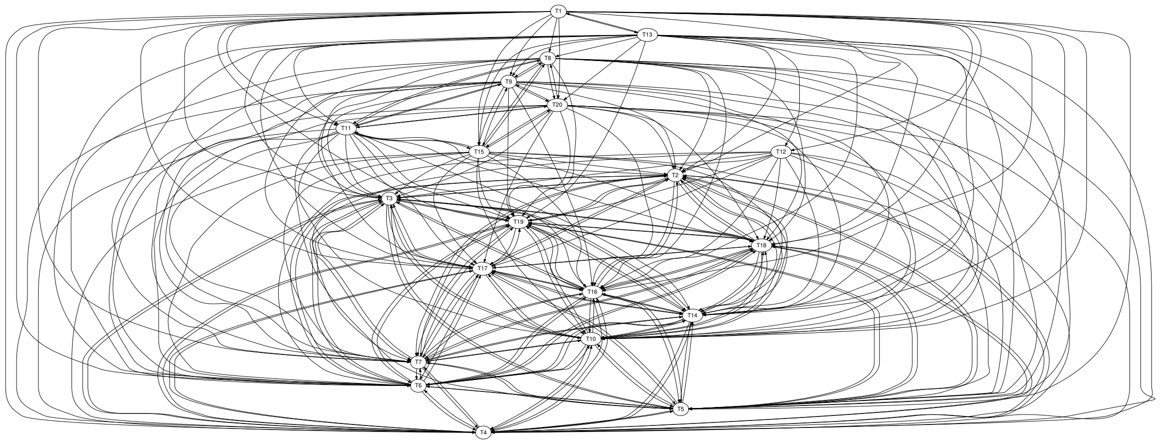


Figure 4.9 Ordering test suites by inclusion relations of $\Gamma^{TOT}_{[M25,P]}(T_i)$.

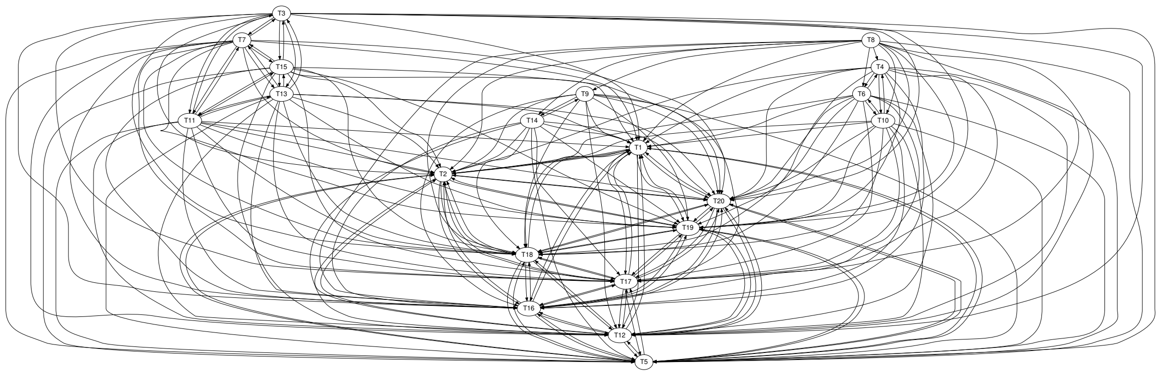


Figure 4.10 Ordering test suites by inclusion relations of $\Gamma^{TOT}_{[M50,P]}(T_i)$.

Table 4.2 Graph Similarity of Semantic Coverage and Mutation Coverage

Graph Similarity	Mutation	$\Gamma^{PAR}_{[M25,P]}(T)$	$\Gamma^{PAR}_{[M50,P]}(T)$	$\Gamma^{TOT}_{[M25,P]}(T)$	$\Gamma^{TOT}_{[M50,P]}(T)$
Mutation	1	0.34	0.35	0.34	0.5
$\Gamma^{PAR}_{[M25,P]}(T)$	0.34	1.0	0.66	1.0	0.46
$\Gamma^{PAR}_{[M50,P]}(T)$	0.35	0.66	1.0	0.66	0.62
$\Gamma^{TOT}_{[M25,P]}(T)$	0.34	1.0	0.66	1.0	0.46
$\Gamma^{TOT}_{[M50,P]}(T)$	0.50	0.46	0.62	0.46	1.0

4.5 Conclusion

4.5.1 Summary and assessment

In this chapter, we discuss two concepts in software testing: the *detector set* of a program P with respect to a specification R is the set of tests that disprove the correctness of P with respect to R. Using detector sets, we introduce the *semantic coverage* of a test suite T for (total or partial) correctness of a program P with respect to specification R as the union of T with the complement of the detector set of P with respect to R. In other words, the semantic coverage of a test suite T is the union of the test data on which P is tested (T) with the test data on which P does not have to be tested ($\overline{\Theta(R, P)}$).

on one hand, we find that the detector set of a program P with respect to a specification R grows larger when we transition from partial correctness to total correctness, when P grows less correct with respect to R, and when R grows more-refined. All of this makes sense when we consider that the purpose of a detector set is to disprove the correctness of P with respect to R (expose program failures).

On the other hand, we find that the semantic coverage of a test suite T for program P with respect to specification R grows larger when T grows larger, when we transition from total correctness to partial correctness, when P grows more-correct, and when R grows less-refined. All of this makes sense when we consider that semantic coverage reflects the effectiveness of a test suite to test a program for correctness against a specification. Consider that a tool is all the more effective than it is intrinsically more powerful (larger T) and the task on which it is deployed is easier (smaller detector set $\Theta(R, P)$).

For illustration, we compute the semantic coverage of twenty test suites of a benchmark program with respect to two sample specifications for total and partial correctness; and we compare the way semantic coverage ranks tets suites with the way mutation coverage does.

We validate the proposed formula of semantic coverage analytically by showing that it is monotonic with respect to several attributes that reflect test suite effectiveness. We also validate it empirically by showing a sample benchmark example that there is much similarity between the ordering derived from semantic coverage with the ordering derived from mutation coverage.

4.5.2 Threats to validity

The empirical validation is clearly incomplete, as it is based on a single benchmark example, but it is intended as a complement to the analytical validation rather than a substitute thereof. The main difficulty of the proposed coverage metric is that it assumes the availability of a specification, and it is difficult to estimate in practice, but our purpose is to propose a measure of coverage that can be used for reasoning about test suites or for comparing test suites. Its applicability for such purposes is the subject of future research.

4.5.3 Related work

Coverage metrics of test suites have been the focus of much research over the years, and it is impossible to do justice to all the relevant work in this area [6, 54–59]; as a first approximation, it is possible to distinguish between code coverage, which focuses on measuring the extent to which a test suite exercises various features of the code and specification coverage, which focuses on measuring the extent to which a test suite exercises various clauses or use cases of the requirements specification. This can be tied to the orthogonal approaches to test data generation, using, respectively, structural criteria and functional criteria. Mutation coverage falls somehow outside of this dichotomy, in that it depends exclusively on the program, not its specification, and that it operates by applying mutation operators, wherever they are applicable, without regard to syntactic coverage; as such, it has often been used as a baseline for assessing the effectiveness of other coverage metrics [6, 47].

Our work differs from these research efforts in a number of ways: perhaps first and foremost, our coverage semantic measure is not a number but a set; as such, it is not totally ordered by numeric inequality, but partially ordered by set inclusion. Second, semantic coverage is not intrinsic to the program, but depends also on the correctness standard used in testing, and the specification with respect to which correctness is judged. Third, semantic coverage is focused on revealing failures rather than diagnosing faults on the grounds that failures are an objectively observable attribute, but faults are hypothesized causes of observed failures.

4.5.4 Research prospects

We are exploring means to use the definition of semantic coverage to derive a function that is independent of the specification, and reflects the diversity of the test suite. We are also considering expanding the empirical validation of our definition of semantic coverage by comparing it to (yet to be defined/ identified) objective measures of test suite effectiveness.

CHAPTER 5

MUTANTS SET EFFECTIVENESS

5.1 Mutant Set Minimization: An Optimization Problem

5.1.1 Motivation

Mutation testing is the art of generating syntactic mutants of a base program and is used primarily to assess the effectiveness of test suites: a test suite T is all the more effective that it can expose semantic differences between the base program and non-equivalent mutants (i.e. execution of M on T yields a different set of outcomes from execution of mutant P on T; we then say that T kills M). One of the main obstacles to the practical usefulness of mutation testing is cost: even small programs can give rise to a large number of mutants [10–13]. This has spurred the interest in minimizing mutant sets.

The problem of minimizing a set of mutants is essentially an optimization problem: Given a set $\mu = \{M_1, M_2, ...M_n\}$ of mutants of P, find a subset μ' of μ of minimal cardinality that has the same effectiveness as μ. This raises the question: how do we define the effectiveness of a mutant set?

5.1.2 Design principles

This chapter aims to propose a definition of mutant set effectiveness. In this section, we discuss the design principles that we adopt for this purpose.

- *The Purpose of a Mutant Set.* To the extent that the effectiveness of an artifact is a reflection of its fitness for a purpose, the first question we must address is: What is the purpose of a mutant set? We postulate that the purpose of a set of program mutants P is to vet test suites for P.
  - *Vetting a Test Suite.* We say that a mutant set μ *vets* a test suite T if and only if T kills every element of μ.

- *Measuring Effectiveness.* In light of the above premise, we resolve to measure the effectiveness of a set of mutants as a function of the effectiveness of the test suites that it vets.
  - *The Purpose of a Test Suite.* To quantify the effectiveness of a test suite, we raise the following question: What is the purpose of a test suite? We posit that the purpose of a test suite is to expose the failures of an incorrect program.
  - *The Effectiveness of a Test Suite.* We resolve to measure the effectiveness of a test suite by the extent to which it exposes the failures of an incorrect program (if the program is correct, then any test suite is maximally effective since there are no failures to expose).
- *Testing Against a Specification.* The determination of whether the behavior of a program for a given input is or is not correct is made with respect to a specification; hence, the quality of a test suite must be assessed with respect to a specification.
- *Standards of Correctness.* Two standards of correctness have traditionally been used to judge a program: partial correctness and total correctness [25–28]; this yields two different measures of test suite effectiveness.
- *Effectiveness as a Partial Ordering Relation.* It may be tempting to quantify the effectiveness of a mutant set by a number, but we resolve not to do so: First, because the effectiveness of a mutant set is a complex multi-dimensional attribute, reducing it to a single, dry number carries the risk of much loss of information. Second, measuring the effectiveness of mutant sets by a number creates an artificial total ordering when no such an ordering exists, in fact, it is easy to imagine two mutant sets that are not comparable in terms of effectiveness, yet numbers are always comparable.

5.2 Mutant Set Effectiveness

5.2.1 Measuring effectiveness

We resolve to quantify the effectiveness of a mutant set by considering the test suites that it vets, whence the following definition.

Definition 17. *Given a program P and a set of mutants of P, say*

$$\mu = \{M_1, M_2, M_3, ...M_N\},$$

the vetted set *of μ is the set of minimal test suites that kill all the mutants of μ (i.e. test suites that kill all the mutants of μ and such that no proper subset thereof does); we denote it by (μ).*

We quantify the effectiveness of a mutant set μ as an aggregate of the semantic coverage of all the test suites in (μ). Given that different elements of (μ) may have different levels of semantic coverage, we aggregate them by computing a lower bound and an upper bound thereof; whence the following definition.

Definition 18. *Given a specification R, a program P and a set μ of mutants of P, we define two quantities to measure the effectiveness of μ:*

- Assured Effectiveness. *This is denoted by $\sigma^A_{\{R,P\}}(\mu)$ and defined as:* $\sigma^A_{\{R,P\}}(\mu) = \bigcap_{T \in (\mu)} \Gamma_{\{R,P\}}(T)$.
- Potential Effectiveness. *This is denoted by $\sigma^P_{\{R,P\}}(\mu)$ and defined as:* $\sigma^P_{\{R,P\}}(\mu) = \bigcup_{T \in (\mu)} \Gamma_{\{R,P\}}(T)$.

Given that there are two versions of semantic coverage (for partial and total correctness), there are actually two versions of assured effectiveness, and two versions of potential effectiveness. We are unable to aggregate the semantic coverage of all the vetted test suites into a single metric, but it is not uncommon for the attribute of a feature to be quantified by two metrics:

- One metric that we use when we are interested in minimizing risk; this is what assured effectiveness represents.
- One metric that we use when we are interested in maximizing potential benefit; this is what potential effectiveness represents.

The following Proposition stems readily from the definition, and is valid for both versions of semantic coverage.

Proposition 16. *Given a program P and a set μ of mutants of P, and given a test suite T in (μ), the following inequations hold:*

$$\sigma^A_{\{R,P\}}(\mu) \subseteq \Gamma_{\{R,P\}}(T) \subseteq \sigma^P_{\{R,P\}}(\mu).$$

5.2.2 Preserving effectiveness

In this section, we consider whether minimization algorithms that reduce the size of mutant sets using the criterion of subsumption [10, 15, 16, 29–33, 40] actually, preserve mutant set effectiveness as defined in this chapter. We consider the original definition of subsumption [15]: Given two mutants M and M' of program P, we say that M subsumes M' if and only if:

P1 There exists some test t such that M and P compute different outcomes (t kills M).

P2 For every possible test t for P, if M computes a different outcome from P on t, then so does M'.

We resolve to refine this definition by considering what is meant by *the outcome* of a program's execution, under what condition do we consider that two execution outcomes are comparable, and under what condition do we consider that two comparable outcomes are identical. When a program P is executed on some initial state s, it may terminate after a finite number of steps in a final state s'; we then say that P *converges* on s. But it may also fail to terminate normally due a wide range of possibilities, such as an infinite loop, a division by zero, an array reference out of bounds, a stack overflow, an arithmetic overflow, etc; we then say that P *diverges* on s. Whereas the original definition of Kurtz et al. [15] appears to assume that programs and mutants converge for all initial states, this is not necessarily the case in practice; in fact many mutation operators are prone to cause mutants to diverge, even when the base program converges. The following definition, due to [37], introduces the concept of *differentiator set* as the set of initial states that cause two programs to yield different outcomes.

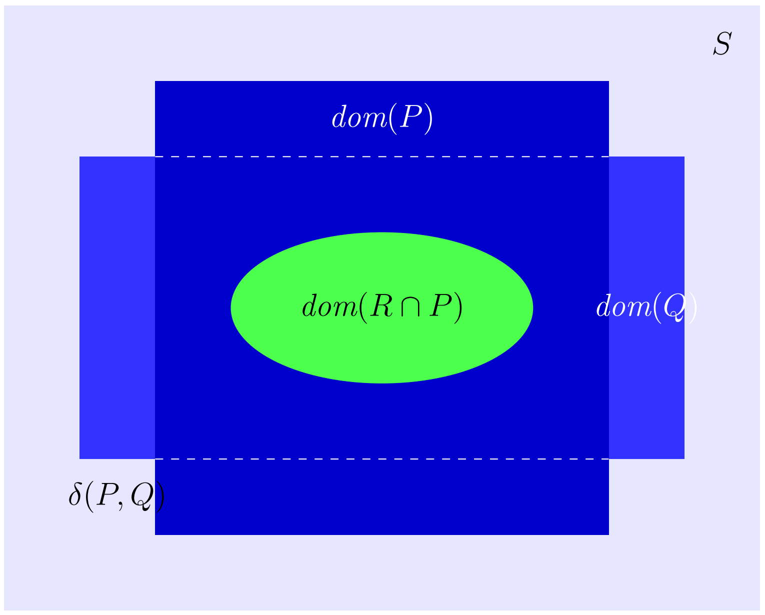


Figure 5.1 Differentiator set of P and Q.

Definition 19. *Given two programs P and Q on space S, the* differentiator set *of P and Q is the set denoted by $\delta(P,Q)$ and defined as:*

$$\delta(P,Q) = (dom(P) \cup dom(Q)) \cap \overline{dom(P \cap Q)}.$$

The differentiator set of P and Q is the set of initial states s such that either P and Q both converge on s and produce different final states or only one of them converges and the other diverges. See Figure 5.1; the differentiator set of P and Q is the area colored in (both shades of) blue. Using differentiator sets, we now introduce the property of mutant subsumption.

Definition 20. *Given a program P on space S and two mutants M and M' of P, we say that M* subsumes *M' if and only if:*

$$\emptyset \subset \delta(P,M) \subseteq \delta(P,M').$$

This definition coincides with the original definition of Kurtz et al. [15] whenever P, M and M' converge for all initial states, but makes provisions for the case when the base program and its mutant diverge for some initial states. With this definition of subsumption, we have the following Proposition.

Proposition 17. *Let μ be a set of mutants of program P and let μ' be $\mu' = \mu \cup \{M'\}$ for some mutant M' that is subsumed by some mutant M of μ. Then μ and μ' have the same potential effectiveness.*

The proof of this proposition is given in [21] By virtue of this Proposition, removing a subsumed mutant preserves the potential effectiveness of a mutant set; this proves that our definition of mutant set effectiveness meets requirement RQ4 (Subsection 4.2.1).

5.3 Assessment and Validation

For the sake of illustration, we consider a benchmark Java class that emulates DEC's *VT100* terminal[1]; this is an open-source product that is routinely used in mutation testing experiments [30]. This class comes with a test suite that includes 35 tests, to which we add two tests that are intended to cause the program to diverge; we may refer to the base program (jTerminal) as jT and the to the test suite as T. One may ask why we would test the program outside its domain; we argue that the determination of test data is driven by the domain of the specification (R), not the domain of the program (jT). To generate mutants of this program, we use *LittleDarwin*, a mutant generator for Java code, due to Parsai et al. [10]. Execution of LittleDarwin on jTerminal yields 94 mutants, numbered m1 to m94; the test of these mutants against the original using the selected test suite kills 48 mutants; for the sake of documentation, we list them below:

```
m1, m2, m7, m8, m9, m10, m11, m12, m13, m14, m15, m16, m17, m18,
m19, m21, m22, m23, m24, m25, m26, m27, m28, m44, m45, m46, m48,
```

[1]`http://www.grahamedgecombe.com /projects /jterminal`

`m49, m50, m51, m52, m53, m54, m55, m56, m57, m58, m59, m60, m61,`

`m62, m63, m83, m88, m89, m90, m92, m93.`

The remaining 46 mutants are semantically equivalent to the pre-restriction of jTerminal to T. We partition these 48 mutants into equivalence classes modulo semantic equivalence; we find that these 48 mutants are partitioned into 31 equivalence classes, and we select a member from each class; we let μ be the set of selected mutants: $\mu =$

`m1, m2, m7, m11, m13, m15, m19, m21, m22, m23, m24, m25, m27, m28, m44,`

`m45, m46, m48, m49, m50, m51, m52, m53, m55, m56, m57, m60, m63, m92, m93.`

We compute the differentiator set of each mutant with respect to jT per definition 19 then we order these mutants by subsumption, per definition 20. We select the maximal nodes of this graph, and we let μ_1 be the set of these mutants. $\mu_1 = \{$`m1, m19, m23, m24, m25, m27, m44, m45, m48, m51, m60`$\}$.

We further introduce two more mutant sets, as follows:

$\mu_2 = \{$`m1, m19, m24, m25, m27, m44, m45, m48, m51, m60`$\}$.

$\mu_1 = \{$`m1, m19, m23, m24, m25, m27, m28, m44, m45, m46, m48, m51, m52, m60`$\}$.

Mutant set μ_2 is obtained from μ_1 by removing mutant `m23`, which is the mutant that is killed by most of the elements of T; mutant set μ_3 is derived from μ_1 by adding three random mutants, `m52, m28, m46`, with the expectation that (per Proposition 17) they would not enhance the potential effectiveness of the set.

We further introduce three specifications, $R1$, $R2$ and $R3$, which are ordered by refinement, so that

$R1 \sqsubseteq R2 \sqsubseteq R3$,

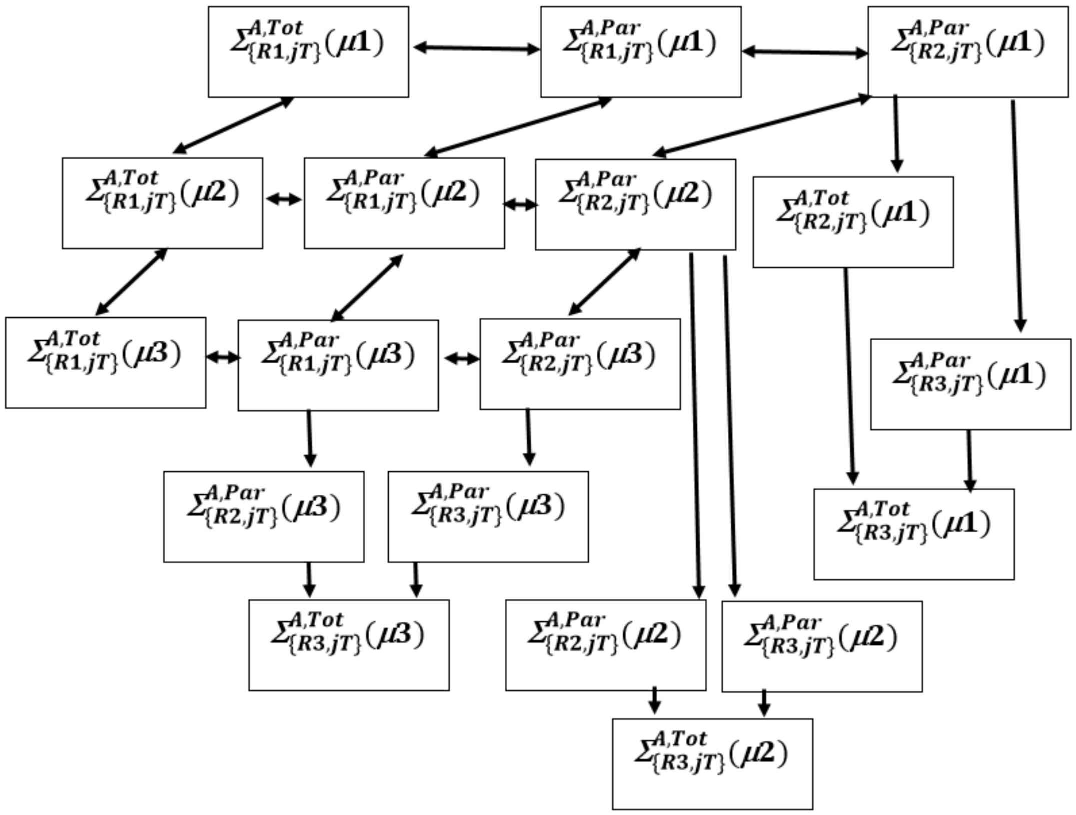
which means that $R1$ captures weaker requirements than $R2$, which captures weaker requirements than $R3$. Also, for the sake of illustration, we choose $R1$ in such a way as to make `jT` partially correct and totally correct with respect to $R1$; we choose $R2$ in such a way as to make `jT` partially correct but not totally correct with respect to $R2$; and we choose $R3$ in such a way as to make `jT` neither partially correct nor totally correct with respect to $R3$.

We compute the assured effectiveness and the potential effectiveness of each mutant set ($\mu1$, $\mu2$, $\mu3$) for partial correctness and total correctness with respect to specifications $R1$, $R2$, and $R3$. Then, we compare these measures of effectiveness for inclusion, whereby larger measures reflect greater effectiveness. For the sake of presentation, we separate the results into two graphs: Figure 5.2 shows the ordering of assured effectiveness, and Figure 5.3 shows the ordering of potential effectiveness; for completeness, imagine that there is an arc from each node in the potential effectiveness graph onto the corresponding node in the assured effectiveness graph.

These graphs bear out the claims that we are making about our measure of effectiveness:

- For each measure of effectiveness, each mutant set and each specification, the effectiveness of the mutant set against partial correctness is greater than its effectiveness against total correctness, as partial correctness, is a weaker property (hence, easier to test).
- For each measure of effectiveness, each mutant set and each standard of correctness (partial/ total), the effectiveness of the mutant set against a less-refined specification is greater than its effectiveness against a more refined specification, as less refined specifications are easier to test against.
- Considering the graph of potential effectiveness, we notice that when we removed mutant `m23` from mutant set $\mu1$ to obtain mutant set $\mu2$, the potential effectiveness was degraded, which is consistent with the property that $\mu1$ is minimal: any reduction thereof degrades its effectiveness. On the other hand, adding mutants `m52, m28, m46` to $\mu1$ to obtain $\mu3$ preserves its effectiveness, since the added mutants are redundant with $\mu1$.

  Interestingly, removing `m23` from $\mu1$ did not affect its assured effectiveness, as we found that $\mu1$ and $\mu2$ have the same assured effectiveness. This, along with Proposition 17,



$\Sigma^{A,Tot}_{\{R1,jT\}}(\mu 1)$
$\Sigma^{A,Par}_{\{R1,jT\}}(\mu 1)$
$\Sigma^{A,Par}_{\{R2,jT\}}(\mu 1)$
$\Sigma^{A,Tot}_{\{R1,jT\}}(\mu 2)$
$\Sigma^{A,Par}_{\{R1,jT\}}(\mu 2)$
$\Sigma^{A,Par}_{\{R2,jT\}}(\mu 2)$
$\Sigma^{A,Tot}_{\{R2,jT\}}(\mu 1)$
$\Sigma^{A,Tot}_{\{R1,jT\}}(\mu 3)$
$\Sigma^{A,Par}_{\{R1,jT\}}(\mu 3)$
$\Sigma^{A,Par}_{\{R2,jT\}}(\mu 3)$
$\Sigma^{A,Par}_{\{R3,jT\}}(\mu 1)$
$\Sigma^{A,Par}_{\{R2,jT\}}(\mu 3)$
$\Sigma^{A,Par}_{\{R3,jT\}}(\mu 3)$
$\Sigma^{A,Tot}_{\{R3,jT\}}(\mu 1)$
$\Sigma^{A,Tot}_{\{R3,jT\}}(\mu 3)$
$\Sigma^{A,Par}_{\{R2,jT\}}(\mu 2)$
$\Sigma^{A,Par}_{\{R3,jT\}}(\mu 2)$
$\Sigma^{A,Tot}_{\{R3,jT\}}(\mu 2)$

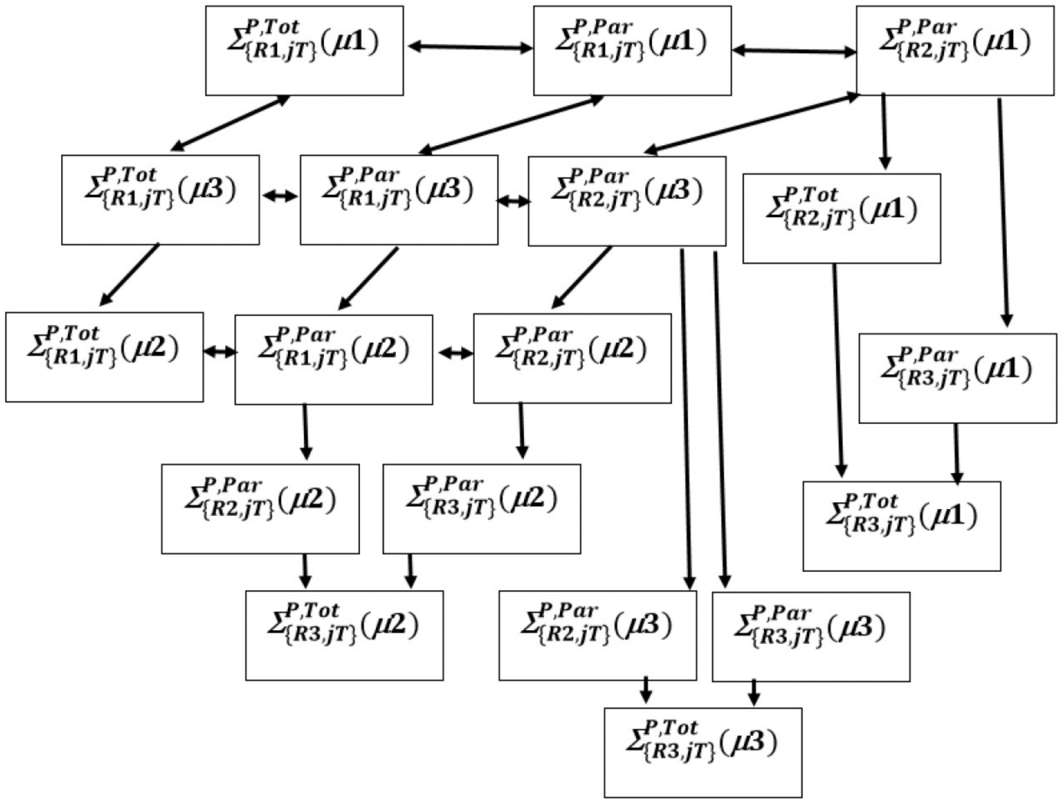
Figure 5.2 Assured effectiveness of $\mu 1$, $\mu 2$, $\mu 3$ for partial and total correctness with respect to $R1$, $R2$, $R3$.

seems to suggest that potential effectiveness is a better measure of mutant set quality than assured effectiveness, though we argue that both are important.

5.4 Conclusion: Summary, Critique, and Prospects

5.4.1 Summary

Much research has focused on minimizing a set of mutants; this is essentially an optimization problem, that ought to be characterized by an objective function and a constraint under which the objective function is minimized. Of course, the assumption is that a mutant set must be minimized without degrading its effectiveness, but this leaves open the question of how we define the effectiveness of a mutant set.



$\Sigma^{P,Tot}_{\{R1,jT\}}(\mu 1)$
$\Sigma^{P,Par}_{\{R1,jT\}}(\mu 1)$
$\Sigma^{P,Par}_{\{R2,jT\}}(\mu 1)$
$\Sigma^{P,Tot}_{\{R1,jT\}}(\mu 3)$
$\Sigma^{P,Par}_{\{R1,jT\}}(\mu 3)$
$\Sigma^{P,Par}_{\{R2,jT\}}(\mu 3)$
$\Sigma^{P,Tot}_{\{R2,jT\}}(\mu 1)$
$\Sigma^{P,Tot}_{\{R1,jT\}}(\mu 2)$
$\Sigma^{P,Par}_{\{R1,jT\}}(\mu 2)$
$\Sigma^{P,Par}_{\{R2,jT\}}(\mu 2)$
$\Sigma^{P,Par}_{\{R3,jT\}}(\mu 1)$
$\Sigma^{P,Par}_{\{R2,jT\}}(\mu 2)$
$\Sigma^{P,Par}_{\{R3,jT\}}(\mu 2)$
$\Sigma^{P,Tot}_{\{R3,jT\}}(\mu 1)$
$\Sigma^{P,Tot}_{\{R3,jT\}}(\mu 2)$
$\Sigma^{P,Par}_{\{R2,jT\}}(\mu 3)$
$\Sigma^{P,Par}_{\{R3,jT\}}(\mu 3)$
$\Sigma^{P,Tot}_{\{R3,jT\}}(\mu 3)$

Figure 5.3 Potential effectiveness of $\mu 1$, $\mu 2$, $\mu 3$ for partial and total correctness with respect to $R1$, $R2$, $R3$.

In this chapter, we propose a measure of the effectiveness of a mutant set, and show that our measure satisfies some attributes that one would expect from a sound definition. Our formula of the mutant set effectiveness is not intrinsic to the mutant set, but also depends on the correctness property that we want the mutant set to help us test: as such, it is monotonic with respect to the standard of correctness that we are trying to prove (it is easier to test for partial than for total correctness) and with respect to the specification we are trying to prove correctness against (it is easier to test a program against a weaker specification than against a stronger/more refined specification). Also, we show (analytically) that our measure of effectiveness is preserved when a redundant mutant is removed, and (empirically) that our measure of effectiveness is degraded when a non-redundant mutant is removed. As a stepping stone toward defining mutant set effectiveness, we have also introduced a measure of test suite effectiveness (semantic coverage), which focuses on a test suite's ability to expose failures [46] (rather than faults); reasoning about observable failures rather than hypothesized faults obviate the need to speculate about what fault(s) may cause an observed failure, and spares us the uncertainty engendered by this speculation.

5.4.2 Related work

In [60], Gopinath et al. propose to quantify the effectiveness of a set of mutants by means of two metrics, namely a measure of variance (reflecting the range of functional diversity that the mutants represent) and a measure of thoroughness or precision (reflecting the mutant's ability to detect subtle variability stemming from stubborn mutations); these two measures appear to capture, intuitively, two complementary aspects of mutant set quality, namely breadth and density. In [61], Feldt et al. introduce a related concept that captures, in intuitively appealing terms, an important attribute of test suites: the concept of *test set diameter*; this concept is defined by means of Kolmogorov's conditional complexity function, which quantifies the amount of similarity (or diversity) between two strings by the length in bits of the shortest program that transforms the longer string into the shorter string. In [53] Shin et al. critique the traditional metric of mutation score and propose a new metric that takes into account, not only the set of mutants that are killed by a given test suite, but also their diversity, as reflected by the disjointness of their differentiator sets. In [62], Kaufman et al. measure the usefulness of a mutant by the probability that this mutant advances the completeness of a test suite towards a tester-defined goal. This metric, called *TCAP* (Test Completeness Advancement Probability) is used to characterize redundant mutants, dominator mutants, subsumed mutants, etc, and ultimately to prioritize mutants for inclusion in a mutant set. In [63] Papadakis et al. present a survey of mutant quality metrics proposed up to 2018.

Our work differs from all these in some fundamental attributes:

- First, while we understand the interest of measuring mutant diversity as an intrinsic attribute of the mutant set, our metric of mutant set effectiveness is also dependent on the goal of testing, as defined by the correctness attribute we are testing for (partial correctness, total correctness) and the specification against which correctness is tested. The same set of mutants may be adequate to test program P for a standard of correctness and a specification, yet totally inadequate for another combination of standards of correctness/ specification.
- Second, we do not quantify the effectiveness of a mutant set by numbers (nor a pair of numbers as in [60]) but rather by sets (two sets: assured effectiveness and potential effectiveness); indeed, we recognize that something as rich, complex and multi-dimensional as the effectiveness of a mutant set cannot be reduced to a single dry number without much loss of information; also reducing it to a single number creates an artificial total ordering to represent what is potentially a (very) partial ordering (see Figures 5.2 and 5.3).
- We derive the quality of a mutant set, not from the attributes of the mutants that are members of the set but rather from the test suites that are vetted by the set.

5.4.3 Threats to validity and prospects

The biggest threat to the validity of our measures of effectiveness is the difficulty of estimating them in practice because they require that we derive the set of minimal test suites that are vetted by the mutant set. Another weakness that needs to be addressed is the dependence of effectiveness with respect to the relative correctness of a program [48]: it ought to be easier to test a more-correct program than a less-correct program; hence, effectiveness ought to be higher for the former than the latter. These matters are under investigation.

CHAPTER 6

MUTATION COVERAGE IS NOT STRONGLY CORRELATED WITH MUTATION COVERAGE

6.1 On the Effectiveness of a Test Suite

6.1.1 Measuring test suite effectiveness

The effectiveness of an artifact can only be defined with with respect to the purpose of the artifact and must reflect the fitness of the artifact to fulfill its declared purpose. The purpose of a test suite is to expose the failures of an incorrect program (if it fails on the test suite) or, equivalently, to prove the correctness of a correct program (if it succeeds on the test suite); the effectiveness of a test suite can be measured by the extent to which it can fulfill this purpose. We further argue that the main purpose of quantifying the attribute of an artifact is primarily to be able to compare artifacts with respect to that attribute: knowing, for instance, the statement coverage of some test suite T1 is useful mostly because it enables us to compare it to that of another test suite, say T2, to determine superiority.

A number of quantitative metrics have been proposed in the past to measure the effectiveness of a test suite; many involve measures of syntactic coverage (statement coverage, branch coverage, condition coverage, path coverage, etc.), since to expose the failures of a program, we need to exercise the faulty source code that causes failures. But syntactic coverage is clearly an imperfect metric, since exercising all the syntactic features of a program is neither necessary nor sufficient to detect all the faults of the program: it is not sufficient because the same fault may be sensitized for some tests but not for others, and it is not necessary because, strictly speaking, only faulty features need to be exercised (of course, in practice, we do not know which features are faulty).

6.1.2 Mutation coverage: a reference by default

To remedy the shortcomings of syntactic coverage metrics, researchers have often resorted to *mutation coverage* as a metric for test suite effectiveness. To the extent that mutations are faithful proxies for faults [7,35,36], the ability of a test suite to kill mutants can be equated to its ability to detect faults in a faulty program. In [64] Parsai and DeMeyer investigate the use of syntactic coverage metrics in the industry, and highlight the shortcomings of such metrics by comparison with mutation coverage.

In [5], Andrews et al. use mutation analysis to assess the cost and effectiveness of four common control flow and data flow criteria: block, decision, C-use, and P-use; they use the experimental data to revisit important software testing questions such as the relationships between fault detection, test suite size, control flow, and data flow.

In [65], Frankl et al. conduct experiments to compare the effectiveness of the mutation testing adequacy criterion to that of the all-uses criterion at various levels of coverage, for randomly generated test suites, they find that at the highest coverage levels, mutation is more effective than all-uses in most cases, but they also find that mutation is more expensive to implement.

In [66], Aaltonen et al. critique an automated testing tool that is used in a classroom environment to give students feedback on the effectiveness of their test data and find that, in practice the the tool may give students credit for high coverage that does not necessarily translate to high effectiveness; initial experimental results indicate that mutation coverage is a better indicator of test suite quality than code coverage.

In [47], Inozemtseva and Holmes conduct an empirical study in which they evaluate the relationship between test suite size, syntactic coverage and test suite effectiveness for large Java programs; the study involves 31,000 test suites and five systems, totaling 724,000 lines of code; the coverage metrics that were studied include statement coverage, decision coverage and condition coverage, and the effectiveness of these criteria was assessed with respect to mutation coverage.

In [67], Li et al. perform an experiment to compare four unit-level software testing criteria: mutation testing, prime path coverage, edge pair coverage, and all-uses. The criteria were assessed on the basis of two parameters, namely, the number of seeded faults that are uncovered and the number of tests needed to satisfy the criteria. Li et al. find that mutation testing exposes more faults and requires fewer tests than all the other criteria.

In [68], Tengeri et al. report on an experiment they conducted on four open-source systems' test suites to compare them for code coverage, test suite reducibility (the extent to which test adequacy is degraded when a test suite is reduced). Their experiment shows that in some situations, code coverage provides sufficient indication of fault density, but mutation coverage and test reducibility are better indicators in most cases.

6.1.3 An imperfect reference metric

While mutation coverage is certainly a more faithful measure of test suite effectiveness than syntactic coverage metrics, it has ample flaws of its own:

- *Failure to acknowledge equivalence.* The most obvious flaw is that, at least in its raw form (ratio of killed mutants over total generated mutants) mutation coverage fails to take into account the fact that mutants may survive a test, not because of a flaw in the test suite, but because the mutants are semantically equivalent to the base program.
- *Failure to acknowledge redundancy.* Even if we knew which mutants are killable, or knew how to estimate their number [14], we would still have a problem: the same mutation score means vastly different things depending on whether the killed mutants

are all semantically distinct, all semantically equivalent, or partitioned into multiple equivalence classes. Hence, for example, if test suite $T1$ kills three equivalent mutants and test suite $T2$ kills two distinct mutants then the mutation score of $T1$ is greater than that of $T2$ even though $T2$ is actually more effective than $T1$.

- *Failure to acknowledge partiality.* If we consider two test suites $T1$ and $T2$, and we find that they kill distinct sets of mutants, we have no objective basis for telling which test the suite is better, yet by assigning them mutation scores, we define a total ordering by default. Whenever we define a total ordering to represent what is, in fact, a partial ordering, we introduce built-in loss of precision: any two test suites can be compared by means of their mutation scores, even when we have no basis for considering that the suite with the higher score is more effective.
- *Failure to acknowledge the role of mutation operators.* Different references cite mutation coverage [5, 47, 64–68] without further qualification, as though all these references are talking about the same metric. In this chapter, we argue, on the basis of empirical observations, that the mutation coverage of a test suite may depend on a very large extent on the operators that are used to generate mutants. Our empirical results exceed our most extreme expectations: in some of the experiments we report below, we find very low statistical correlations between the values of mutation coverage obtained for different mutant generation policies; some are negative.

This has at least three immediate practical consequences:

- First, a measure of mutation coverage is meaningful only with reference to a particular mutant generation policy; hence, we cannot merely specify the mutation coverage of a test suite without specifying the mutation policy that is used to generate mutants.
- Second, mutation coverage experiments can be compared only if they refer to the same mutant generation policy.
- Third, to enhance the usefulness of empirical experiments on mutation coverage, it may be advantageous for researchers to agree on a small set of mutant generation policies [4, 5, 7, 36, 69]; then comparisons between experiments can only be made if they are based on the same policy.

The recognition that mutation operators have an impact on the measurement of mutation coverage is not new: Mresa and Bottaci [70] and Offut et al. [71] have observed and studied the variability of mutation coverage as a function of mutation operators. In this chapter, we analyze the statistical correlation between mutation coverage values obtained for the same test suites using different mutant generation policies; we do so by means of four different measures of mutation coverage, which we discuss below.

6.1.4 Quantifying mutation coverage

For the purposes of our experiment, we use four measures of mutation coverage:

- *Raw Mutation Score, RMS.* This is the traditional definition of mutation score as the ratio of killed mutants over the total number of generated mutants.
- *Prorated Mutation Score, PMS.* Assuming we have identified and excluded equivalent mutants, this is the ratio of killed mutants over the number of killable (non-equivalent) mutants.
- *Equivalence-based Mutation Score, EMS.* For the purpose of this metric, we assume not only that we have identified and excluded those mutants that are equivalent to the base program we also assume that we have partitioned the set of killable mutants into equivalence classes modulo semantic equivalence. We know by definition that once a test suite kills a mutant, it necessarily kills all the mutants in the same equivalence class. Then *EMS* is the ratio of the number of equivalence classes killed over the total number of equivalence classes.
- *Mutation Tally, MT.* Test suite effectiveness cannot be determined solely on the basis of the number of mutants killed, regardless of how we count them, because mutants are not created equal: In [62] Kaufman et al. introduce the criterion of *TCAP: Test Completeness Advancement Probability* as a way to quantify mutant usefulness, and show that this measure may vary widely between mutants. This leads us to conclude that the only way to claim with certainty that $T1$ is better than (or as good as) $T2$ (for a particular set of mutants) is that all the mutants killed by $T2$ are killed by $T1$. Whence we define a new attribute of test suite effectiveness: The *mutation tally* of test suite T for a set of mutants μ is the set of mutants in μ that are killed by T. Unlike the three previous metrics, mutation tally, is not a number but a set; as such, this attribute ranks test suites by a partial ordering (set inclusion); this is not a bug but a feature, since test suite effectiveness is itself a partially ordered attribute.

For the sake of illustration, we consider a benchmark Java program from the `ArrayUtil` class, and we apply the *AND* mutation operators of LittleDarwin [10], which yield 14 mutants:

$$\mu = \{M1, M3, M5, M7, M10, M12, M14, M17, M19, M22, M24,$$
$$M27, M30, M41\}.$$

For the sake of argument, we use a test suite $T0$ of size 104 elements to identify mutants that are equivalent to the base program and to partition killable mutants by semantic equivalence; using $T0$, we kill six out the 14 mutants, and conclude that the surviving mutants are equivalent to the base program. Killed mutants:

$\mu' = \{M10, M12, M14, M22, M24, M41\}$.

Using the same test suite, we partition the killed mutants into equivalence classes modulo semantic equivalence, and we find the following partition:

$C1 = \{M10, M12, M14\}$.

$C2 = \{M22\}$. $C3 = \{M24\}$. $C4 = \{M41\}$.

We consider three test suites,

- $T5$, of size 21, which kills mutant $\{M41\} = C4$.
- $T12$, of size 62, which kills mutants
  $\{M22, M24, M41\} = C2 \cup C3 \cup C4$.
- $T16$, of size 83, which kills mutants
  $\{M10, M12, M14, M22, M24\} = C1 \cup C2 \cup C4$.

These sets represent the mutation tallies of the test suites; as for their numeric mutation scores (*RMS, PMS, EMS*), they are given in the following table:

Test	*RMS*		*PMS*		*EMS*	
Suite	Formula	Value	Formula	Value	Formula	Value
$T5$	1/14	0.071	1/6	0.167	1/4	0.250
$T12$	3/14	0.214	3/6	0.500	3/4	0.750
$T16$	5/14	0.357	5/6	0.833	3/4	0.750

In terms of mutation tallies, $T5$ is lower than $T12$ and $T16$, while $T12$ and $T16$ are unrelated. The numeric mutation scores give three distinct orderings of $T5$, $T12$ and $T16$:

- *RMS*: $T12$, $T16$, $T5$.
- *PMS*: $T5$, $T12$, $T16$.
- *EMS*: $T5$, $\{T12, T16\}$.

6.2 On the Divergence of Mutation Coverage

The literature in mutation testing refers to mutation coverage as if it were a uniform measure of test suite effectiveness; it does not make explicit reference to the mutant generation policy on which mutation coverage is based [30, 64, 72, 73]; in this chapter, we show that actually the mutant generation policy does matter, and we quantify by means of statistical analysis the extent to which it does. To this effect, we consider nine mutant-generation policies, implemented by two distinct tools, apply them to the same benchmark program, and we compute the mutation coverage of twenty test suites under each of the mutant generation policies; then, we analyze the statistical correlations between the results we obtain for the nine policies.

6.2.1 Experimental set up

The parameters of our experiment are as follows:

- *The Sample Program.* For the sample program, we have selected *jTerminal* a simulator of a VT100 terminal, due to [74].
- The two mutant generator tools are: *Major* [75] and *LittleDarwin* [10].
  We deployed Major with five operator configurations:
  - *AOR: Arithmetic Operator Replacement.* This mutation operator produced 28 mutants.
  - *COR: Conditional Operator Replacement.* This mutation operator produced 70 mutants.
  - *ROR: Relational Operator Replacement.* This mutation operator produced 161 mutants.
  - *STD: Statement Delection.* This mutation operator produced 97 mutants.
  - *All: {AOR, COR, ROR, STD}*. Together, this set of mutation operators produced 338 mutants.

  As for *LittleDarwin*, it was deployed with the following mutation operators:
  - *AND: AND Replacement Operator.* LittleDarwin labeled 14 mutants under this header; this operator replaces instances of AND with OR.
  - *OR: OR Replacement Operator.* LittleDarwin labeled 27 mutants under this header; this operator replaces instances of OR with AND.

  - *ROR: Replacing Relational Operators.* LittleDarwin labeled 82 mutants under this header; this operator mutates comparison operators (e.g., $a \geq b$ into $a \leq b$).
  - *All* = {*AND,OR,ROR*}. LittleDarwin gives a total of 123 mutants.

- *The Base Test Suite*, $T0$. The selected benchmark program (*jTerminal*) comes with a test suite of size 35 elements, to which we add two tests of our own, intended specifically to trip the program, i.e., make it fail to converge. We refer to this test suite as $T0$, and we assume (as a working hypothesis) that it is sufficiently large to determine semantic equivalence; a study by Hall [76] finds that the probability that two programs are semantically distinct given that they produce the same output for N inputs decreases exponentially with N.
- *The Sample Test Suites*, $T1$, ... $T20$. We select twenty subsets of $T0$, of varying sizes, as follows:
  - $T1$...$T5$: Distinct random subsets of $T0$, of size 32.
  - $T6$...$T10$: Distinct random subsets of $T0$, of size 27.
  - $T11$...$T15$: Distinct random subsets of $T0$, of size 22.
  - $T16$...$T20$: Distinct random subsets of $T0$, of size 36.

For each mutation experiment (application of a mutant generator to *jTerminal* using a specific operator or set of operators), we perform the following steps:

- *Estimate RMS.* For each test suite Ti, $1 \leq i \leq 20$, we execute the mutants on the test suite Ti and see how many mutants are killed by Ti. The ratio of killed mutants over the total number of generated mutants is the raw mutation score (*RMS*) of Ti.
- *Estimate PMS.* We execute all the mutants on $T0$ and see how many mutants survive the test. As a working hypothesis, we assume that the surviving mutants are semantically equivalent to the base program, and compute the prorated mutation score of each test suite Ti as the number of mutants killed by Ti over the number of mutants killed by $T0$.
- *Estimate EMS.* We focus on the mutants that are killed by $T0$ and we partition them into semantic equivalence classes, again using $T0$: if two mutants return the same output for all the elements of $T0$, we consider that they are semantically equivalent. Once the set of killable mutants is partitioned into semantic equivalence classes, we compute the equivalence-based mutation score (*EMS*) of each test suite Ti as the ratio of the number of equivalence classes killed by Ti over the number of equivalence classes killed by $T0$. Unlike PMS, EMS gives credit for the number of distinct mutants killed, and does not reward a test suite for killing the same mutant several times.
- *Compute MT and Rank test suites.* Whereas RMS, PMS, and EMS are numeric metrics, the mutation tally (*MT*) is an attribute in a partially ordered set (to reflect the partial nature of test suite effectiveness). For mutation tally, we map each test suite Ti to the set of mutants that it kills, and we consider the inclusion relationships between the mutation tallies of test suites.

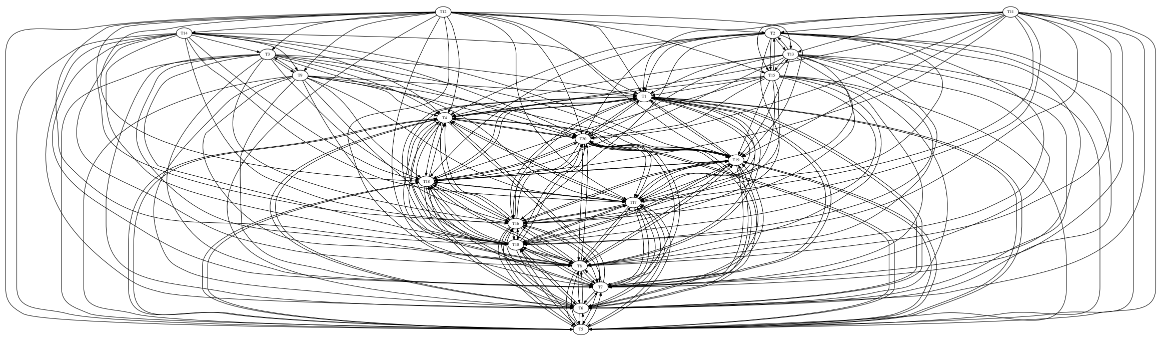


Figure 6.1 Graph of mutation Tally, for littleDarwin (All).

Once we compute these four metrics for all twenty test suites under all nine mutant generation policies, we analyze the relationships between the vectors of mutation coverage obtained for the mutant generation policies. For numeric metrics (*RMS*, *PMS*, *EMS*), we analyze the statistical correlations between the vectors of metrics. For the mutation tally, we consider the graphs that represent inclusion relationships between the mutation tallies and estimate the degree of similarity of each pair of graphs; we let the measure of similarity of two graphs $G1 = (V, E1)$ and $G2 = (V, E2)$ on the same set of nodes (V) be the Jaccard index of their sets of edges, i.e. $\frac{|E1 \cap E2|}{|E1 \cup E2|}$.

6.2.2 Raw data

Page limitation precludes us from presenting all our data in detail, hence, we show only some selected samples and refer the interested reader to the following webpage for details: `http://web.njit.edu/~sma225/thesis/chapter6/`.

For the sake of illustration, we show the vectors of RMS, PMS and EMS for LittleDarwin (All) and Major (All) in, respectively, Tables 6.1 and 6.2. The graphs that rank the test suites according to the inclusion relations between their mutation tally for LittleDarwin (All) and Major (All) are given in Figures 6.1 and 6.2, respectively.

Table 6.1 Mutation Coverage for LittleDarwin (All): 123 Mutants

Test Suite	Killed Mutant	Killed Equiv.C.	RMS	PMS	EMS
T0	59	47	0.460937	1.000000	1.000000
T1	35	28	0.273437	0.593220	0.595744
T2	36	29	0.281250	0.610169	0.617021
T3	38	29	0.296875	0.644067	0.617021
T4	29	23	0.226562	0.491525	0.489361
T5	32	26	0.250000	0.542372	0.553191
T6	41	35	0.320312	0.694915	0.744680
T7	48	38	0.375000	0.813559	0.808510
T8	56	44	0.437500	0.949152	0.936170
T9	44	33	0.343750	0.745762	0.702127
T10	43	36	0.335937	0.728813	0.765957
T11	54	42	0.421875	0.915254	0.893617
T12	58	46	0.453125	0.983050	0.978723
T13	52	42	0.406250	0.881355	0.893617
T14	55	43	0.429687	0.932203	0.914893
T15	56	44	0.437500	0.949152	0.936170
T16	58	46	0.453125	0.983050	0.978723
T17	58	46	0.453125	0.983050	0.978723
T18	57	45	0.445312	0.966101	0.957446
T19	56	44	0.437500	0.949152	0.936170
T20	58	46	0.453125	0.983050	0.978723

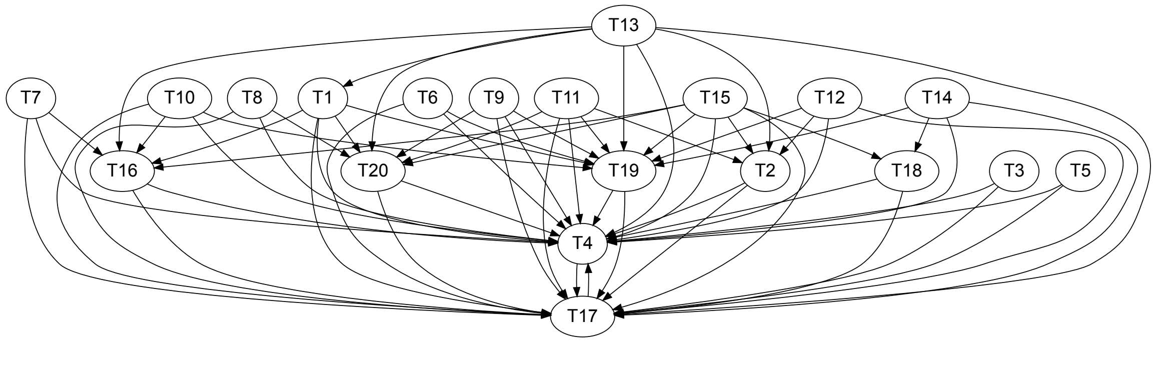


Figure 6.2 Graph of mutation tally, for major (All).

Table 6.2 Mutation Coverage for Major (All): 338 Mutants

Test Suite	Killed Mutant	Killed Equiv.C.	RMS	PMS	EMS
T0	148	79	0.437869	1.000000	1.000000
T1	137	73	0.405325	0.925675	0.924050
T2	141	75	0.417159	0.952702	0.949367
T3	137	74	0.405325	0.925675	0.936708
T4	148	79	0.437869	1.000000	1.000000
T5	129	72	0.381656	0.871621	0.911392
T6	126	71	0.372781	0.851351	0.898734
T7	120	67	0.355029	0.810810	0.848101
T8	140	75	0.414201	0.945945	0.949367
T9	140	76	0.414201	0.945945	0.962025
T10	123	70	0.363905	0.831081	0.886075
T11	119	64	0.352071	0.804054	0.810126
T12	114	64	0.337278	0.770270	0.810126
T13	110	58	0.325443	0.743243	0.734177
T14	125	70	0.369822	0.844594	0.886075
T15	126	67	0.372781	0.851351	0.848101
T16	146	77	0.431952	0.986486	0.974683
T17	148	79	0.437869	1.000000	1.000000
T18	147	78	0.434911	0.993243	0.987341
T19	146	78	0.431952	0.986486	0.987341
T20	147	78	0.434911	0.993243	0.987341

Table 6.3 Correlation Matrix for RMS

RMS		LittleDarwin				Major				
		All	AND	OR	Relation	All	AOR	COR	ROR	STD
LittleDarwin	All	1.0								
	AND	0.766	1.0							
	OR	0.659	0.279	1.0						
	REL	0.984	0.674	0.643	1.0					
Major	All	0.0152	0.114	0.115	**-0.055**	1.000				
	AOR	**-0.219**	**-0.153**	**-0.037**	**-0.228**	0.682	1.0			
	COR	0.201	**-0.005**	0.491	0.177	0.651	0.174	1.0		
	ROR	0.087	0.321	**-0.043**	**-0.008**	0.813	0.392	0.297	1.0	
	STD	0.100	0.200	0.079	0.029	0.810	0.544	0.489	0.650	1.0

6.2.3 Analysis

Even a casual look at Tables 6.1 and 6.2 reveals that the mutation coverage of test suites $T0$ ... $T20$ under the two mutant generator policies diverge widely: it suffices to compare the RMS column of Table 6.1 with the RMS column of Table 6.2; the same can be observed by comparing the PMS columns and the EMS columns of the two tables. Also, even a casual look at the graphs in Figures 6.1 and 6.2 reveals that these two graphs are very different in the way they rank test suites $T0$ ... $T20$. Note that both graphs are transitive; whenever there is an arc from Ti to Tj and from Tj to Tk there is also an arc from Ti to Tk. Whereas it is common to draw the transitive root of transitive graphs, we have resolved to draw the graph in full (including its transitive arcs), because we want the Similarity measure will be used to reflect the graphs as drawn.

The lack of correlation between the numeric mutation coverage metrics (RMS, PMS, EMS) is vastly confirmed by the correlation matrices given in Table 6.3 for RMS, Table 6.4 for PMS and Table 6.5 for EMS. This is also confirmed for the mutation tally by the similarity matrix shown in table 6.6. Negative entries in these tables are highlighted, for emphasis.

Table 6.4 Correlation Matrix for PMS

PMS		LittleDarwin				Major				
		All	AND	OR	Relation	All	AOR	COR	ROR	STD
LittleDarwin	All	1.0								
	AND	0.765	1.0							
	OR	0.659	0.279	1.0						
	REL	0.983	0.674	0.643	1.0					
Major	All	0.0152	0.114	0.115	**-0.055**	1.0				
	AOR	**-0.218**	**-0.153**	**-0.037**	**-0.228**	0.682	1.0			
	COR	0.201	**-0.005**	0.491	0.177	0.651	0.175	1.0		
	ROR	0.087	0.321	**-0.042**	**-0.008**	0.813	0.392	0.296	1.0	
	STD	0.100	0.200	0.079	0.029	0.810	0.543	0.488	0.651	1.0

Table 6.5 Correlation Matrix for EMS

EMS		LittleDarwin				Major				
		All	AND	OR	Relation	All	AOR	COR	ROR	STD
LittleDarwin	All	1.0								
	AND	0.752	1.0							
	OR	0.719	0.422	1.0						
	REL	0.986	0.678	0.706	1.00					
Major	All	**-0.067**	**-0.005**	0.048	**-0.119**	1.0				
	AOR	**-0.208**	**-0.172**	0.056	**-0.213**	0.751	1.0			
	COR	0.162	0.095	0.121	0.126	0.611	0.064	1.0		
	ROR	0.001	0.098	0.177	**-0.044**	0.934	0.719	0.505	1.0	
	STD	**-0.089**	0.014	**-8.176**	**-0.138**	0.991	0.714	0.608	0.924	1.0

Table 6.6 Similarity Matrix for Mutation Tally

MT		LittleDarwin				Major				
		All	AND	OR	Relation	All	AOR	COR	ROR	STD
LittleDarwin	All	1.0								
	AND	0.278	1.0							
	OR	0.568	0.390	1.0						
	REL	0.299	0.407	0.444	1.0					
Major	All	0.263	0.179	0.195	0.243	1.000				
	AOR	0.842	0.304	0.444	0.341	0.313	1.0			
	COR	0.389	0.342	0.376	0.303	0.363	0.436	1.0		
	ROR	0.461	0.317	0.367	0.541	0.570	0.548	0.516	1.0	
	STD	0.267	0.178	0.199	0.241	0.985	0.317	0.371	0.565	1.0

The first observation that we make about these results is that Tables 6.3 and 6.4 are identical; this is because RMS and PMS are in a linear relationship ($PMS = \lambda \times RMS$, where λ is the ratio of killable mutants); in subsequent experiments, we show only one copy of this matrix.

Among the numeric coverage metrics, it is fair to consider that EMS is perhaps the most meaningful metric since it excludes equivalent mutants from consideration (hence does not penalize a test suite for failing to kill an equivalent mutant), and is based on a count of equivalence classes modulo semantic equivalence (hence, does not give undue credit for killing the same mutant multiple times). Hence, we focus on Table 6.5. Within this table, perhaps the most meaningful correlation is that between LittleDarwin (All) and Major (All), since these mutant generation policies involve the highest number of mutants. The correlation between the EMS values assigned to the test suites $T1$ ... $T20$ under these two mutant generation policies is -0.0763. When we resolved to initiate this study, we were expecting to find a correlation in the range of 0.5 to 0.8, not a negative correlation altogether. The less perfect mutation coverage metrics of RMS and PMS show a correlation of 0.015176, also extremely low, suggesting that mutation coverage under one policy is (almost) statistically independent of mutation coverage under another. All our data is available online for interested readers to check:

`http://web.njit.edu/~sma225/thesis/chapter6/`.

Even more meaningful than EMS is, in our opinion, the mutation tally of a test suite: the only way we can tell that a test suite Ti is better than (or as good as) a test suite Tj is if (and only if) all the mutants that are killed by Tj are killed by Ti. Hence, we are interested in the similarity index of the graphs that rank test suites $T1$ ... $T20$ according to inclusion relationships between their mutation tallies. Table 6.6 shows the similarity indices between all the graphs of mutation tallies obtained for all the mutant generation policies. If we focus again on the graphs generated for LittleDarwin (All) and Major (All), given respectively in Figures 6.1 and 6.2, we find a similarity index of 0.263158. In other words, these two graphs agree on merely one quarter of the relationships that they define between test suites. Three-quarters of the inclusion relationships that are found in one graph are absent from the other; this reflects how vastly divergent these two measures of mutation coverage are. This, in turn, reflects the importance of the mutant generation policy in determining the mutation coverage of a test suite.

6.3 Further Empirical Observations

In the experiment of Section 6.2, we have explored the relationship between measures of mutation coverage based on different mutant generators (*LittleDarwin* vs. *Major*). From an observation of Tables 6.3, 6.4, 6.5 and 6.6, we find that the divergence between measures of mutation coverage is greater across different tools than it is between mutation operators of the same tool. In this section, we show the results of three experiments that involve different mutation operators within the same mutant generator tool (*LittleDarwin*): an experiment where the test suite sizes range over a narrow interval; an experiment where the test suite sizes range over a wide interval, and an experiment where the number of mutants are very large, and the test suite sizes range over the breadth of the cardinality of $T0$. In all cases, we find varying levels of divergence between the scores of mutation coverage, ranging from measurable to considerable. These three experiments are the subject of the next three subsections; for the second and third experiment, we merely specify the parameters of each experiment then presents the three matrices that reflect the relationships between measures of mutation coverage.

6.3.1 First experiment: a narrow spread of test suite sizes

For this experiment, we resolve to spread the sizes of the test suites $T1$... $T20$ narrowly as a percentage of the size of test suite $T0$. The parameters of this experiment are as follows:

- *Base Program*: `ArrayUtils`.
- *Base Test Suite Size ($T0$)*: 104.
- *Test Suite Sizes*: We form four sets of test suites; each set is a randomly selected subset of $T0$.
  - $T1$...$T5$: Distinct random subsets of $T0$ of size 10.
  - $T6$...$T10$: Distinct random subsets of $T0$ of size 21.
  - $T11$...$T15$: Distinct random subsets of $T0$ of size 31.
  - $T16$...$T20$: Distinct random subsets of $T0$ of size 42.
- *Mutation Generation Policies.* We have selected three mutation generation policies defined by three sets of operations provided by LittleDarwin:
  - *AND Replacement Operators.* This policy produces 14 mutants.
  - *Relational Operators.* This policy produces 82 mutants.
  - *OR Replacement Operators.* This policy produces 27 mutants.

Table 6.7 gives the correlation matrix between the RMS and PMS scores of the test suites under the different mutant generation policies; and table 6.8 gives the correlation matrix of the equivalence-based mutation scores of the twenty test suites under the three mutant generation policies. Figures 6.3, 6.5 and 6.10 show, respectively, the mutation tally graphs derived from the mutation experiments that use *AND Replacement*, *OR Replacement*, and *Relational Operation Replacement*; even a casual observation of these graphs shows that they rank test suites $T0$ ... $T20$ differently. This is confirmed by considerting Table 6.9, which gives the similarity index of the graphs of mutation tally for the three selected mutation strategies.

Whereas the similarity indices between the graphs of mutation tally are very low (ranging between 0.29 and 0.41), the statistical correlations between their numeric mutation

Table 6.7 Correlation RMS and PMS: First Experiment

RMS	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.000000	0.606179	0.611844
OR Replacement		1.000000	0.888815
Relational Ops			1.000000

Table 6.8 Correlation Matrix of EMS: First Experiment

EMS	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.000000	0.723452	0.710286
OR Replacement		1.000000	0.881895
Relational Ops			1.000000

Table 6.9 Similarity Index of MT Graphs: First Experiment

MT	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.0	0.41	0.29
OR Replacement		1.0	0.38
Relational Ops			1.0

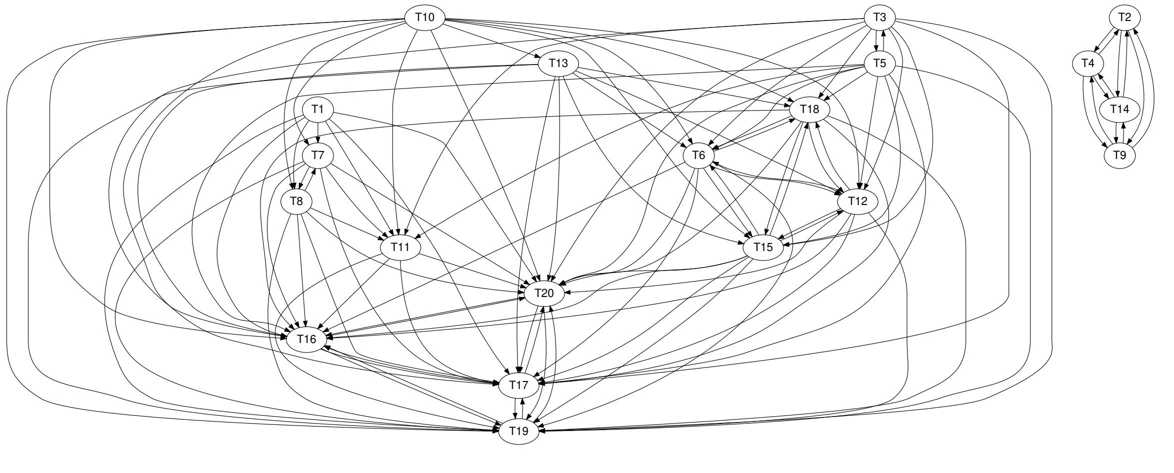


Figure 6.3 Graph of mutation tally, for AND replacement, first experiment.

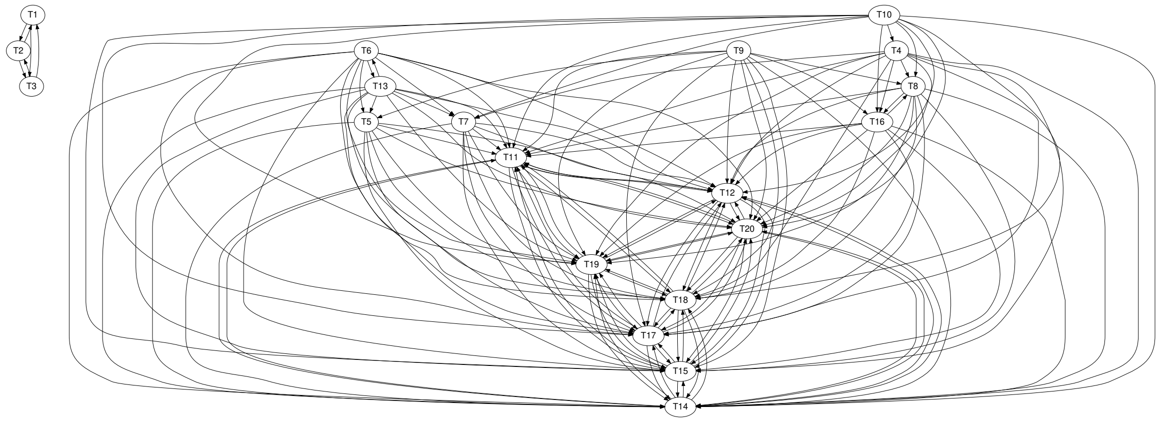


Figure 6.4 Graph of mutation tally, for AND replacement, first experiment.

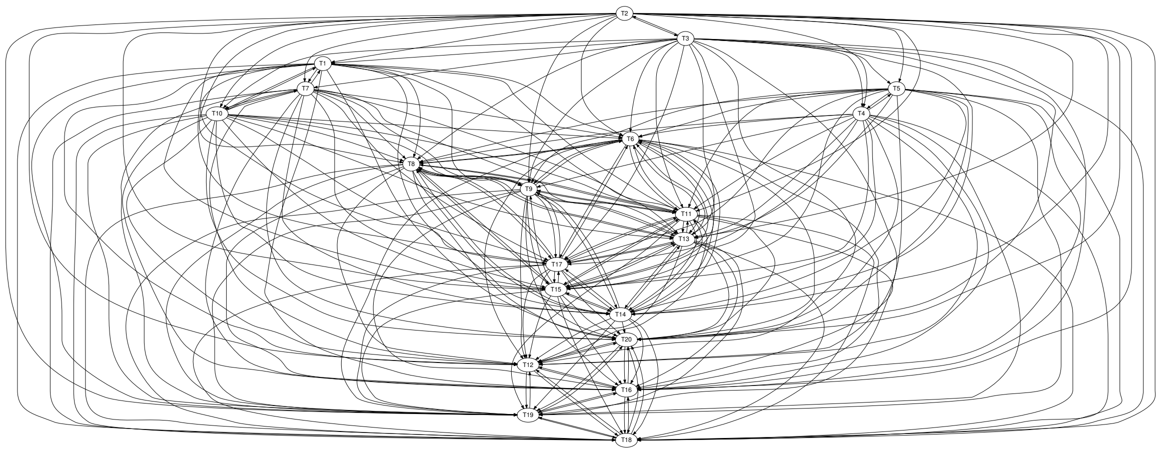


Figure 6.5 Graph of mutation tally, for OR replacement, first experiment.

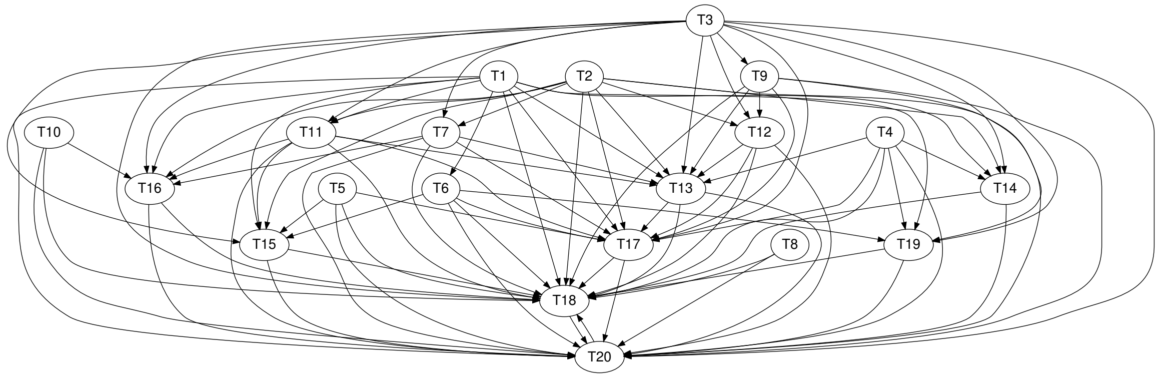


Figure 6.6 Graph of mutation tally, for relational operators, first experiment.

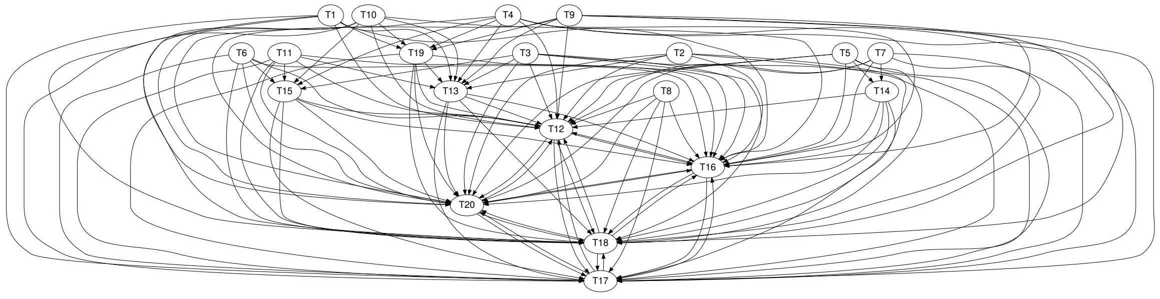


Figure 6.7 Graph of mutation tally, for relational operators, first experiment.

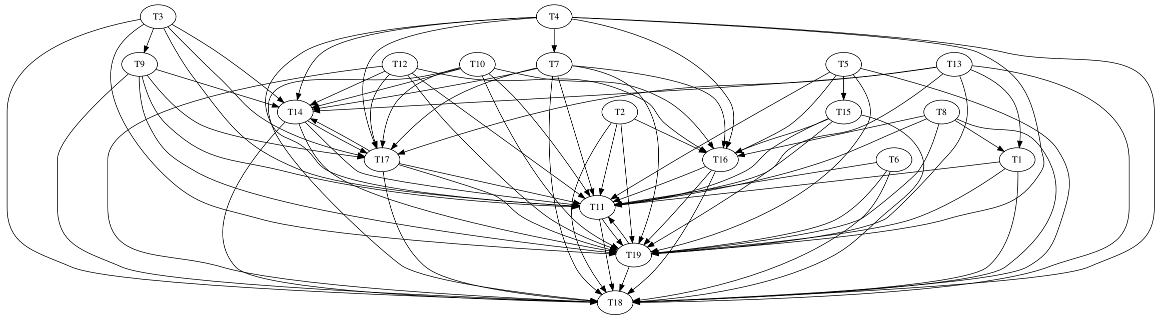


Figure 6.8 Graph of mutation tally, for relational operators, first experiment.

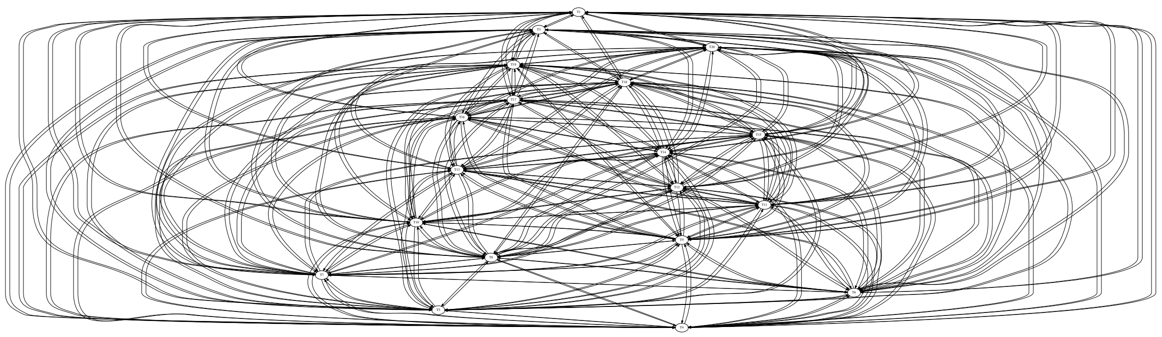


Figure 6.9 Graph of mutation tally, for relational operators, first experiment.

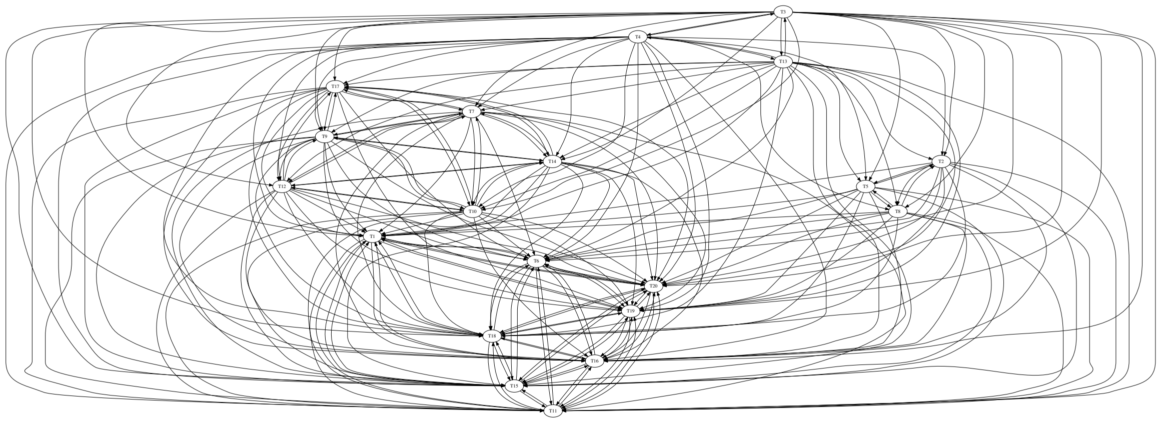


Figure 6.10 Graph of mutation tally, for relational operators, first experiment.

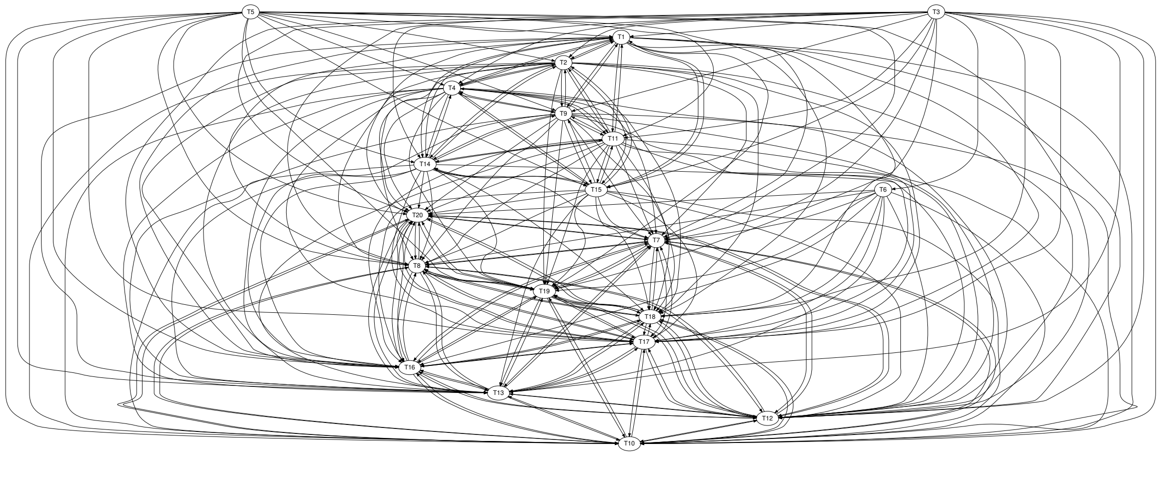


Figure 6.11 Graph of mutation tally, for relational operators, first experiment.

scores are not as low (ranging between 0.60 and 0.88). We qualify this observation with two premises: First, we consider that mutation tally is a more meaningful reflection of relative effectiveness (the property of a test suite to be more effective than another) than the numeric mutation scores (*RMS*, *PMS*, *EMS*). Second, we do not claim that mutation scores are always un-correlated; we only claim that they are not always correlated.

6.3.2 Second experiment: a wide spread of test suite sizes

Whereas in the previous experiment, we used test suite sizes that are a small fraction of the size of $T0$ (10%, 20%, 30%, 40%), in this experiment, we select higher sizes; by spreading test suite sizes over a wider range of values, we anticipate the estimates of test suite effectiveness to differ more widely between different test suites. The parameters of this experiment are as follows:

- *Base Program*: `ArrayUtils`.
- *Base Test Suite Size ($T0$)*: 104.
- *Test Suite Sizes*: We form four sets of test suites, each set is a randomly selected subset of $T0$.
  - $T1...T5$: Dinstinct random subsets of $T0$ of size 21.
  - $T6...T10$: Distinct random subsets of $T0$ of size 42.
  - $T11...T15$: Distinct random subsets of $T0$ of size 62.
  - $T16...T20$: Distinct random subsets of $T0$ of size 83.
- *Mutation Generation Policies.* We have selected three mutation generation policies defined by three sets of operations provided by LittleDarwin:
  - *AND Replacement Operators.* This policy produces 14 mutants.
  - *Relational Operators.* This policy produces 82 mutants.
  - *OR Replacement Operators.* This policy produces 27 mutants.

Table 6.10 shows the correlation matrix of RMS and PMS for the three mutant generation policies; table 6.11 gives the correlation matrix of the equivalence-based mutation scores

Table 6.10 Correlation Matrix of RMS and PMS: Second Experiment

RMS	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.0	0.2793361	0.67401821
OR Replacement		1.0	0.6430149
Relational Ops			1.0

Table 6.11 Correlation Matrix of EMS: Second Experiment

EMS	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.0	0.4225233	0.67879997
OR Replacement		1.0	0.706133
Relational Ops			1.0

of the twenty test suites under the three mutant generation policies. Table 6.12 gives the similarity index of the graphs of mutation tally for the three selected mutation policies.

6.3.3 Third experiment: a larger pool of mutants

In this experiment, we select a program and mutation operators that yield larger mutant pools to get past possible side effects. The parameters of this experiment are as follows:

- *Base Program*: `jTerminal`, a simulator of a VT100 terminal [74].
- *Base Test Suite Size ($T0$)*: 35.

Table 6.12 Similarity Index of the MT Graphs: Second Experiment

MT	AND Replacement	OR Replacement	Relational Ops
AND Replacement	1.0	0.39	0.41
OR Replacement		1.0	0.44
Relational Ops			1.0

Table 6.13 Correlation Matrix of RMS, PMS: Third Experiment

RMS	ROR	COR	AORB
ROR	1.0	0.543853	0.594772
COR		1.0	0.082144
AORB			1.0

Table 6.14 Correlation Matrix for EMS: Third Experiment

EMS	ROR	COR	AORB
ROR	1.0	0.452972	0.662319
COR		1.0	0.082144
AORB			1.0

- *Test Suite Sizes*: We form four sets of test suites, each set is a randomly selected subset of $T0$.
  - $T1...T5$: Distinct random subsets of $T0$ of size 14.
  - $T6...T10$: Distinct random subsets of $T0$ of size 19.
  - $T11...T15$: Distinct random subsets of $T0$ of size 24.
  - $T16...T20$: Distinct random subsets of $T0$ of size 29.
- *Mutation Generation Policies.* We deploy *LittleDarwin* the following mutation operators, and give the number of mutants produced by each:
  - *Relational Operator Replacement (ROR)*: 165 mutants.
  - *Conditional Operator Replacement (COR)*: 39 mutants.
  - *Arithmetic Operation Replacement, Binary (AORB)*: 21 mutants.

Note that the *ROR* and *COR* operators used in this experiment are different from the *ROR* and *COR* operators used in the experiment reported on in Section 6.2: whereas the former are *LittleDarwin* operators, the latter are *Major* operators; hence, the fact that they produce a different number of mutants on the same base program is not an issue. Table 6.13 shows the correlation matrix of RMS and PMS for the three mutant generation policies; table 6.14 gives the correlation matrix of the equivalence mutation scores (EMS) of the twenty test suites under the three mutant generation policies. Table 6.15 gives the similarity index of the graphs of mutation tally for the three selected mutation strategies.

Table 6.15 Similarity Matrix of MT Graphs: Third Experiment

RMS	ROR	COR	AORB
ROR	1.0	0.2225	0.3665
COR		1.0	0.5770
AORB			1.0

6.4 Conclusion

6.4.1 Summary

In software testing, the mutation coverage of a test suite is often equated with its effectiveness, for good reason: to the extent that mutations represent faults [7, 35, 36], a test suite that detects mutations is likely to detect faults. This has led researchers to consider mutation coverage as a defacto standard of test suite effectiveness, so much so that a finding such as *"XYZ Coverage is Not Strongly Correlated to Mutation Coverage"* is usually perceived as the *kiss of death* of *XYZ* [5, 47, 64–68]. In this chapter, we extend this courtesy to *Mutation Coverage* by attempting to prove that, depending on what mutant generation policy one adopts, mutation coverage itself is not strongly correlated with mutation coverage.

Also, when researchers talk about mutation coverage without explicitly citing the mutant generation policy on which the mutation coverage is defined, they assume implicitly that somehow, the mutation coverage is independent of the mutation generation policy, or at least is not highly affected by it.

The empirical results presented in this chapter strongly undermine this assumption:

- Our data shows that when different mutant generation tools are used, the estimates of mutation coverage can differ very widely: The lower left quadrant of Table 6.5 contains some very low figures, including, interestingly, negative values; even zero values (of which there are many) are very surprising, since they mean that mutation coverage metrics assigned by different mutant generation experiments are statistically independent of each other.

- Even different mutant generation operators within the same tool may yield surprisingly low correlation values: we find some very low figures, as low as 0.08 between mutation coverage metrics of test suites for the same program, stemming from the same mutant generation tool.
- Also, to the extent that mutation tally is ultimately the most reliable indicator of relative effectiveness, even a casual look at the graphs of mutation tally show that they differ significantly from each other for different mutation policies. This observation is confirmed by the similarity indices of the graphs, which range between 0.22 and 0.57.

6.4.2 Threats to validity

The results presented here stem from a small scale experiment, and do not support any broad claims; but our goal is not to prove any claim, as much as it is to disprove the assumption that mutation coverage is independent of the mutant generation policy.

6.4.3 Implications and prospects

The short-term implication of this study is that we should consider that mutation coverage is not an attribute of the test suite alone, but must refer explicitly to the mutant generation policy; also, mutation scores cannot be compared across mutation policies, but must be compared only for the same policy. Another, conceptual, implication is the need to explore other means to assess the effectiveness of test suites in a way that better reflects the intrinsic attributes of the test suite, the program being tested, and the specification/ oracle against which it is tested.

CHAPTER 7

WHAT COVERAGE METRICS MEAN, AND WHAT THEY DO NOT

7.1 Assessing Test Suite Effectiveness

7.1.1 Coverage metrics: an imperfect compromise

To quantify the effectiveness of test suites, researchers and practitioners have routinely used *coverage metrics*, i.e. metrics that quantify to what extent a test suite exercises syntactic features (statements, branches, conditions, paths, etc) of the program under test [45]; the rationale for this approach is that in order for a test suite to sensitize a fault, it must execute the code that contains the fault. An alternative approach to quantifying the effectiveness of a test suite is to equate the effectiveness of a test suite with its ability to kill mutants of the program under test; the rationale of this approach is that, to the extent that mutations are faithful representation of actual faults [4, 5, 7, 8], the ability to detect faults is the same as the ability to kill mutants.

All these metrics are clearly imperfect: Syntactic coverage metrics are imperfect because exercising a program's syntactic attributes is neither necessary nor sufficient to detect all its faults. It is insufficient because not all executions of a faulty statement sensitize the fault, and not all fault sensitizations lead to a program failure. Strictly speaking, exercising all the syntactic features of a program are not necessary either to detect all its faults: it suffices only to exercise its faulty features (of course, in practice, we do not know which features are faulty; hence, we aim to execute all of them). While mutation coverage is usually seen as a more reliable measure of test suite effectiveness, it has issues of its own:

- *Mutant Equivalence.* In its raw form, the mutation score of a test suite may mean vastly different things depending on the fraction of semantically equivalent mutants to the base program. A mutation score of 0.5 is excellent if half of the mutants are equivalent to the base program, much less so if no mutant is.

- *Mutant Redundancy.* The same mutation score of a test suite may mean vastly different things depending on the extent of equivalence between the mutants that it killed: there is a difference between killing 50 semantically distinct mutants and killing the same mutant 50 times (by killing 50 syntactically distinct but semantically equivalent mutants).
- *Dependence on Mutation Policy.* In [22] we show that the mutation score of the same test suite can vary widely depending on what mutant generator we use and what mutation operators we activate within the same mutant generator. If a test suite has a mutation score of 0.2 for one mutation policy and a mutation score of 0.8 for another, what can we infer about its effectiveness?

Generally, all coverage metrics share two other weaknesses:

- *Inherent Imprecision.* Given two test suites, T and T', it is not always possible to adjudicate the question of which is better; they may be unrelated and incomparable; hence, the relation of *being a better test suite* is not a total ordering, but rather a partial ordering. By assigning numeric values to assess test suite effectiveness, we define a total ordering to represent what is essentially a partial ordering; hence, we create an inherent built-in source of imprecision since any two numeric coverage metrics can always be compared, even when their corresponding test suites are not in an ordering relation.
- *Context Independence.* All the coverage metrics are defined as a function of the test suite and the program under test, but whether a test suite T is effective also depends on the correctness property that we are using T for, as well as the specification against which correctness is tested.

7.1.2 Agenda

in this chapter, we envision to analyze the validity/soundness of existing coverage metrics

as measures of test suite effectiveness. To this effect, we proceed in three steps:

- First, we consider the concept of *semantic coverage*, introduced in [20], and discuss how it is designed to represent test suite effectiveness.
- Second, we discuss why we resolve to use semantic coverage as the the standard against which we assess coverage metrics.
- Third, we report on an experiment where we took a benchmark program, selected twenty test suites thereof, computed their respective semantic coverage, then compared the ordering defined by semantic coverage with the ordering defined by each traditional coverage metric.

The semantic coverage of a test suite is defined not only with respect to the program under test, but also with respect to the standard of correctness that the program is being tested for (partial correctness, total correctness), and the specification against which correctness is tested. Semantic coverage cannot be estimated easily in practice; hence, we do not view it as an alternative to common coverage metrics, but we use it to assess how much credibility we can lend to these metrics.

In Section 7.2, we present and prove properties about semantic coverage to justify further why we are adopting it as the basis against which we assess other coverage metrics. Section 7.3 forms the main body of our work: we discuss the details of our experiment, present the empirical results of our experiment, comment on them, and draw some conclusions and observations. We conclude in Section 7.4 with a summary of our paper, a critique of its results, and future research prospects.

7.2 Using Semantic Coverage

The semantic coverage of a test suite T for a standard of correctness (partial or total) of a program P with respect to a specification R depends on four factors: T, P, R and the standard of correctness. In this section, we discuss how one would want a measure of test suite effectiveness to vary as a function of each of these parameters, then we show that semantic coverage does meet the declared criteria. We start with citing and justifying the criteria.

- *Monotonicity with respect to T.* Of course, we want the effectiveness of a test suite to be monotonic with respect to T: if we replace T by a superset, we get a higher semantic coverage.
- *Monotonicity with respect to R.* Specifications are ordered by refinement whereby a more refined specification represents a more stringent requirement. We argue that it is easier to test a program for correctness against a specification R than against a specification R' that refines R; indeed, a more refined specification involves a larger input domain (hence, a larger set to cover) and stronger output conditions (hence,

more conditions to verify). Whence we expect that the same test suite T has lower semantic coverage for more refined specifications: i.e., semantic coverage ought to decrease when R grows more refined.

- *Monotonicity with respect to P.* If and only if program P' is more-correct than program P, the detector set of P' is a subset of the detector set of P, which means that we have fewer failures of P' to reveal than failures of P. Hence, the semantic coverage of a test suite T ought to be higher for a more correct program.
- *Monotonicity with respect to the standard of correctness.* Total correctness is a stronger property than partial correctness, hence, it is more difficult to test a program for total correctness than for partial correctness. Consequently, the same test suite T ought to have a lower semantic coverage for total correctness than for partial correctness (the same tool would be less effective against a more difficult task than an easier task).

With the exception of the first monotonicity property (with respect to T), one would be hard-pressed to claim or prove any of the monotonicity properties cited above about any traditional coverage metric (e.g., statement coverage). We present below Propositions to the effect that semantic coverage satisfies all these monotonicity properties; these are due to [20], and are given without proof.

Proposition 18. Monotonicity with respect to T. *Given a program P on space S and a specification R on S, and given two subsets T and T' of S, if $T \subseteq T'$ then:*

$$\Gamma_{R,P}^{TOT}(T) \subseteq \Gamma_{R,P}^{TOT}(T'),$$

$$\Gamma_{R,P}^{PAR}(T) \subseteq \Gamma_{R,P}^{PAR}(T').$$

Proposition 19. Monotonicity with respect to the standard of correctness. *Given a program P on space S, a specification R on S, and test suite T (subset of S), the semantic coverage of T for partial correctness of P with respect to R is greater than or equal to the* *semantic coverage for total correctness of P with respect to R:*

$$\Gamma_{R,P}^{TOT}(T) \subseteq \Gamma_{R,P}^{PAR}(T).$$

Proposition 20. Monotonicity with respect to relative total correctness of P. *Given a specification R on space S and two programs P and P' on S, and a subset T of S. If P' is more-totally-correct than P with respect to R then:*

$$\Gamma_{[R,P]}^{TOT}(T) \subseteq \Gamma_{[R,P']}^{TOT}(T).$$

Proposition 21. Monotonicity with respect to relative partial correctness of P. *Given a specification R on space S and two programs P and P' on S, and a subset T of S. If P' is more-partially-correct than P with respect to R then:*

$$\Gamma_{[R,P]}^{PAR}(T) \subseteq \Gamma_{[R,P']}^{PAR}(T).$$

Proposition 22. Monotonicity with respect to Refinement of R. *Given a program P on space S and two specifications R and R' on S, and a subset T of S. If R' refines R then:*

$$\Gamma_{[R',P]}^{TOT}(T) \subseteq \Gamma_{[R,P]}^{TOT}(T).$$

$$\Gamma_{[R',P]}^{PAR}(T) \subseteq \Gamma_{[R,P]}^{PAR}(T).$$

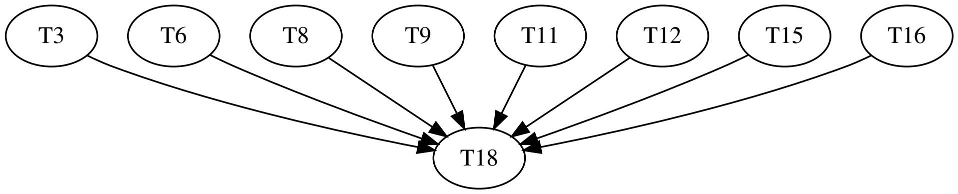


Figure 7.1 Sorting $T1$ ... $T20$ by semantic coverage of P with respect to $R1$ for partial correctness.

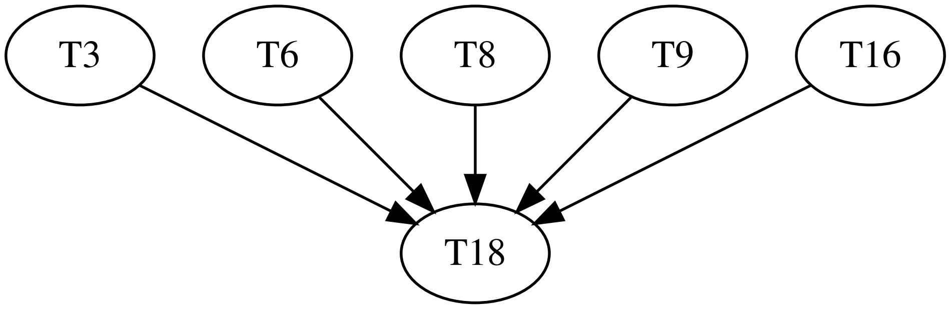


Figure 7.2 Sorting $T1$ ... $T20$ by semantic coverage of P with respect to $R1$ for total correctness.

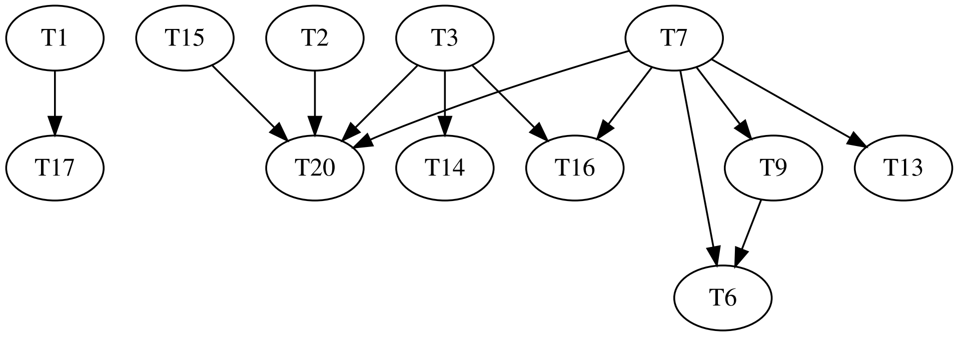


Figure 7.3 Sorting $T1$ ... $T20$ by Semantic Coverage of P with respect to $R2$ for partial correctness.

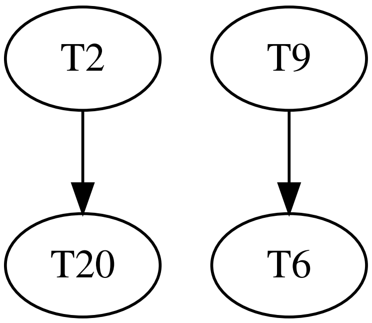


Figure 7.4 Sorting $T1$ ... $T20$ by semantic coverage of P with respect to $R2$ for total correctness.

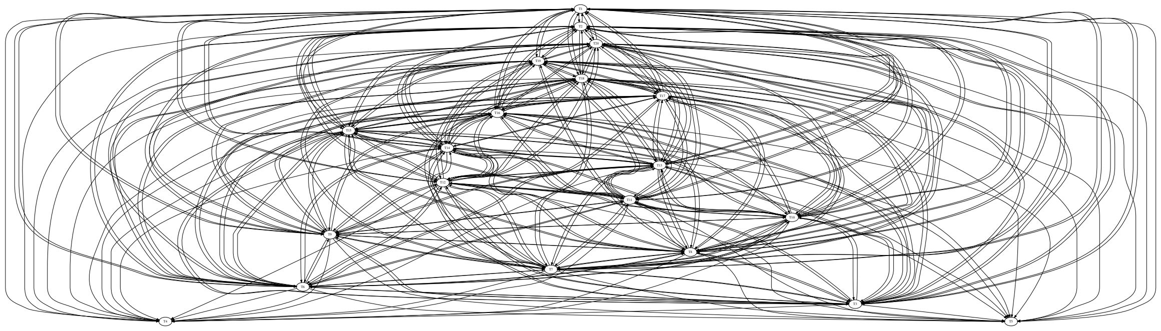


Figure 7.5 Sorting $T1$ ... $T20$ by semantic coverage of P with respect to $R3$ for partial correctness.

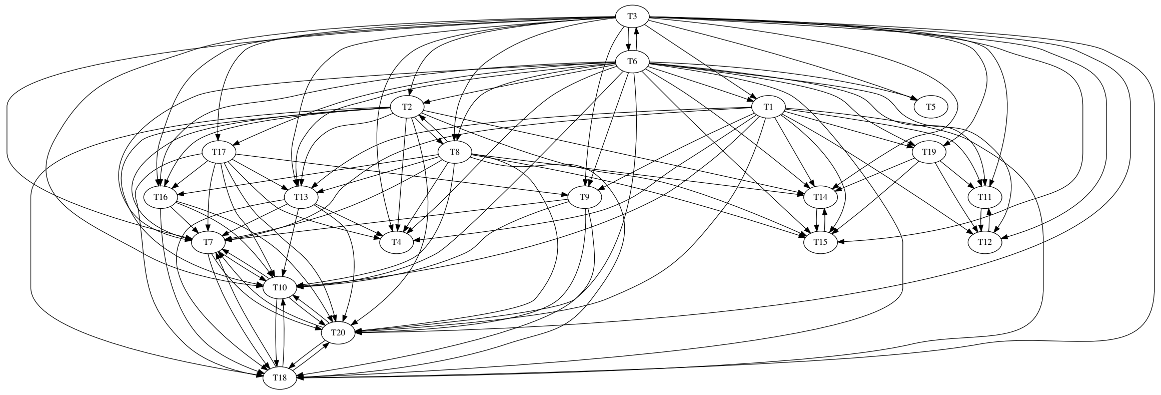


Figure 7.6 Sorting $T1$ ... $T20$ by semantic coverage of P with respect to $R3$ for total correctness.

7.3 What Coverage Metrics Mean

7.3.1 Experiment design

In light of the stepwise process that we followed to derive the formula of semantic coverage, and in light of the monotonicity properties that we have shown this formula to satisfy, we resolve to use semantic coverage as a basis for assessing the validity of coverage metrics to measure test suite effectiveness. To this effect, we run an experiment where:

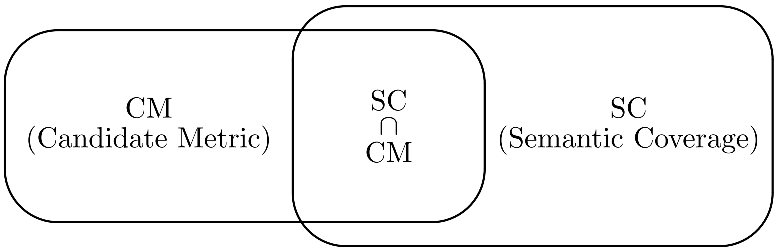
- We consider a benchmark program P, along with its associated benchmark test class $T0$.
- We derive twenty random subsets of $T0$, say $T1$ ... $T20$, whose sizes range between 40% and 60% of the size of $T0$.
- We select three specifications, say $R1$, $R2$ and $R3$ for P, and we compute the semantic coverage of each test suite $T1$ .. $T20$ for partial correctness and total correctness with respect to $R1$, $R2$, $R3$; this gives us six ordering relations between test suites $T1$ ... $T20$ defined by semantic coverage.
- We compute traditional coverage metrics (statement coverage, branch coverage, line coverage, mutation coverage, etc.) of the test suites $T1$ ... $T20$ using online tools.
- We check to what extent the traditional coverage metrics are correlated with (give us information about) the semantic coverage of the test suite. In other words, if we find that some test suite Ti has a greater coverage metric than some test suite Tj, how confident can we be that Ti has higher semantic coverage than Tj?
- To answer the above question, we proceed as follows: We draw the graph of semantic coverage of the test suites for all six combinations cited above (partial correctness and total correctness with respect to $R1$, $R2$, $R3$). Then for each coverage metric, say CM, we compute the number of ordering relations by semantic coverage that are borne out by the coverage metric, and we estimate three performance indicators: Precision; Recall; and Jaccard Index. See Figure 7.7.

7.3.2 Experiment's implementation

The following parameters can characterize our experiment:

- *The Program, P.* The sample program that we use for this experiment is a method called `createNumber()` of the Java class `NumberUtils.java`, from the commons benchmark (`commons-lang3-3.13.0-src`)[1]. The size of the selected method is 170 lines.

[1]`https://commons.apache.org/proper/commons-lang/`



$$\text{precision} = \frac{|SC \cap CM|}{|CM|} \quad \text{Jaccard} = \frac{|SC \cap CM|}{|SC \cup CM|} \quad \text{recall} = \frac{|SC \cap CM|}{|SC|}$$

SC: Set of Inequalities by Semantic Coverage. CM: Set of Inequalities by Candidate Metric.

Figure 7.7 Assessing compliance between CM and SC.

- *Base Test Suite,* $T0$. We consider the test class that comes with the selected program: `class NumberUtilsTest.java`. This class includes 107 tests.
- *Test Suites* $T1$, $T2$, ... $T20$. To generate these test suites, we run the following script, where `rand()` returns random numbers between 0.0 (inclusive) and 1.0 (exclusive):

```
threshold = 0.4;
for (int i=1; i<=20; i++)
   {print ("test suite t",i);
    int size=0;  threshold = threshold + 0.01;
    for (int j=1; j<=107; j++)
        {if (rand()<=threshold) {print (j); size++;}
    print ("size of test suite t",i,": ", size);}
```

This yields the following sizes for the test suites:

T0	107
T1	48
T2	40
T3	42
T4	47
T5	48
T6	46
T7	48
T8	45
T9	46
T10	55
T11	61
T12	59
T13	58
T14	50
T15	59
T16	61
T17	70
T18	80
T19	62
T20	70

- *Syntactic Metrics.* We use *jaCoCo* (`https://www.eclemma.org/jacoco/`) to compute *Statement Coverage* and *Branch Coverage* of the test suites $T1$ ... $T20$. We use *PiTest* (`https://pitest.org/`) to compute the *Line Coverage* of the test suites.

- *Mutation Coverage.* We define three metrics for mutation coverage.

  - *RMS: Raw Mutation Score.* This is the usual mutation score, defined as the ratio of the number of killed mutants over the number of generated mutants. We use *PiTest* for this purpose.

  - *PMS: Prorated Mutation Score.* We consider that any mutant that survives all the tests of $T0$ must be equivalent to the base program, and we define the Prorated Mutation Score of a test Ti, for $1 \leq i \leq 20$, as the ratio of the number of mutants killed by Ti over mutants killed by $T0$.

  - *EMS: Equivalence Based Mutation Score.* Rather than count individual mutants, we count equivalence classes of mutants modulo semantic equivalence, where we consider that two mutants are semantically equivalent if and only if they generate the same outputs for all tests in $T0$. Then *EMS* of test suite Ti is the ratio of the number of equivalence classes killed by Ti over the number of equivalence classes killed by $T0$.

- *Specifications.* To generate specifications in the form of binary relations, we run the base program on all the tests of the test class and record the (input, output) pairs. Then we scan these pairs and make some modifications as follows:

  - *R1.* For $R1$, we change the output for each fifth input (i.e., 5th, 10th, 15th, etc...). This means that the base program fails the test for each one of these.

- *R2*. For $R2$, we change the output for each seventh input (i.e., 7th, 14th, 21st, etc..).
- *R3*. For $R3$, we change the output for each eleventh input (i.e., 11th, 22nd, 33rd, etc...).

The semantic coverage of test suites $T1$ ... $T20$ for the six experiments (partial correctness and total correctness with respect to $R1$, $R2$, $R3$) are computed using the formulas given in Section 7.2, and compared for inclusion. The inclusion relationships are represented in the graphs depicted by Figures 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6.

Then for each coverage metric CM (statement coverage, branch coverage, line coverage, RMS, PMS, and EMS); we review how many inequalities provided by the metric are borne out by semantic coverage (SM); this gives us the cardinality of $(CM \cap SC)$; see Figure 7.7. The cardinality of SC for a semantic coverage experiment with respect to a given specification and correctness standard is simply the number of arcs in the corresponding graph. The cardinality is SC is, interestingly, the same for all the numeric metrics: 190. Indeed, given any set of 20 distinct numbers, there are exactly 190 inequalities between them ($\frac{20\times 19}{2}$), since the largest number is greater than 19; the second largest is greater than 18, etc. As for the cardinality of $(SC \cup CM)$, it can be estimated by the following formula:

$$|SC \cup CM| = |SC| + 190 - |SC \cap CM|.$$

7.3.3 Experimental data

Table 7.1 shows the values computed for statement coverage, branch coverage, line coverage, and RMS for the 20 test suites $T1$ ... $T20$. Table 7.2 shows in greater detail how the prorated mutation score and the equivalence-based mutation score are evaluated by analyzing mutant equivalence and redundancy.

Table 7.1 Metrics Table: $T1$ ... $T20$

Test Suite	Statement Coverage	Branch Coverage	Line Coverage	RMS
T0	.94	.83	.83	.71
T1	.90	.79	.80	.67
T2	.84	.72	.74	.57
T3	.87	.79	.78	.69
T4	.88	.76	.80	.67
T5	.85	.75	.76	.67
T6	.87	.77	.79	.69
T7	.93	.79	.80	.69
T8	.90	.78	.76	.65
T9	.88	.77	.80	.68
T10	.89	.75	.80	.69
T11	.90	.80	.81	.69
T12	.93	.80	.81	.69
T13	.87	.77	.79	.68
T14	.91	.76	.80	.64
T15	.91	.80	.82	.69
T16	.92	.80	.81	.67
T17	.92	.78	.80	.66
T18	.93	.82	.79	.68
T19	.94	.81	.82	.70
T20	.89	.80	.78	.68

Table 7.2 Mutation Coverage for: $T1$ ... $T20$

Test Suite	Killed Mutants	Killed EC	PMS	EMS
T0	89	67	1	1
T1	85	57	0.95505618	0.850746269
T2	69	40	0.775280899	0.597014925
T3	86	62	0.966292135	0.925373134
T4	81	51	0.91011236	0.76119403
T5	83	59	0.93258427	0.880597015
T6	84	58	0.943820225	0.865671642
T7	81	54	0.91011236	0.805970149
T8	80	49	0.898876404	0.731343284
T9	83	54	0.93258427	0.805970149
T10	82	55	0.921348315	0.820895522
T11	84	58	0.943820225	0.865671642
T12	83	57	0.93258427	0.850746269
T13	83	59	0.93258427	0.880597015
T14	75	48	0.842696629	0.71641791
T15	86	62	0.966292135	0.925373134
T16	85	60	0.95505618	0.895522388
T17	82	58	0.921348315	0.865671642
T18	86	63	0.966292135	0.940298507
T19	86	62	0.966292135	0.925373134
T20	82	53	0.921348315	0.791044776

Table 7.3 Precision, Recall and Jaccard Index for Statement Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	5	7	1	5	14	14	
Precision	0.02631	0.03684	0.00526	0.02631	0.07368	0.07368	0.04035
Recall	1	0.875	0.5	0.41666	0.62162	0.20175	0.60250
Jaccard	0.02631	0.03664	0.00523	0.02538	0.29741	0.14902	0.09000

Table 7.4 Precision, Recall and Jaccard Index for Branch Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	5	8	1	5	38	49	
Precision	0.02380	0.03809	0.00476	0.02380	0.18095	0.23333	0.08412
Recall	1	1	0.5	0.41666	0.34234	0.14327	0.56704
Jaccard	0.02380	0.03809	0.00473	0.02304	0.13427	0.09741	0.05356

Table 7.3 shows the precision, recall, and Jaccard index of statement coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.4 shows the precision, recall, and Jaccard index of branch coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.5 shows the precision, recall, and Jaccard index of line coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.5 Precision, Recall and Jaccard Index for Line Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	2	2	1	4	6	38	
Precision	0.0095	0.0095	0.00476	0.01904	0.02857	0.18095	0.04206
Recall	0.4	0.25	0.5	0.33333	0.05405	0.11111	0.2747
Jaccard	0.00938	0.00925	0.00473	0.01834	0.01904	0.07392	0.02245

Table 7.6 Precision, Recall and Jaccard Index for Raw Mutation Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	4	6	2	7	9	29	
Precision	0.02105	0.03157	0.01052	0.03684	0.04736	0.15263	0.05
Recall	0.8	0.75	1	0.58333	0.08108	0.08479	0.54986
Jaccard	0.02094	0.03125	0.01052	0.03589	0.03082	0.05765	0.03118

Table 7.7 Precision, Recall and Jaccard Index for Prorated Mutation Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	4	6	2	7	9	35	
Precision	0.02105	0.03157	0.01052	0.03684	0.04736	0.18421	0.05526
Recall	0.8	0.75	1	0.58333	0.08108	0.10233	0.55279
Jaccard	0.02094	0.03125	0.01052	0.03589	0.03082	0.07042	0.03331

Table 7.6 shows the precision, recall, and Jaccard index of raw mutation coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.7 shows the precision, recall, and Jaccard index of prorated mutation coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.8 shows the precision, recall, and Jaccard index of equivalence-based mutation coverage for the six experiments defined by partial correctness and total correctness with respect to $R1$, $R2$ and $R3$.

Table 7.8 Precision, Recall and Jaccard Index for Equivalence-based Mutation Coverage

Term	TR1	PR1	TR2	PR2	TR3	PR3	Averages
$\|SC\|$	5	8	2	12	111	342	
$\|SC \cap CM\|$	5	8	2	6	26	30	
Precision	0.02631	0.04210	0.01052	0.03157	0.13684	0.15789	0.0675
Recall	1	1	1	0.5	0.23423	0.08771	0.63699
Jaccard	0.02631	0.04210	0.01052	0.03061	0.09454	0.05976	0.04397

Table 7.9 Summary Performance of Coverage Metrics

	Statement	Branch	Line	RMS	PMS	EMS
Precision	0.04035	0.08412	0.04206	0.05000	0.05526	0.0675
Recall	0.60250	0.56704	0.2747	0.54986	0.55279	0.63699
Jaccard Index	0.09000	0.05356	0.02245	0.03118	0.03331	0.04397

7.4 Concluding Remarks

7.4.1 Summary

in this chapter, we present some empirical data stemming from a software testing experiment whose aim is to assess to what extent traditional coverage metrics of test suites reflect the test suite's effectiveness to detect faults. To this effect, we introduce the concept of *detector set*, which is the set of inputs that reveal the failures of an incorrect program, then we define the effectiveness of a test suite by the extent to which the test suite encompasses elements of the detector set, or equivalently, how few elements of the detector set are outside the test suite; we refer to this measure of effectiveness as *semantic coverage*. Taking this definition as a baseline, we consider a number of traditional coverage metrics, and evaluate, on the basis of a sample example, to what extent is the ranking of test suites by means of the metrics coincides with their ranking by semantic coverage.

7.4.2 Observations

The results of our empirical observations are summarized in Table 7.9. We readily recognize that ours is a small-scale experiment, consisting of a simple program, three specifications, and six observations in total. With this qualification in mind, it is fair to say that no coverage metric comes out looking very good:

- The precision of these six metrics varies between 0.04 and 0.07. This means that out of 100 cases where statement coverage, for example, finds that Ti is better than Tj (by virtue of having higher statement coverage) only 4% of the cases are borne out by semantic coverage (i.e., cases where Ti contains more failure-revealing inputs than

Tj). If we had a friend who lied 96% of the time and said the truth only 4% of the time, we would not believe anything he said.

- With the exception of the recall of line coverage, the recall varies between 0.54 and 0.64, which is not very high if we consider that we assigned random numbers to the test suites (in lieu of their coverage metrics) we would get a recall of about 0.5.
- The Jaccard index of two sets is the ratio of elements they have in common (cardinality of their intersection) over the total number of elements that either have (cardinality of their union); it combines precision and recall in a single attribute. The Jaccard index between the coverage metrics and semantic coverage varies between 0.02 and 0.09, clearly low values if we consider that the coverage metrics ought to reflect a test suite's ability to detect faults (by exposing failures).

Covering statements, branches, lines, paths, conditions, etc is not an end in itself; it is a means to exercise the code thoroughly so as to sensitize its faults. The observations made in this study seem to suggest that covering syntactic features does not appear to necessarily lead to revealing program failures. It is noteworthy that much of the low score of precision stems readily from the fact that numeric coverage metrics define a total ordering between test suites since all pairs of numbers can be compared, whereas not all pairs of test suites can be compared for effectiveness; hence, the lack of precision is inherent to numeric metrics.

7.4.3 Prospects

We envision the following directions for further research:

- Further analysis is required to justify using semantic coverage as ground truth in the study of test suite effectiveness or to derive better definitions of ground truth.
- Also, more experiments are needed to assess and compare the various coverage metrics in use nowadays by researchers and practitioners.
- We envision to broaden this study by analyzing other, more sophisticated coverage metrics (e.g., path coverage) for the purposes of this study, we included only coverage metrics for which we found reliable automated tools.

CHAPTER 8

CONCLUSION

In this thesis, I have conducted several empirical studies on software testing, focusing in particular on mutation testing and the analysis of test suite effectiveness, which is the primary justification for mutation testing.

The number of mutants of a base program increases very quickly as a function of the size of the program; hence, using mutation testing in realistic contexts, where programs have tens of thousands of lines of code is prohibitive due to the cost of testing large numbers of mutants. As a result, much research has focused on reducing the number of mutants in a mutation testing experiment while preserving its effectiveness. An important idea that was proposed to reduce the size of mutant sets is the idea of recognizing subsumption relations between mutants and deleting subsumed mutants. In Chapter 2, I argue that the definition of subsumption is incomplete in the sense that it assumes that the base program and all its mutants converge (i.e., terminate normally without attempting any illegal operation) for all the elements of a test suite, but in practice, this is far from true, in fact, many mutation operators are prone to causing mutants to diverge even when the base program converges. Hence, we propose three distinct definitions of mutant subsumption, and I develop Python scripts that analyze the outputs of a mutant to derive its differentiator set with respect to each interpretation of subsumption. Also, using another Python script that highlights inclusion relations between sets, I can generate subsumption graphs of given mutants for each of the three interpretations (delta0, delta1, delta2).

In Chapter 3, I analyze the return on investment in the process of reducing the size of a mutant set by subsumption. Indeed, the most natural way to reduce the size of a set of mutants is to partition the set by means of semantic equivalence, then select a mutant in each equivalence class and drop all the remaining mutants of the same class. Also, whereas subsumption was initially defined as a relation between mutants, we feel that it is more natural to define it as a relation between equivalence classes of mutants, modulo semantic equivalence. This raises the question: once we have reduced a mutant set by equivalence, how much more can subsumption reduce it? and is the extra effort and risk commensurate with the additional reduction?

Whether it is done by equivalence or by subsumption, the problem of reducing the size of a mutant set is essentially an optimization problem where the objective function is the size of the set and the constraint under which this optimization is attempted to preserve the effectiveness of the mutant set. This raises the question: how do we quantify the effectiveness of a mutant set? One simple way to answer this question is to argue that the effectiveness of a mutant set can be judged by the effectiveness of the test suites that the mutant set vets, where a mutant set vets a test suite if and only if the test suite kills all the mutants of the set. This, in turn, raises the question of how we define the effectiveness (in Chapter 4) of a test suite? We define the effectiveness of a test suite by assessing to what extent a test suite reveals (or does not reveal) all the failures of a program: we call this the *semantic coverage* of the test suite, which we use to define two metrics of mutant set effectiveness (in Chapter 5): the *assured effectiveness* and the *potential effectiveness*, which are, respectively, the smallest and the largest semantic coverage of all the test suites that the mutant set vets. We show that removing a subsumed mutant from a mutant set does not reduce its potential effectiveness.

Whereas there are several ways to assess the effectiveness of a test suite, mutation coverage is often viewed as the best measure of test suite effectiveness, and is often used as a basis for judging the quality of other measures of coverage (statement coverage, branch coverage, condition coverage, path coverage, etc). It is common, in the software testing literature, to see references to *mutation coverage*, as if this measure were uniform across mutant generation tools and mutant generation policies (even for the same tool). In Chapter 6, we run an experiment in which we compute the mutation coverage of a set of test suites for several mutant generation tools, and several mutant generation policies, and we show that the results obtained for the same test suite vary significantly depending on the mutant generation operators that are used. This raises the question: If a test suite gets a mutation score of 0.2 in one mutation testing experiment and a score of 0.80 in another, what can we infer about it? More broadly, it means that we cannot compare the results of mutation testing experiments unless they are using the same mutant generator tools, and deploying the same mutant generation operators.

Many of the coverage metrics used traditionally to assess the effectiveness of a test suite is based on the assumption that a good test suite is one that exercises syntactic features of the program under test. But in fact, exercising all the syntactic features of a program is neither necessary nor sufficient to reveal all of the program's failures. It is not sufficient because the same fault may be sensitized for some inputs but not for others, and it is not necessary because only faulty statements need to be exercised. In Chapter 7, we run an experiment in which we assess the extent to which traditional metrics are correlated with a test suite's ability to reveal failures; this experiment enables us to analyze how traditional coverage metrics may indicate (or fail to indicate) whether a test suite is adequate to reveal the failures of an incorrect programs. Most well-known metrics prove to be very inefficient, at least as far as our empirical study is concerned.

Prospects for further research include revisiting all the empirical studies conducted in this thesis to broaden their scope and refine their conclusions. In particular, we are interested in further elucidating the relationship between mutation coverage (as a proxy for the ability of a test suite to detect faults) and semantic coverage (as a proxy for the ability of a test suite to reveal failures). This study would enable us to answer the question: if a test suite T is better than a test suite T' at detecting faults, is it also, better at revealing failures?

APPENDIX A

APPENDICES: PYTHON SCRIPTS COMPOSED FOR THIS RESEARCH

In this appendix, I presented all the scripts that had been written in Python to analyze the data from different benchmarks. All these scripts are available on[1];

A.1 Run Test On Mutants

```python
import os
import subprocess
import shutil
mutants_dir='mutants'
target='src/main/java/org/apache/commons/lang3/math/NumberUtils.java'
for filename in os.listdir(mutants_dir):
    source = os.path.join(mutants_dir, filename)
    mutant_name=filename.split('.java')[0]



 if os.path.isfile(source):
     print(source)
     shutil.copyfile(source, target)
    subprocess.run(['mvn','clean','compile','test', '-Drat.
   numUnapprovedLicenses=100'], capture_output=True, shell=True, text=True
   )
```

[1]https://github.com/SAMIA-CLOUD/EMPIRICAL-EXPLORATION-OF-SOFTWARE-TESTING/

```
    print(filename)
     os.rename('test_results/n0.txt', 'test_results/M'+mutant_name+'.txt')
```

Listing A.1 Run Test On Mutants.

A.2 Extract Equivalence Classes

```
def loadMutant(filename):
  file=open('test results/'+filename+'.txt');
  lines=file.readlines();
  data={};
  for line in lines:
    line=line.strip().split(",");
    test=line[0].strip();
    output=line[1].strip();
    data[test]=output;
  count=len(data);
  return data;

def isSame(res_a,res_b):
  for t in res_a:
    if res_a[t]!=res_b[t]:
      return False;
  return True;

```

```
base_results=loadMutant('Base'); #load the results of the base program

mutants=['m'+str(i) for i in range(1,134)]; #list of all mutants
results={};
for m in mutants:
  results[m]=loadMutant(m); #load the results of each mutant (test number:
    test outcome)
equivalent_mutants={};
minimal_set=[]; #minimal set is initially empty
minimal_set_print=[]
for m_a in mutants: #for each mutant m_a
  to_add_m_a=True;
  for m_b in minimal_set: #for each mutant in minimal set m_b
    if m_b==m_a:
      continue;
    if isSame(results[m_a],results[m_b]): #if m_a and m_b are equivalent,
    don't add m_a to minimal set and add it to the equivalence class of m_b
      to_add_m_a=False;
      equivalent_mutants[m_b].append(m_a);
      break;
  if to_add_m_a==True: #if m_a is not equivalent to any mutants in m_b, add
     it to the minimal set.
    minimal_set.append(m_a);
    minimal_set_print.append("'"+m_a+"'");
    equivalent_mutants[m_a]=[];

writer=open("analysis_equivalence.txt","w");
writer.write("minimal set by equivalence:\n");
writer.write(", ".join(minimal_set_print)+"\n\n\n");
writer.write("Equivalent Classes:\n");
count=1;
for m in equivalent_mutants:
  writer.write("Class "+str(count)+": "+m+", "+", ".join(equivalent_mutants
    [m])+"\n");
  count=count+1;
writer.close();
writer = open("number of mutants per class.txt","w")
for m in equivalent_mutants:
    writer.write(m+","+str(1+len(equivalent_mutants[m]))+"\n");
writer.close();
```

Listing A.2 Extract Equivalence Set Classes.

```
def loadMutant(filename):
    file=open('test results/'+filename+'.txt');
    lines=file.readlines();
    data={};
    for line in lines:
        line=line.strip().split(",");
        if line[0]=='':
            continue
        test=line[0].strip();
        output=line[1].strip();
        data[test]=output;
    count=len(data);
    return data;

def MutantsPerClass():
    file=open('number of mutants per class.txt');
    number = {}
    lines=file.readlines();
    for line in lines:
        line=line.strip().split(",");
        number[line[0]]=int(line[1])
    return number

def isSame(res_a,res_b):
    for t in res_a:
        if res_a[t]!=res_b[t]:
            return False;
    return True;

def isError(test_output):
    if ("java.lang" in test_output) or ("error" in test_output):
        return True
    return False
base_results=loadMutant('Base'); #load the results of the base program


mutants=['m1', 'm7', 'm10', 'm12', 'm13', 'm14', 'm16', 'm17', 'm19', 'm20'
    , 'm23', 'm24', 'm25', 'm26', 'm28', 'm29', 'm31', 'm35', 'm36', 'm37',
    'm39', 'm40', 'm42', 'm43', 'm44', 'm45', 'm48', 'm49', 'm50', 'm51', '
    m53', 'm55', 'm56', 'm57', 'm58', 'm59', 'm67', 'm102', 'm113', 'm114',
    'm115', 'm118', 'm126', 'm127', 'm129', 'm132']

results={};
for m in mutants:
    results[m]=loadMutant(m); #load the results of each mutant (test number
    : test outcome)

number = MutantsPerClass()
d1={};
d2={};
d0={};
for m in mutants:
    d1[m]=[];
    d2[m]=[];
    d0[m]=[];
    for t in results[m]:
        #for d0, we assume error is an output and we compare strings
        print(m,t)
        if results[m][t]!=base_results[t]:
            d0[m].append(t)
        if not(isError(results[m][t]) and isError(base_results[t])):
            if not isError(results[m][t]) and  not isError(base_results[t])
    :
                if results[m][t]!=base_results[t]:
                    d1[m].append(t);
                    d2[m].append(t);
            else:
                    d2[m].append(t);

writer=open("delta_analysis.csv",'w');
writer.write("Mutants,Delta 0\n");
for m in mutants:
    if len(d0[m])==0:
        d0[m]=["Empty"];
    if len(d1[m])==0:
        d1[m]=["Empty"];
    if len(d2[m])==0:
        d2[m]=["Empty"];
    tmp=[m,";".join(d0[m]),"\n"];
    writer.write(",".join(tmp));
writer.close();
```

Listing A.3 Get Delta.

A.3 Compute Killed Mutants by Each Test

```
def MutantsPerClass():
    file=open('number of mutants per class.txt');
    number = {}
    lines=file.readlines();
    for line in lines:
        line=line.strip().split(",");
        number[line[0]]=int(line[1])
    return number
number = MutantsPerClass();
dictionary={};
delta=0
file=open('delta_analysis.csv');
lines=file.readlines();
for i in range(1,len(lines)):
    line=lines[i].strip().split(",");
    mutant=int(line[0][1:])
    tests=line[delta+1].strip().split(';');
    if 'Empty' in tests:
        print(mutant)
    for t in tests:
        if t not in dictionary:
            dictionary[t]=[];
        dictionary[t].append(mutant)

results={};
file=open('Subsets.txt');
lines=file.readlines();
killed_mutants={}
for line in lines:
    line=line.strip().split(":");
    t_name=line[0];
    tests=line[1].split(',');
    results[t_name]=set();
    for t in tests:
        if t not in dictionary:
            continue;
        mutants_killed=dictionary[t];
        for m in mutants_killed:
            results[t_name].add(m)
    temp=sorted(results[t_name])
    temp = ['m'+str(j) for j in temp]
    killed_mutants[t_name]=str(sum([number[i] for i in temp]))
    results[t_name] = temp;

writer = open('mutants_by_Ti'+str(delta)+'.csv','w')
writer.write("Test Suite Subset, Number of Killed Equiv,Number of killed
    mutants, Minimal Mutants\n")
for Ti in results:
    writer.write(Ti+","+str(len(results[Ti]))+","+killed_mutants[Ti]+","+";
    ".join(results[Ti])+"\n")
writer.close()
```

Listing A.4 Killed Mutants by Each Test.

A.4 Extract Operators From Mutants

```
file = open('vt100TerminalMutantsOperators.txt')
lines = file.readlines()
operators={}

for line in lines:
    line=line.strip().split(',')
    if line[1] not in operators:
        operators[line[1]] = []
    operators[line[1]].append('M'+line[0])
all_writer=open('ALL.txt','w')
```

```
for op in operators:
    writer=open(op+'.txt','w')
    writer.write("\n".join(operators[op]))
    all_writer.write("\n".join(operators[op]))
    writer.close()
all_writer.close()
```

Listing A.5 Get Operators From Mutants.

A.5 Generate Test Sets

```
import random
threshold = 0.2
for i in range(1,21):
   #generate ti
    print("------------------------------------")
    threshold = threshold + 0.03
    tests_in_ti = []
    for j in range(1,108)://maybe 137
        if random.random()<=threshold:
            tests_in_ti.append(str(j))
    print(f"test suite t{i} ({len(tests_in_ti)}):");
    print(",".join(tests_in_ti))
```

Listing A.6 Generate Test Sets.

A.6 Compute Similarirty

```
def loadEdges(filename):
    file=open(filename+'_graph_Ti0.txt')
    lines=file.readlines();
    data=[];
    for l in lines:
        data.append(l.strip())
    return data;
operators=['RelationalOperatorReplacement','ANDOperatorReplacement','
   OROperatorReplacement']
writer=open("results.txt","w")
for i in range(0,len(operators)-1):
    for j in range(i+1,len(operators)):
        o1=operators[i]
        o2=operators[j]
        g1 = loadEdges(o1);
        g2 = loadEdges(o2);
        writer.write("Measure between " +o1+" and "+o2+" \n")
        union = set(g1+g2)
        intersection = [value for value in g2 if value in g1]
        writer.write("Cardinality of union: "+str(len(union))+"\n")
        writer.write("Cardinality of intersection: "+str(len(intersection))
   +"\n")
```

```
        writer.write("Similarity Measure: "+str(len(intersection)/len(union
    ))+"\n\n\n\n")
writer.close()
```

Listing A.7 Get Similarity.

A.7 Generate Random Test Subset

```
'''
Rules: We need 20 subsets T1 to T20
T1-T5: 32 random
T6-T10: 27 random
T11-15: 22 random
T16-T20: 36 random
'''

import random
T=[i for i in range(1,38)]

subsets=[]
#T1 to T5
t=0;
while t<5:
    temp=list(T)
    for r in range(0,5):
        element_to_remove = random.choice(temp);
        temp.remove(element_to_remove)
    if temp not in subsets:
        subsets.append(temp)
        t+=1;
#T6-T10
t=0;
while t<5:
    temp=list(T)
    for r in range(0,10):
        element_to_remove = random.choice(temp);
        temp.remove(element_to_remove)
    if temp not in subsets:
        subsets.append(temp)
        t+=1;
#T11-T15
t=0;
while t<5:
    temp=list(T)
    for r in range(0,15):
        element_to_remove = random.choice(temp);
        temp.remove(element_to_remove)
    if temp not in subsets:
        subsets.append(temp)
        t+=1;
#T16-T20
t=0;
while t<5:
    temp=list(T)
    for r in range(0,1):
        element_to_remove = random.choice(temp);
        temp.remove(element_to_remove)
    if temp not in subsets:
        subsets.append(temp)
        t+=1;
writer=open("Subsets.txt","w")
for i in range(0,len(subsets)):
    name='T'+str(i+1);
    l=[str(j) for j in subsets[i]]
    writer.write(name+":"+",".join(l)+"\n")
writer.close()
```

Listing A.8 Generate Random Test Subset.

A.8 Extract Inclusion By Test Based On Delta Zero

```
delta=0
#read and process mutants killed by tests

```

```

def loadTests():
    data={}
    file = open('Subsets.txt')
    lines=file.readlines();
    for i in range(0,len(lines)):
        line=lines[i].strip().split(":");
        test_suite=line[0]
        tests=line[1].split(',');
        data[test_suite]=tests;
    return data;
data= loadTests();


inclusions=[];
for t1 in data:
    for t2 in data:
        if t1==t2:
            continue;
        if(all(x in data[t2] for x in data[t1])):
            inclusions.append('"'+t1+'" -> "'+t2+'"');



```

```
writer = open("graph_inclusions_tests.txt",'w');
writer.write("\n".join(inclusions));
writer.close();
```

Listing A.9 Extract Inclusion By Test.

A.9 Extract Inclusion By Mutants

```

delta=1

#read and process mutants killed by tests

def loadTests():
    data={}
    file = open('mutants_by_Ti'+str(delta)+'.csv')
    lines=file.readlines();
    for i in range(1,len(lines)):
        line=lines[i].strip().split(",");
        test_name=line[0]
        mutants=line[2].split(';');
        data[test_name]=mutants;
    return data;

tests = loadTests();
```

```

inclusions=[];
for t1 in tests:
    for t2 in tests:
        if t1==t2:
            continue;
        if(all(x in tests[t2] for x in tests[t1])):
            inclusions.append('"'+t1+'" -> "'+t2+'"');

writer = open("graph_"+str(delta)+".txt",'w');
writer.write("\n".join(inclusions));
writer.close();
```

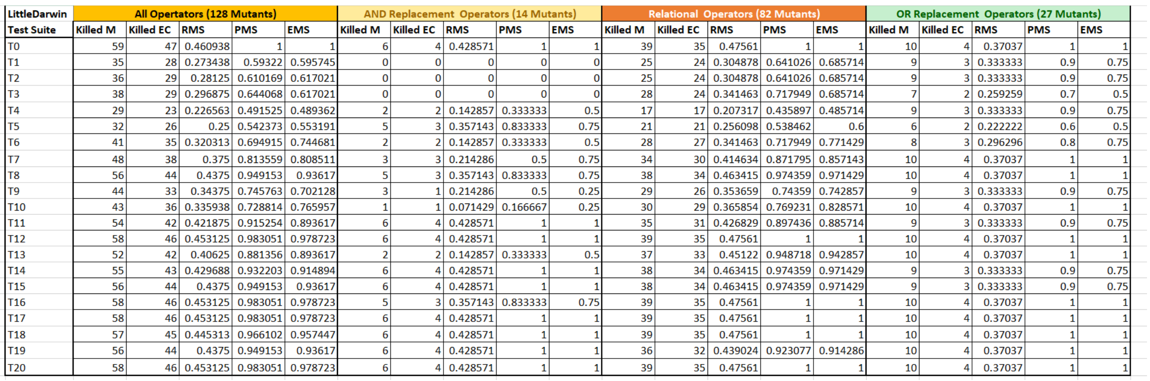
Listing A.10 Extract Inclusion By Mutants.

APPENDIX B

APPENDICES: RESULTS COLLECTED IN EXCELS FILE

In this appendix, I presented some of the raw data in Excel obtained using Script Extract Operators From Mutants, which was used in the previous appendix A.4. The benchmark is Jterminal class VT100.

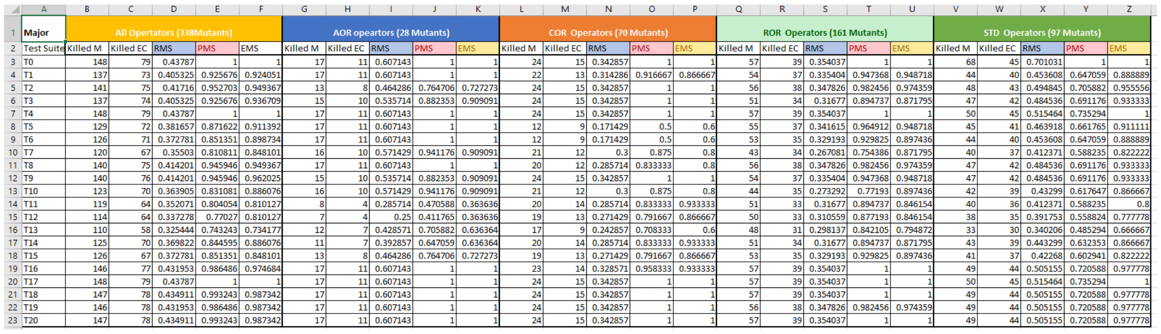
B.1 Extract Operators From VT100 Jterminal by Using LittleDarwine Tool



LittleDarwin	All Opertators (128 Mutants)					AND Replacement Operators (14 Mutants)					Relational Operators (82 Mutants)					OR Replacement Operators (27 Mutants)				
Test Suite	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS
T0	59	47	0.460938	1	1	6	4	0.428571	1	1	39	35	0.47561	1	1	10	4	0.37037	1	1
T1	35	28	0.273438	0.59322	0.595745	0	0	0	0	0	25	24	0.304878	0.641026	0.685714	9	3	0.333333	0.9	0.75
T2	36	29	0.28125	0.610169	0.617021	0	0	0	0	0	25	24	0.304878	0.641026	0.685714	9	3	0.333333	0.9	0.75
T3	38	29	0.296875	0.644068	0.617021	0	0	0	0	0	28	24	0.341463	0.717949	0.685714	7	2	0.259259	0.7	0.5
T4	29	23	0.226563	0.491525	0.489362	2	2	0.142857	0.333333	0.5	17	17	0.207317	0.435897	0.485714	9	3	0.333333	0.9	0.75
T5	32	26	0.25	0.542373	0.553191	5	3	0.357143	0.833333	0.75	21	21	0.256098	0.538462	0.6	6	2	0.222222	0.6	0.5
T6	41	35	0.320313	0.694915	0.744681	2	2	0.142857	0.333333	0.5	28	27	0.341463	0.717949	0.771429	8	3	0.296296	0.8	0.75
T7	48	38	0.375	0.813559	0.808511	3	3	0.214286	0.5	0.75	34	30	0.414634	0.871795	0.857143	10	4	0.37037	1	1
T8	56	44	0.4375	0.949153	0.93617	5	3	0.357143	0.833333	0.75	38	34	0.463415	0.974359	0.971429	10	4	0.37037	1	1
T9	44	33	0.34375	0.745763	0.702128	3	1	0.214286	0.5	0.25	29	26	0.353659	0.74359	0.742857	9	3	0.333333	0.9	0.75
T10	43	36	0.335938	0.728814	0.765957	1	1	0.071429	0.166667	0.25	30	29	0.365854	0.769231	0.828571	10	4	0.37037	1	1
T11	54	42	0.421875	0.915254	0.893617	6	4	0.428571	1	1	35	31	0.426829	0.897436	0.885714	9	3	0.333333	0.9	0.75
T12	58	46	0.453125	0.983051	0.978723	6	4	0.428571	1	1	39	35	0.47561	1	1	10	4	0.37037	1	1
T13	52	42	0.40625	0.881356	0.893617	2	2	0.142857	0.333333	0.5	37	33	0.45122	0.948718	0.942857	10	4	0.37037	1	1
T14	55	43	0.429688	0.932203	0.914894	6	4	0.428571	1	1	38	34	0.463415	0.974359	0.971429	9	3	0.333333	0.9	0.75
T15	56	44	0.4375	0.949153	0.93617	6	4	0.428571	1	1	38	34	0.463415	0.974359	0.971429	9	3	0.333333	0.9	0.75
T16	58	46	0.453125	0.983051	0.978723	5	3	0.357143	0.833333	0.75	39	35	0.47561	1	1	10	4	0.37037	1	1
T17	58	46	0.453125	0.983051	0.978723	6	4	0.428571	1	1	39	35	0.47561	1	1	10	4	0.37037	1	1
T18	57	45	0.445313	0.966102	0.957447	6	4	0.428571	1	1	39	35	0.47561	1	1	10	4	0.37037	1	1
T19	56	44	0.4375	0.949153	0.93617	6	4	0.428571	1	1	36	32	0.439024	0.923077	0.914286	10	4	0.37037	1	1
T20	58	46	0.453125	0.983051	0.978723	6	4	0.428571	1	1	39	35	0.47561	1	1	10	4	0.37037	1	1

Figure B.1 Extract operators from VT100 Jterminal by using littleDarwine tool.

B.2 Extract Operators From VT100 Jterminal by Using Major Tool

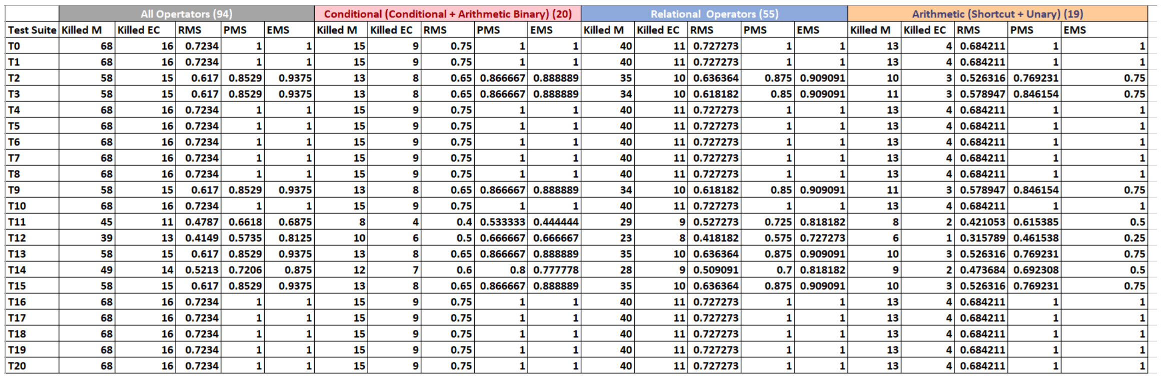


	A	B	C	D	E	F	G	H	I	J	K
1	Major	All Opertators (338Mutants)					AOR opeartors (28 Mutants)				
2	Test Suite	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS
3	T0	148	79	0.43787	1	1	17	11	0.607143	1	1
4	T1	137	73	0.405325	0.925676	0.924051	17	11	0.607143	1	1
5	T2	141	75	0.41716	0.952703	0.949367	13	8	0.464286	0.764706	0.727273
6	T3	137	74	0.405325	0.925676	0.936709	15	10	0.535714	0.882353	0.909091
7	T4	148	79	0.43787	1	1	17	11	0.607143	1	1
8	T5	129	72	0.381657	0.871622	0.911392	17	11	0.607143	1	1
9	T6	126	71	0.372781	0.851351	0.898734	17	11	0.607143	1	1
10	T7	120	67	0.35503	0.810811	0.848101	16	10	0.571429	0.941176	0.909091
11	T8	140	75	0.414201	0.945946	0.949367	17	11	0.607143	1	1
12	T9	140	76	0.414201	0.945946	0.962025	15	10	0.535714	0.882353	0.909091
13	T10	123	70	0.363905	0.831081	0.886076	16	10	0.571429	0.941176	0.909091
14	T11	119	64	0.352071	0.804054	0.810127	8	4	0.285714	0.470588	0.363636
15	T12	114	64	0.337278	0.77027	0.810127	7	4	0.25	0.411765	0.363636
16	T13	110	58	0.325444	0.743243	0.734177	12	7	0.428571	0.705882	0.636364
17	T14	125	70	0.369822	0.844595	0.886076	11	7	0.392857	0.647059	0.636364
18	T15	126	67	0.372781	0.851351	0.848101	13	8	0.464286	0.764706	0.727273
19	T16	146	77	0.431953	0.986486	0.974684	17	11	0.607143	1	1
20	T17	148	79	0.43787	1	1	17	11	0.607143	1	1
21	T18	147	78	0.434911	0.993243	0.987342	17	11	0.607143	1	1
22	T19	146	78	0.431953	0.986486	0.987342	17	11	0.607143	1	1
23	T20	147	78	0.434911	0.993243	0.987342	17	11	0.607143	1	1

	A	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	Major	COR Operators (70 Mutants)					ROR Operators (161 Mutants)					STD Operators (97 Mutants)				
2	Test Suite	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS
3	T0	24	15	0.342857	1	1	57	39	0.354037	1	1	68	45	0.701031	1	1
4	T1	22	13	0.314286	0.916667	0.866667	54	37	0.335404	0.947368	0.948718	44	40	0.453608	0.647059	0.888889
5	T2	24	15	0.342857	1	1	56	38	0.347826	0.982456	0.974359	48	43	0.494845	0.705882	0.955556
6	T3	24	15	0.342857	1	1	51	34	0.31677	0.894737	0.871795	47	42	0.484536	0.691176	0.933333
7	T4	24	15	0.342857	1	1	57	39	0.354037	1	1	50	45	0.515464	0.735294	1
8	T5	12	9	0.171429	0.5	0.6	55	37	0.341615	0.964912	0.948718	45	41	0.463918	0.661765	0.911111
9	T6	12	9	0.171429	0.5	0.6	53	35	0.329193	0.929825	0.897436	44	40	0.453608	0.647059	0.888889
10	T7	21	12	0.3	0.875	0.8	43	34	0.267081	0.754386	0.871795	40	37	0.412371	0.588235	0.822222
11	T8	20	12	0.285714	0.833333	0.8	56	38	0.347826	0.982456	0.974359	47	42	0.484536	0.691176	0.933333
12	T9	24	15	0.342857	1	1	54	37	0.335404	0.947368	0.948718	47	42	0.484536	0.691176	0.933333
13	T10	21	12	0.3	0.875	0.8	44	35	0.273292	0.77193	0.897436	42	39	0.43299	0.617647	0.866667
14	T11	20	14	0.285714	0.833333	0.933333	51	33	0.31677	0.894737	0.846154	40	36	0.412371	0.588235	0.8
15	T12	19	13	0.271429	0.791667	0.866667	50	33	0.310559	0.877193	0.846154	38	35	0.391753	0.558824	0.777778
16	T13	17	9	0.242857	0.708333	0.6	48	31	0.298137	0.842105	0.794872	33	30	0.340206	0.485294	0.666667
17	T14	20	14	0.285714	0.833333	0.933333	51	34	0.31677	0.894737	0.871795	43	39	0.443299	0.632353	0.866667
18	T15	19	13	0.271429	0.791667	0.866667	53	35	0.329193	0.929825	0.897436	41	37	0.42268	0.602941	0.822222
19	T16	23	14	0.328571	0.958333	0.933333	57	39	0.354037	1	1	49	44	0.505155	0.720588	0.977778
20	T17	24	15	0.342857	1	1	57	39	0.354037	1	1	50	45	0.515464	0.735294	1
21	T18	24	15	0.342857	1	1	57	39	0.354037	1	1	49	44	0.505155	0.720588	0.977778
22	T19	24	15	0.342857	1	1	56	38	0.347826	0.982456	0.974359	49	44	0.505155	0.720588	0.977778
23	T20	24	15	0.342857	1	1	57	39	0.354037	1	1	49	44	0.505155	0.720588	0.977778

Figure B.2 Extract operators from VT100 Jterminal by using major tool.

B.3 Extract Different Operators From VT100 Jterminal by Using LittleDarwine Tool



	All Opertators (94)					Conditional (Conditional + Arithmetic Binary) (20)					Relational Operators (55)					Arithmetic (Shortcut + Unary) (19)				
Test Suite	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS	Killed M	Killed EC	RMS	PMS	EMS
T0	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T1	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T2	58	15	0.617	0.8529	0.9375	13	8	0.65	0.866667	0.888889	35	10	0.636364	0.875	0.909091	10	3	0.526316	0.769231	0.75
T3	58	15	0.617	0.8529	0.9375	13	8	0.65	0.866667	0.888889	34	10	0.618182	0.85	0.909091	11	3	0.578947	0.846154	0.75
T4	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T5	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T6	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T7	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T8	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T9	58	15	0.617	0.8529	0.9375	13	8	0.65	0.866667	0.888889	34	10	0.618182	0.85	0.909091	11	3	0.578947	0.846154	0.75
T10	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T11	45	11	0.4787	0.6618	0.6875	8	4	0.4	0.533333	0.444444	29	9	0.527273	0.725	0.818182	8	2	0.421053	0.615385	0.5
T12	39	13	0.4149	0.5735	0.8125	10	6	0.5	0.666667	0.666667	23	8	0.418182	0.575	0.727273	6	1	0.315789	0.461538	0.25
T13	58	15	0.617	0.8529	0.9375	13	8	0.65	0.866667	0.888889	35	10	0.636364	0.875	0.909091	10	3	0.526316	0.769231	0.75
T14	49	14	0.5213	0.7206	0.875	12	7	0.6	0.8	0.777778	28	9	0.509091	0.7	0.818182	9	2	0.473684	0.692308	0.5
T15	58	15	0.617	0.8529	0.9375	13	8	0.65	0.866667	0.888889	35	10	0.636364	0.875	0.909091	10	3	0.526316	0.769231	0.75
T16	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T17	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T18	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T19	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1
T20	68	16	0.7234	1	1	15	9	0.75	1	1	40	11	0.727273	1	1	13	4	0.684211	1	1

Figure B.3 All, conditional, relational, and arithmetic operators.

REFERENCES

[1] A. Mili and F. Tchier, *Software Testing: Operations and Concepts.* Hoboken, NJ: John Wiley and Sons, 2015.

[2] R. A. DeMillo, 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, April 1978.

[3] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, “On fault representativeness of software fault injection,” *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[4] A. Namin, J. Andrews, and D. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings, ICSE 2008*, 2008, pp. 351–360.

[5] J. Andrews, L. Briand, Y. Labiche, and A. Siami Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *Software Engineering, IEEE Transactions on*, vol. 32, pp. 608–624, 09 2006.

[6] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[7] R. Just, D. Jalali, L. Inozemtseva, M. 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 Symposium on Foundations of Software Engineering(FSE)*, 2014.

[8] J. Andrews, L. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” 2005, pp. 402–411.

[9] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” 2014.

[10] A. Parsai, A. Murgia, and S. Demeyer, “Littledarwin: A feature-rich and extensible mutation testing framework for large and complex java systems,” in *FSEN 2017, Foundations of Software Engineering*, 2016.

[11] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, “Proteum /im 2.0: An integrated mutation testing environment,” in *Mutation Testing for the New Century*, W. E. Wong, Ed. Springer Verlag, 2001, vol. 24, pp. 91–101.

[12] H. Coles, “Real world mutation testing,” 2017. [Online]. Available: https://pitest.org/

[13] Y. S. Ma and J. Offutt, “Mu java,” George Mason University, http://cs.gmu.edu/ offutt/-mujava/, Tech. Rep., 2020.

[14] I. Marsit, A. Ayad, D. Kim, M. Latif, J. Loh, M. N. Omri, and A. Mili, “The ratio of equivalent mutants: A key to analyzing mutation equivalence,” *Journal of Systems and Software*, July 2021.

[15] B. Kurtz, P. Amman, M. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Proceedings, 7th International Conference on Software Testing, Validation and Verification Workshops*, 2014.

[16] B. Kurtz, P. Ammann, and J. Offutt, "Static analysis of mutant subsumption," in *Proceedings, IEEE 8th International Conference on Software Testing, Verification and Validation Workshops*, 2015.

[17] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, January 2019.

[18] S. AlBlwi, I. Marsit, B. Khaireddine, A. Ayad, J. Loh, and A. Mili, "Generalized mutant subsumption," in *Proceedings, ICSOFT 2022*, Lisbon, Portugal, July 2022.

[19] S. Alblwi and A. Ayad, "Minimizing mutant sets by equivalence and subsumption," *International Journal of Computer and Systems Engineering*, vol. 18, no. 1, pp. 21 – 27, 2024. [Online]. Available: https://publications.waset.org/vol/205

[20] S. Alblwi, A. Ayad, and A. Milil, "Semantic coverage: Measuring test suite effectiveness," in *Proceedings, International Conference on Software Technology*, Rome, Italy, July 2023.

[21] B. K. I. M. Samia Alblwi, Amani Ayad and A. Mili, "Quantifying the effectiveness of mutant sets," in *In 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022.

[22] A. A. Samia Alblwi and A. Mili, "Mutation coverage is not strongly correlated with mutation coverage," in *The 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 2024.

[23] E. Hehner, *A Practical Theory of Programming.* Heidelberg, Germany: Springer-Verlag, 1993.

[24] C. C. Morgan, *Programming from Specifications, Second Edition*, ser. International Series in Computer Sciences. London, UK: Prentice Hall, 1998.

[25] D. Gries, *The Science of Programming.* Heidelberg, Germany: Springer Verlag, 1981.

[26] E. Dijkstra, *A Discipline of Programming.* Saddle River,NJ: Prentice Hall, 1976.

[27] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.

[28] Z. Manna, *A Mathematical Theory of Computation.* New York, NY: McGraw-Hill, 1974.

[29] B. Souza, "Identifying mutation subsumption relations," in *Proceedings, IEEE / ACM International Conference on Automated Software Engineering*, December 2020, pp. 1388–1390.

[30] A. Parsai and S. Demeyer, "Dynamic mutant subsumption analysis using littledarwin," in *Proceedings, A-TEST 2017*, Paderborn, Germany, September 4-5 2017.

[31] X. Li, Y. Wang, and H. Lin, "Coverage based dynamic mutant subsumption graph," in *Proceedings, International Conference on Mathematics, Modeling and Simulation Technologies and Applications*, 2017.

[32] M. C. Tenorio, R. V. V. Lopes, J. Fechina, T. Marinho, and E. Costa, "Subsumption in mutation testing: An automated model based on genetic algorithm," in *Proceedings, 16th International Conference on Information Technology –New Generations.* Springer Verlag, 2019.

[33] M. A. Guimaraes, L. Fernandes, M. Riberio, M. d'Amorim, and R. Gheyi, "Optimizing mutation testing by discovering dynamic mutant subsumption relations," in *Proceedings, 13th International Conference on Software Testing, Validation and Verification*, 2020.

[34] A. Mili, M. Frias, and A. Jaoua, "On faults and faulty programs," in *Proceedings, RAMICS 2014*, ser. LNCS, P. Hoefner, P. Jipsen, W. Kahl, and M. E. Mueller, Eds., vol. 8428, 2014, pp. 191–207.

[35] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings, ICSE*, 2005.

[36] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proceedings, ISSTA*, 2011.

[37] A. Mili, "Differentiators and detectors," *Information Processing Letters*, vol. 169, 2021.

[38] B. K. I. M. J. M. L. Samia AlBlwi, Amani Ayad and A. Mili, "Subsumption, correctness and relative correctness." Social Science Research Network(SSRN), 2023. [Online]. Available: https://ssrn.com/abstract=4598614

[39] B. Khaireddine, M. Martinez, and A. Mili, "Program repair at arbitrary fault depth," in *Proceedings, ICST 2019*, Xi'An, China, April 2019.

[40] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proceedings, Eighth IEEE International Working Conference on Software Code Analysis and Manipulation*, Beijing, China, September 2008, pp. 249–258.

[41] H. D. Mills, V. R. Basili, J. D. Gannon, and D. R. Hamlet, *Structured Programming: A Mathematical Approach.* Boston, MA: Allyn and Bacon, 1986.

[42] E. Brinksma, M. Stoelinga, and L. B. Briones, "A semantic framework for test coverage," in *Automated Technology for Verification and Analysis(ATVA 2006).* Berlin, Heidelberg: Springer, 2006, p. 399–414.

[43] A. Ayad, I. Marsit, J. Loh, M. N. Omri, and A. Mili, "Estimating the number of equivalent mutants," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Xi'An, China, 2019, pp. 112–121.

[44] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *ICSE 2014 Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, 2014, p. 919–930.

[45] A. P. Mathur, *Foundations of Software Testing.* Saddle River, NJ: Pearson, 2014.

[46] A. Avizienis, J. C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[47] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings, 36th International Conference on Software Engineering.* New York, NY: ACM Press, 2014, p. 435–445.

[48] N. Diallo, W. Ghardallou, and A. Mili, "Correctness and relative correctness," in *Proceedings, 37th International Conference on Software Engineering, NIER track*, Firenze, Italy, May 20–22 2015.

[49] B. Khaireddine, M. Martinez, and A. Mili, "Program repair at arbitrary fault depth," in *Proceedings, ICST 2019 Tools Track*, Xi'An, China, April 2019.

[50] J. V. Wright, "A lattice theoretical basis for program refinement," Dept. of Computer Science, Åbo Akademi, Finland, Tech. Rep., 1990.

[51] B. Aichernig, E. Jobstl, and M. Kegele, "Incremental refinement checking for test case generation," in *Tests and Proofs*, 2013, pp. 1–19.

[52] R. Banach and M. Poppleton, "Retrenchment, refinement and simulation," in *ZB: Formal Specifications and Development in Z and B*, ser. Lecture Notes in Computer Science. Springer, December 2000, pp. 304–323.

[53] D. Shin, S. Yoo, and D.-H. Bae, "A theoretical and empirical study of diversity-aware mutation adequancy criterion," *IEEE TSE*, vol. 44, no. 10, October 2018.

[54] M. R. Lyu, J. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," *IEEE transactions on reliability*, vol. 43, no. 4, pp. 527–535, 1994.

[55] R. Lingampally, A. Gupta, and P. Jalote, "A multipurpose code coverage tool for java," in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07).* IEEE, 2007, pp. 261b–261b.

[56] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security.* IEEE, 2015, pp. 151–156.

[57] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–33, 2015.

[58] K. E. Someoliayi, S. Jalali, M. Mahdieh, and S.-H. Mirian-Hosseinabadi, "Program state coverage: a test coverage metric based on executed program states," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2019, pp. 584–588.

[59] T. Ball, "A theory of predicate-complete test coverage and generation," in *International Symposium on Formal Methods for Components and Objects.* Berlin, Heidelberg: Springer, 2004, pp. 1–22.

[60] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "Measuring effectiveness of mutant sets," in *Proceedings, Ninth International Conference on Software Testing*, Chicago, IL, April 11-15 2016.

[61] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *Proceedints, Ninth International Conference on Software Testing*, Chicago, IL, April 11-15 2016.

[62] S. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing mutants to guide mutation testing," in *Proceedings, ICSE 2022*, Pittsburgh, PA, May 2022.

[63] M. Papadakis, T. T. Chekam, and Y. L. traon, "Mutant quality indicator," in *IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018.

[64] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 1–24, 08 2020.

[65] P. Frankl, S. Weiss, and C. Hu, "All-uses versus mutation testing: An experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, 08 2000.

[66] K. Aaltonen, P. Ihantola, and O. Seppala, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," in *Companion to the 25th Annual ACM SIGPLAN Conference on OOPSLA*, Reno, NV, 10 2010, pp. 153–160.

[67] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, 2009, pp. 220 – 229.

[68] D. Tengeri, L. Vidacs, A. Beszedes, J. Jasz, G. Balogh, B. Vancsics, and T. Gyimothy, "Relating code coverage, mutation score and test suite reducibility to defect density," in *Proceedings, 2016 IEEE 9th International Conference on Software Testing, Verification and Validation Workshops*, 04 2016, pp. 174–179.

[69] S. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro, "Proteum/fsm: A tool to support finite state machine validation based on mutation testing," in *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society.* IEEE Computer Society, 1999, pp. 96–104.

[70] E. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: an empirical study," *Software Testing Verification and Reliability*, vol. 9, pp. 205–232, 1999.

[71] J. Offutt, A. Lee, G. Rothermel, R. 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, 1996.

[72] R. Just, F. Schweiggert, and G. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 612–615.

[73] R. Just, M. Ernst, and G. Fraser, “Efficient mutation analysis by propagating and partitioning infected execution states,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, 2014.

[74] G. Edgecombe, “A swing component with emulates a vt100 terminal,” 2023. [Online]. Available: ”http://www.grahamedgecombe.com/projects/jterminal”

[75] R. Just, “The major mutation framework: Efficient and scalable mutation analysis for java,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, 2014, pp. 21–25.

[76] R. Hall, “Generalized behaviour-based retrieval,” in *Proceedings, International Conference on Software Engineering*, Baltimore, MD, May 1993.