2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

# MESSI: Mutant Evaluation by Static Semantic Interpretation

Matthew Patrick*, Manuel Oriol*† and John A. Clark*
*University of York
Heslington, York
United Kingdom
{mtp, manuel, jac}@york.ac.uk
†Industrial Software Systems
ABB Corporate Research
Baden-Dättwil, Switzerland
manuel.oriol@ch.abb.com

***Abstract*—Mutation testing is effective at measuring the adequacy of a test suite, but it can be computationally expensive to apply all the test cases to each mutant. Previous research has investigated the effect of reducing the number of mutants by selecting certain operators, sampling mutants at random, or combining them to form new higher-order mutants. In this paper, we propose a new approach to the mutant reduction problem using static analysis. Symbolic representations are generated for the output along the paths through each mutant and these are compared with the original program. By calculating the range of their output expressions, it is possible to determine the effect of each mutation on the program output. Mutants with little effect on the output are harder to kill. We confirm this using random testing and an established test suite. Competent programmers are likely to only make small mistakes in their programming code. We argue therefore that test suites should be evaluated against those mutants that are harder to kill without being equivalent to the original program.**

***Keywords*-mutation testing; sampling; static analysis;**

## I. Introduction

Mutation testing uses artificial faults to determine the fault finding capability of a test suite. For this to be effective, the artificial faults must be representative of actual faults in the software. As there are potentially an infinite number of artificial faults, techniques have been devised to select subsets from those that are available. The coupling and competent programmer hypotheses suggest that experienced programmers make small mistakes and that complex failures are linked to simple failures [1] [2]. Offutt [3] uses these hypotheses to claim that simple mutation operators are sufficient to detect complex faults. Yet, just because a mutation is small syntactically does not mean it is useful at representing simple failures. A small change in syntax can have a large effect on semantics [4]. This paper introduces a new technique for Mutant Evaluation by Static Semantic Interpretation (MESSI).

Unlike previous approaches to mutation testing, MESSI determines the usefulness of each mutant using static analysis, without reference to any particular test suite. MESSI can be used to select mutants according to their semantic similarity with the original program. If the symbolic execution of a mutant shows its output to have a similar range to the original program, we predict it to be close to the original program in semantics. The mutants most similar to the original program are selected for mutation testing. In our experiments, the average mutant in the top quarter is less than half as likely to be killed by a random test case than the average mutant in the remaining three quarters. The new technique provides an independent selection of semantically small mutations and forgoes the expense of evaluating mutants against a test suite.

The paper is organised as follows. Section II explores the background motivation behind the MESSI technique and Section III describes its general principles. Section IV gives further implementation details and section V explains the experimental setting. The results are evaluated in Section VI. Section VII discusses the related work to this research and section VIII presents our conclusions.

## II. Background

### A. Mutation testing

Software testing provides confidence in the correctness of software. A good test suite should aim to find errors in the software [5]. Mutation testing evaluates test suites against a set of artificial faults. Faults exist as incorrect steps, processes or data definitions somewhere within the programming code [6]. A test suite will reveal a fault if it has an effect on one of the local variables and then propagates that effect through to the output [7].

Artificial faults are mutations of the original programming code, typically with one small change in syntax. When test data has been found that causes a mutant to behave differently to the original program, we say the mutant has been killed. The proportion of mutants killed by a test suite is known as its mutation score (see Equation 1). A test suite with a high mutation score can be expected to perform well at finding actual faults in the software. Experimental research has shown mutation testing to be more stringent than other testing techniques and a good predictor of the real fault finding capability of a test suite [8][9].

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.161

$$Mutation\ score = \frac{number\ of\ mutants\ killed}{total\ number\ of\ mutants^*} \quad (1)$$

$$Cost\ reduction = 1 - \frac{number\ of\ mutants\ in\ sample}{total\ number\ of\ mutants^*} \quad (2)$$

*non-equivalent mutants

### B. The semantics of mutation

One of the greatest challenges to the validity of mutation testing is the number of mutants that are semantically equivalent to the original program. Equivalent mutants can skew the assessment of a test suite because they produce the same output as the original program for every possible input. For seven large Java programs, 45% of the mutants not detected by the test suite were shown to be equivalent [13]. Equivalent mutants occur when the mutation can never be exercised, its effect is later cancelled out or it is corroded away by other operations in the program [7]. Techniques have been devised to identify equivalent mutants using program slicing [10], compiler optimisation [11], constraint solving [12] and, more recently, impact assessment [13]. Equivalent mutants are still however difficult to remove completely.

It seems paradoxical therefore that the most useful mutants are those similar to the original program in semantics. The competent programmer hypothesis explains why most programs are either correct or very close to being correct. Competent programmers can make mistakes that have a large semantic effect, but they notice and remove them quickly without need for further testing. More subtle faults are harder to find. The semantic size of a mutation can be measured according to the number of times it is killed by a large random test suite. Mutations with a small semantic size are useful at representing actual faults. Ideally mutants should be semantically similar without being equivalent to the original program.

Algorithm 1 performs the remainder operation on variables *x* and *y*. It is presented along with three mutations of its programming code. Mutant M3 is equivalent to the original program because it post-increments variable *mod*, which is no longer used after the print statement. Mutant M1 will affect the output of every input, except if *x* and *y* are both 1, or if *x* is 0 and *y* is not. Mutant M2 only affects the output if *div* is less than zero. It is tempting to conclude that M2 is semantically smaller than M1, but further analysis is required to determine their exact semantic size.

**Algorithm 1** Remainder operation

| | | |
|---|---|---|
| 1: | $div \leftarrow x/y^\dagger$ | M1: $div \leftarrow x * y$ |
| 2: | **if** $div < 0$ **then** | |
| 3: | $mod \leftarrow (div * y) - x$ | M2: $mod \leftarrow (div * y) + x$ |
| 4: | **else** | |
| 5: | $mod \leftarrow x - (div * y)$ | |
| 6: | **end if** | |
| 7: | print(mod) | M3: print(mod++) |

†Integer division

### C. Subset selection

Typically it is not feasible to use every possible mutant of the program under test, even after all the equivalent mutants have been removed. It is therefore necessary to select a subset of mutants that allow the test suite to be evaluated within a reasonable period of time. Once the equivalent mutants have been removed, the reduction in cost achieved by a subset can be calculated as the proportion of non-equivalent mutants that are excluded (see Equation 2).

Mutants can be selected either for their semantic similarity to the original program, or their ability to represent the effectiveness of the original set. Both of these goals can be achieved through the use of random test suites. The first goal considers subsets effective if the test suite gives a low mutation score because mutants are difficult to kill if they are semantically similar to the original program. The second goal seeks subsets that give as high a mutation score as possible, relative to the original set of mutants. In other words, a test suite capable of killing all the mutants in an effective subset should also kill most of the mutants in the original set. Although the second goal has been popular in research, the competent programmer hypothesis suggests the first goal selects more useful mutants.

A subset of mutants can be selected either by reducing the number of mutation operators or by sampling mutants once they have been produced. It saves computation time to select the most efficient operators in advance. In experiments with Fortran, one mutation operator produced over half of the equivalent mutants and removing all but two of the operators gave a mutation score of 97.18% on the original set [11][14]. Sampling allows a greater variety of mutants to be selected. A 10% random sample gave on average 97.56% mutation score on the original set [14]. It is also possible to use syntactic and semantic analysis to optimise sampling for the program under test. Semantic analysis is more expensive, but provides a better evaluation of the effect of a mutation on the program's behaviour.

### D. Symbolic execution

Symbolic execution allows the abstract exploration of a program's semantics. Instead of executing the program with a concrete instance of the value of each variable, it represents the input variables symbolically. As the program is executed, a path condition and symbolic output expression is produced for each path. Figure 1 shows an example of this technique applied to Algorithm 1. The input variables *x* and *y* are represented symbolically to reveal that if X/Y is less than zero, the output will be ((X/Y)*Y)-X, otherwise it will be X-((X/Y)*Y). As this is a simple example, the semantics are immediately apparent. Both paths produce output with the same magnitude, but one is the negative of the other. Complications arise when the program includes loops or data structures such as objects, arrays and pointers.

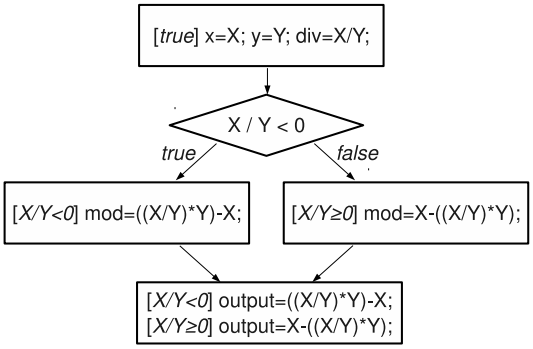


Figure 1. Symbolic Execution

## III. MESSI PRINCIPLES

MESSI evaluates mutants according to their semantic similarity with the original program. Semantic similarity (S) is calculated as the sum of differences between the minimum and maximum output values for each path (p) through a mutant (m) and the original program (o). A mutant is semantically similar to the original program if it has a low value for semantic similarity. Algorithm 2 outlines the process of using MESSI to select mutants. The minimum and maximum output value is determined for each path by processing its symbolic output expression. Mutants are selected if they are similar but not equivalent to the original program.

MESSI calculates minimum and maximum values progressively. Input variables have a prescribed range and constants have the same minimum and maximum value. The minimum and maximum values for arithmetic operations are given in Table II. The calculations for addition and subtraction are the same regardless of the sign of the inputs. Multiplication and division are more complicated. The minimum result of division is Min(L)/Max(R) when all the input values are positive, but Max(L)/Min(R) when all the input values are negative. Arithmetic operations eventually combine through sub-expressions to form the range of every path through a program.

**Algorithm 2** Using MESSI to select mutants

1) Calculate the minimum and maximum output values for each mutant
2) Sum up the differences into values for semantic similarity,

$$S_m = \sum_{p \in Paths} \begin{matrix} |Min(m) - Min(o)| \\ + \\ |Max(m) - Max(o)| \end{matrix}$$

3) Remove all the equivalent mutants using random testing and inspection
4) Order the mutants according to their similarity value
5) Select the mutants most similar to the original method

Table I shows the application of these calculations to Algorithm 1 with input values in the range 1 to 100. It is important to note that because MESSI computes values conservatively, the resulting range contains some values that are not possible. For example, MESSI uses the minimum value of *y* in line 1, but its maximum value in line 5 to calculate the maximum output along the second branch. Nevertheless, the similarity values confirm that mutant M3 is likely to be equivalent and that M2 is semantically more similar to the original program than M1.

Table I
SYMBOLIC EXECUTION FOR SELECTIVE MUTATION ON ALGORITHM 1

| | Path | Output | Min | Max | Similarity |
|---|---|---|---|---|---|
| **Original** | Left | ((X/Y)*Y)-X | -99.99 | 9900 | - |
| | Right | X-((X/Y)*Y) | -9900 | 99.99 | - |
| **M1** | Left | ((X*Y)*Y)-X | -99 | 999999 | 990099.99 |
| | Right | X-((X*Y)*Y) | -999999 | 99 | 990099.99 |
| **M2** | Left | ((X/Y)*Y)+X | 1.01 | 10100 | 301 |
| | Right | X-((X/Y)*Y) | -9900 | 99.99 | 0 |
| **M3** | Left | ((X/Y)*Y)-X | -99.99 | 9900 | 0 |
| | Right | X-((X/Y)*Y) | -9900 | 99.99 | 0 |

## IV. IMPLEMENTATION

MESSI should work well with most tools for symbolic execution and mutation testing. The experiments in this paper use JPF-symbc and muJava because they offer flexibility for customisation and produce each mutant as a separate program. muJava has twelve method-level mutation operators, selected to modify arithmetic, relational, logical and conditional expressions in the programming code [15]. Java Pathfinder (JPF) is an open source model checker and Java virtual machine, developed by NASA to find bugs in concurrency [16]. JPF-symbc performs symbolic execution by storing symbolic attributes along with each variable on the stack [17]. It is capable of processing both integer and real numeric values and includes constraint solving packages

Table II
THE MINIMUM AND MAXIMUM RESULTS OF EACH OPERATION

| | **Min(D)** | **Max(D)** |
|---|---|---|
| **L + R** | Min(L) + Min(R) | Max(L) + Max(R) |
| **L - R** | Min(L) − Max(R) | Max(L) − Min(R) |
| **L * R** | if((Min(L) ≥ 0) && (Min(R) ≥ 0))<br>  return Min(L) ∗ Min(R)<br>else if ((Max(L) ≤ 0) && (Max(R) ≤ 0))<br>  return Max(L) ∗ Max(R)<br>else return Smallest(Min(L) ∗ Max(R),<br>  Max(L) ∗ Min(R)) | if((Min(L) ≥ 0) && (Max(R) ≤ 0))<br>  return Min(L) ∗ Max(R)<br>else if ((Max(L) ≤ 0) && (Min(R) ≥ 0))<br>  return Max(L) ∗ Min(R)<br>else return Biggest(Max(L) ∗ Max(R),<br>  Min(L) ∗ Min(R)) |
| **L / R** | if((Min(L) ≥ 0) && (Min(R) ≥ 0))<br>  return Min(L) / Max(R)<br>else if ((Max(L) ≤ 0) && (Max(R) ≤ 0))<br>  return Max(L) / Min(R)<br>else return Smallest(Max(L) / Smallest(Max(R), -1)),<br>  Min(L) / Biggest(Min(R), 1)) | if((Min(L) ≥ 0) && (Max(R) ≤ 0))<br>  return Min(L) / Min(R)<br>else if ((Max(L) ≤ 0) && (Min(R) ≥ 0))<br>  return Max(L) / Max(R)<br>else return Biggest(Max(L) / Biggest(Min(R), 1)),<br>  Min(L) / Smallest(Max(R), -1)) |
| **L % R** | if(Min(R) ≥ 0) return 0<br>else return Max(-Max(L), Min(R) + 1) | if(Max(R) ≤ 0) return 0<br>else return Min(Max(L), Max(R) - 1) |
| **-L** | -Max(L) | -Min(L) |

for finding input values to exercise each path. muJava is mostly a tool for research, but JPF-symbc found a serious bug in the Onboard Abort Executive for the NASA Crew Exploration Vehicle [17].

Pre-processing transforms the software under test into a form that is suitable for use with JPF-symbc. Methods should have straightforward numerical inputs and outputs and not employ any side effects. The pre-processing procedure extracts methods to be tested from the classes in which they are contained, along with any methods on which they depend. It then introduces local constants to replace all calls to retrieve data from outside of the class and removes any calls to output data via a side effect. To avoid path explosion problems, each loop is only allowed to run once. It should be noted that these preprocessing transformations are not semantically preserving. The pre-processing procedure could be made more sophisticated by including appropriate coding strategies for non-numerical or side effect input and output, but this would make the process slower and more complicated.

Once the semantic similarity values have been calculated, it is necessary to remove the equivalent mutants before selection is performed. If the similarity value is zero, the mutant is highly similar but not necessarily equivalent, because it may still have different branch conditions. Therefore mutant with zero similarity value are executed with one million random input values to determine whether they are equivalent. When none of the input values produce a difference, the mutant is checked manually for equivalence. Mutants can then be selected in the order of their similarity value, resolving any ties at random. The selected mutants represent faults that are difficult for a programmer to detect because they only affect program behaviour under certain circumstances.

## V. EXPERIMENTS

Experiments were set up to answer three main research questions about our new mutant selection technique:

1) **How difficult is it to kill mutants selected by MESSI?**
2) **Does MESSI highlight mutants not killed by existing test suites?**
3) **What is the ideal size of subset to use when selecting mutants with MESSI?**

The first research question is addressed by applying random testing to nineteen methods from six classes of the Java standard library (see Table III). We selected methods according to the following criteria: Their return type must be numeric, they must take at least one parameter, all their parameters must be numeric and they must occupy over 1KB in memory. Numerical software is used because of limitations of the technique and large methods are chosen to avoid trivial results. Each mutant is assessed according to whether it is killed and the number of times it is killed by a large suite of random test cases. If MESSI is successful, we would expect it to select mutants that are killed infrequently by the test suite.

We address the second research question using TCAS, the Traffic Collision Avoidance System developed by researchers at Siemens [19]. TCAS was created to investigate the effectiveness of coverage criteria in testing, so it has a large test suite of black-box and white-box test cases. TCAS consists of 173 lines of code and 9 methods. Although, the original program is written in C, we translated it into Java maintaining as many of the original characteristics as possible. For example, logical values are represented as integers in the same way there are in C. If MESSI is successful, it should reveal many potential faults not covered by the existing test cases.

Table III
SUBJECT METHODS FROM THE JAVA STANDARD LIBRARY [18]

| | | LOC | Mutants | Equivalent Mutants |
|---|---|---|---|---|
| **java.math.BigDecimal** | | | | |
| 1 | $int\_checkScale(long)$ | 16 | 60 | 15 |
| 2 | $int\_longCompareMagnitude(long, long)$ | 18 | 44 | 15 |
| 3 | $long\_longMultiplyPowerTen(long, int)$ | 18 | 98 | 28 |
| **java.math.BigInteger** | | | | |
| 4 | $int\_getInt(int)$ | 18 | 46 | 0 |
| **javax.swing.JTable** | | | | |
| 5 | $int\_limit(int, int, int)$ | 5 | 36 | 1 |
| **javax.swing.plaf.basic.BasicTabbedPaneUI** | | | | |
| 6 | $int\_calculateMaxTabHeight(int)$ | 8 | 42 | 2 |
| 7 | $int\_calculateMaxTabWidth(int)$ | 8 | 42 | 2 |
| 8 | $int\_calculateTabAreaHeight(int, int, int)$ | 8 | 51 | 1 |
| 9 | $int\_calculateTabAreaWidth(int, int, int)$ | 8 | 51 | 1 |
| 10 | $int\_getNextTabIndex(int)$ | 4 | 17 | 0 |
| 11 | $int\_getNextTabIndexInRun(int, int)$ | 12 | 63 | 18 |
| 12 | $int\_getPreviousTabIndex(int)$ | 4 | 50 | 18 |
| 13 | $int\_getPreviousTabIndexInRun(int, int)$ | 12 | 91 | 24 |
| 14 | $int\_getRunForTab(int, int)$ | 9 | 43 | 6 |
| 15 | $int\_lastTabInRun(int, int)$ | 11 | 81 | 11 |
| **javax.swing.plaf.basic.BasicTreeUI** | | | | |
| 16 | $int\_findCenteredX(int, int)$ | 5 | 49 | 49 |
| 17 | $int\_getRowX(int, int)$ | 3 | 25 | 0 |
| **javax.swing.text.AsyncBoxView** | | | | |
| 18 | $float\_getInsetSpan(int)$ | 5 | 27 | 1 |
| 19 | $float\_getSpanOnAxis(int)$ | 7 | 17 | 1 |

The final research question is addressed for 10%, 25%, 50% and 100% samples of mutants from the Java standard library. We determine the difficulty of killing each mutant as the proportion of random test cases that produce a different output. The experiment uses thirty test suites of one million test cases, each with random input values between -1000 and 1000 inclusive. Samples are taken thirty times to achieve a fair average, using the same seeds to resolve ties in similarity. If MESSI is successful, the smaller the sample the harder the mutants should be to kill.

## VI. RESULTS AND ANALYSIS

### A. How difficult is it to kill mutants selected by MESSI?

In general MESSI selects mutants that are more difficult to kill. Table V shows the average mutation score of samples made from 19 methods in the Java standard library. Of the first ten percent of mutants selected, 42.6% of them were killed, compared to 69.4% of the complete set. Table VI shows the probability that an arbitrary test case will kill an arbitrary mutant. The probability of killing a mutant in the first quarter is 20.4%, compared to 38.4% in the complete set. This implies that the average mutant in the top quarter is less than half as likely to be killed by a random test case than the average mutant in the remaining three quarters.

There are however exceptions where MESSI is shown to be less effective. As the random test suite kills all the non-equivalent mutants of *int_calculateMaxTabHeight(int)*, selecting a subset has no effect on the mutation score. The probability of killing a mutant in the first ten percent is also higher on average than in the first quarter. In particular, the probability of killing mutants of *int_lastTabInRun(int,int)* is higher in all of the subsets than in the complete set. The Wilcoxon and student T-tests in Table IV show that although the top quarter of mutants *is* less likely to be killed than the remaining three quarters, there is insufficient statistical evidence that it is less than half as likely.

Table IV
STATISTICAL ANALYSIS FOR QUESTION A

| | remainder-25% | remainder/2-25% |
|---|---|---|
| Mean | 22.4 | 0.100 |
| Variance | 491 | 153 |
| T-Value | 4.40 | 0.040 |
| Critical | 2.552 **(reject)** | 2.552 **(accept)** |
| Wilcoxon | 181+ 9- | 98+ 92- |
| Critical | 37 **(reject)** | 37 **(accept)** |

### B. Does MESSI highlight mutants not killed by test suites?

The results of the experiment with TCAS show MESSI to have little effect on the mutation score of the existing test suite. The first ten percent of mutants have a mutation score of 0.104, the first twenty-five percent 0.093, the first fifty percent 0.154 and the complete set 0.096. The probability of an arbitrary test case killing an arbitrary mutant is 20.6% for the first ten percent, 19.2% for the first twenty-five percent,

19.4% for the first fifty percent and 64.6% for the complete set. MESSI selects mutants less frequently killed, but is unable to highlight mutants not killed by the test suite.

The mixed results confirm the weaknesses and strengths of using MESSI to select mutants. As the test suite was specifically designed for TCAS, it contains test cases to detect faults difficult to kill by random testing. MESSI is not effective enough to find faults missed by well-designed test suites, but it does still offer some insight into the construction of a new test suite.

### *C. What is the ideal size of subset to use when selecting mutants with MESSI?*

The results of Table V and VI are illustrated graphically in Figures 2 and 3. It is clear that the greatest difference in mutation score and probability is made between the fifty and one-hundred percent sample sizes. There is a smaller difference between twenty-five and fifty percent and the difference between ten and twenty-five percent appears negligible. The first twenty-five percent of mutants are slightly more difficult to kill, but selecting the top fifty percent may allow for more faults to be revealed.

The Wilcoxon and student T-tests in Table VII show a significant mutation score between fifty and one-hundred percent and twenty-five and fifty percent. For mutant-killing probability, only the difference between fifty and one-hundred percent is significant. The ideal sample size is therefore either fifty or twenty-five percent of the complete set. Ultimately, the choice is between computational time and fault-finding effectiveness.

Table V
MUTATION SCORE FOR 19 METHODS FROM THE JAVA STANDARD LIBRARY

| | **10%** | **25%** | **50%** | **100%** |
|---|---|---|---|---|
| 1 | 0.725 | 0.739 | 0.752 | 0.822 |
| 2 | 0.817 | 0.829 | 0.812 | 0.828 |
| 3 | 0.267 | 0.239 | 0.404 | 0.671 |
| 4 | 0.500 | 0.527 | 0.500 | 0.609 |
| 5 | 0.522 | 0.504 | 0.647 | 0.829 |
| 6 | 1.000 | 1.000 | 1.000 | 1.000 |
| 7 | 0.442 | 0.433 | 0.550 | 0.775 |
| 8 | 0.447 | 0.456 | 0.720 | 0.860 |
| 9 | 0.493 | 0.486 | 0.720 | 0.863 |
| 10 | 0.733 | 0.658 | 0.658 | 0.765 |
| 11 | 0.175 | 0.200 | 0.227 | 0.444 |
| 12 | 0.211 | 0.229 | 0.563 | 0.788 |
| 13 | 0.033 | 0.031 | 0.023 | 0.162 |
| 14 | 0.500 | 0.489 | 0.500 | 0.514 |
| 15 | 0.644 | 0.618 | 0.599 | 0.710 |
| 16 | 0.000 | 0.000 | 0.208 | 0.612 |
| 17 | 0.333 | 0.378 | 0.500 | 0.760 |
| 18 | 0.633 | 0.611 | 0.641 | 0.615 |
| 19 | 0.267 | 0.225 | 0.250 | 0.563 |
| **Mean:** | 0.426 | 0.455 | 0.541 | 0.694 |

Table VI
PROBABILITY TO KILL MUTANT FOR 19 METHODS FROM THE STANDARD JAVA LIBRARY

| | **10%** | **25%** | **50%** | **100%** |
|---|---|---|---|---|
| 1 | 22.7% | 24.7% | 22.9% | 29.2% |
| 2 | 12.5% | 13.3% | 11.9% | 13.8% |
| 3 | 5.95% | 6.07% | 4.29% | 8.57% |
| 4 | 13.8% | 14.2% | 13.0% | 13.7% |
| 5 | 26.3% | 28.7% | 24.5% | 32.5% |
| 6 | 59.1% | 60.1% | 69.4% | 84.1% |
| 7 | 26.1% | 26.1% | 38.2% | 65.2% |
| 8 | 37.9% | 37.3% | 67.3% | 82.3% |
| 9 | 41.2% | 40.6% | 67.3% | 82.7% |
| 10 | 73.3% | 65.8% | 65.8% | 72.1% |
| 11 | 4.79% | 5.45% | 5.72% | 13.9% |
| 12 | 14.4% | 17.1% | 26.3% | 45.5% |
| 13 | 0.00% | 0.00% | 0.00% | 3.68% |
| 14 | 7.22% | 9.6% | 9.45% | 12.2% |
| 15 | 11.9% | 14.3% | 13.8% | 10.2% |
| 16 | 0.00% | 0.00% | 7.85% | 28.3% |
| 17 | 33.3% | 37.8% | 50.0% | 75.5% |
| 18 | 25.0% | 19.7% | 22.6% | 34.6% |
| 19 | 0.00% | 0.00% | 0.00% | 31.3% |
| **Mean:** | 21.9% | 21.4% | 26.7% | 38.4% |

## VII. RELATED WORK

Most research into selective mutation focuses on selecting a subset of mutants that perform well compared to the original set. Barbosa et al. [20] compared the effectiveness of four random sampling strategies against three operator selection techniques for 32 small C programs. Their results show how significant cost reduction can be achieved and it still be possible to generate test data to kill most of the mutants in the original set. Hussain [21] improved upon these results further by clustering mutants according to the syntactic similarity and selecting mutants from each cluster to promote variety. The problem with these techniques is that they treat all mutants as being equally useful. MESSI improves upon this by favouring mutants that are semantically similar to the original program.

Recent research attempts to optimise the selection of mutants through search-based techniques. Adamopoulos et al. [22] co-evolved the selection of mutants and test cases to find mutants that are hard to kill and test cases capable of killing them. Jia and Harman [23] search for higher order mutants as combinations of multiple mutations that are syntactically complex but semantically similar to the original program. Both approaches evaluate mutants using a test suite. Mutants not killed by any test case are considered equivalent, even if they are really just difficult to kill. In this way, these approaches ignore mutants the competent programmer hypothesis considers most useful for mutation testing. MESSI evaluates mutants independently from their test suite and provides an absolute rather than relative assessment of their usefulness.

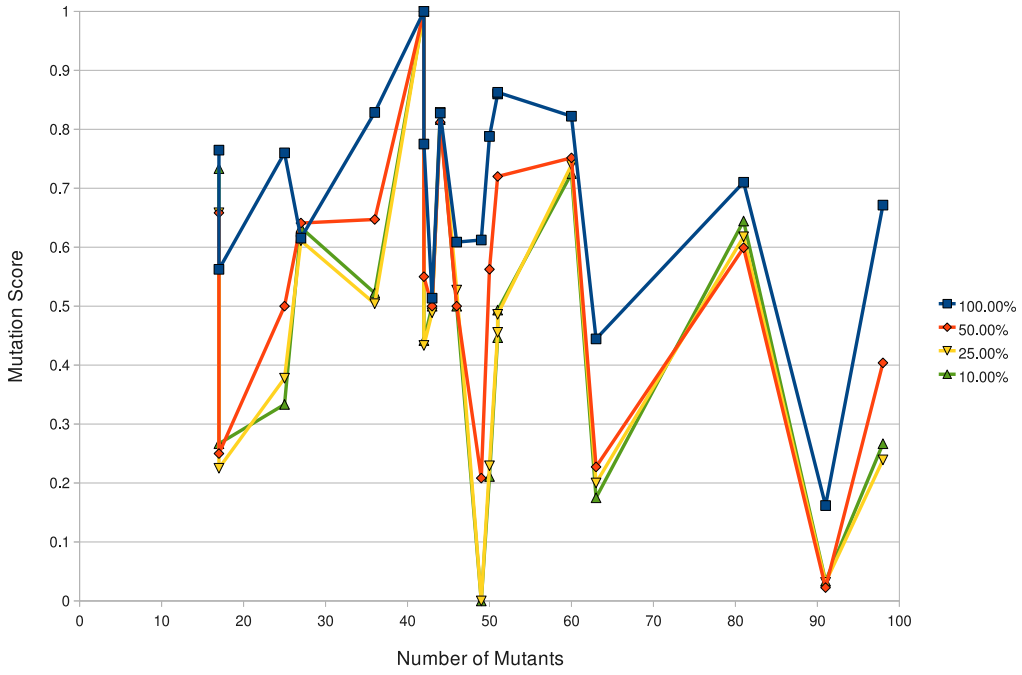


Figure 2. Experiment Results - Mutation Score

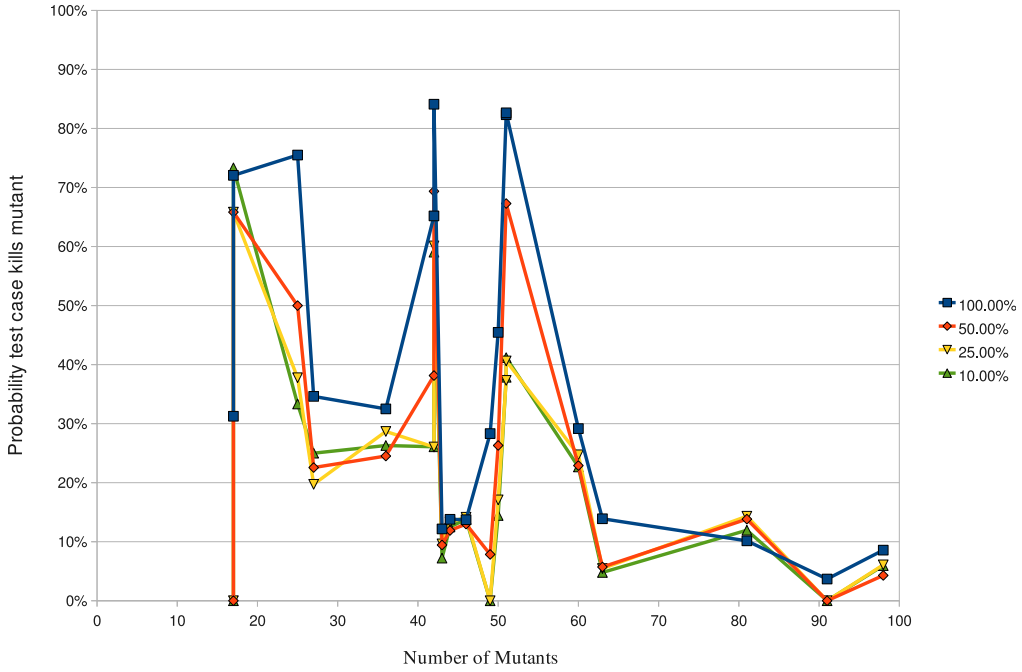


Figure 3. Experiment Results - Probability

There are other techniques for comparing mutant semantics without the use of a test suite. Ellims et al. [24] suggested measuring memory usage or execution time, but unpredictable factors involving cache and underlying system software can affect these results. Schuler and Zeller [13] compared the number of statements exercised and the values returned by each procedure. They were able to remove most of the equivalent mutants using this technique, but this also removed a considerable number of non-equivalent mutants. MESSI provides a more definite evaluation of semantics. A mutant cannot be equivalent if it has a different symbolic output. This still does not remove the necessity of manual investigation, but at least it reduces the number of mutants that require inspection.

Table VII
STATISTICAL ANALYSIS FOR QUESTION C

Mutation score:

| | **100%-50%** | **50%-25%** | **25%-10%** |
|---|---|---|---|
| Mean | 0.153 | 0.085 | -0.005 |
| Variance | 0.95 | 0.012 | 0.001 |
| T-Value | 5.853 | 3.355 | -0.751 |
| Critical | 2.552 **(reject)** | 2.552 **(reject)** | 2.552 **(accept)** |
| Wilcoxon | 168+ 3- | 135.5+ 17.5- | 66+ 87- |
| Critical | 32 **(reject)** | 27 **(reject)** | 27 **(accept)** |

Probability:

| | **100%-50%** | **50%-25%** | **25%-10%** |
|---|---|---|---|
| Mean | 11.5% | 5.23% | 2.86% |
| Variance | 97.5% | 95.8% | 27.2% |
| T-Value | 5.149 | 2.378 | 0.458 |
| Critical | 2.552 **(reject)** | 2.552 **(accept)** | 2.552 **(accept)** |
| Wilcoxon | 185+ 5- | 130+ 60- | 117+ 53- |
| Critical | 37 **(reject)** | 37 **(accept)** | 32 **(accept)** |

## VIII. CONCLUSIONS

MESSI was shown to select mutants that are difficult to kill. It achieves this by comparing their symbolic output through static analysis. The competent programmer hypothesis claims faulty programs are typically similar to the correct program in semantics, so MESSI should select useful mutants. MESSI does not however always select the most difficult to kill mutants. Although removing half the mutants has a significant effect on the mutation score and probability of killing each mutant, there is little difference between the twenty-five and fifty percent samples. In the case of TCAS, MESSI has little effect on the mutation score at all. Therefore, MESSI is a useful approach, but effort should be made to improve its reliability.

The current version of MESSI calculates semantic difference without taking path conditions into account. Mutated path conditions can leave the symbolic output unchanged. This may explain why the top ten percent of mutants are often easier to kill than the top twenty-five percent. In addition, our experimental evaluation of MESSI is based on random testing with thirty trials of a million test cases. As the input range was limited from -1000 to 1000, there is a lot of redundancy for programs that only have one input. The goal with the next version of MESSI will be to increase the sophistication of its semantic interpretation and evaluate its performance using larger and more complex programs.

MESSI will be rewritten as a new plug-in for JPF that does not rely on JPF-symbc. Every numerical value will be represented at run-time with a minimum and maximum value. Research ideas will be incorporated regarding complex objects and non-numerical types. Objects for example, can be compared be expanding their sub-classes and inspecting their data fields. Boolean types can be treated as integers that have a very limited range. Other ideas to be considered include taking into account the distribution rather than just the range of the output and recording dynamic information in the form of program traces to better direct the approach. The next version of MESSI should be more effective and evaluate paths more quickly.

## REFERENCES


[1] 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, 1978.

[2] T. A. Budd, "Mutation analysis of program test data," Ph.D dissertation, Dept, Comp. Sci., Yale Univ., New Haven, CT, 1980.

[3] A. J. Offutt, "Investigations of the software testing coupling effect," in *ACM Trans. Softw. Eng. Meth.*, vol. 1, no. 1, pp. 5-20, Jan. 1992.

[4] A. J. Offutt, "A semantic model of program faults," in *Proc. ISSTA*, 1996, pp. 195-200.

[5] G. J. Myers, T. Badgett and C. Sandler, *The art of software testing* 3rd ed. Hoboken, NJ: Wiley, 2011, ch. 2, pp. 5.

[6] *IEEE standard glossary of software engineering terminology*, IEEE Standard 610.12, 1990.

[7] J. M. Voas, and K. W. Miller, "Software testability: the new verification," in *IEEE Softw.*, vol. 12, no. 3, pp. 17-28, May 1995.

[8] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing: an experimental comparison of effectiveness," in *J. Syst. Softw.*, vol. 38, no. 3, pp. 235-253, Sept. 1997.

[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proc. ICSE*, 2005, pp. 402-411.

[10] R. Hierons, M. Harman, S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," in *Softw. Test. Verif. Rel.*, vol. 9, no. 4, pp. 233-262, Dec. 1999.

[11] A. J. Offutt, and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants, in *Softw. Test. Verif. Rel.*, vol. 4, no. 3, pp. 131-154, Dec. 1994.

[12] A. J. Offutt, and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," in *Softw. Test. Verif. Rel.*, vol. 7, no. 3, pp. 165-192, Sep. 1997.

[13] D. Schuler, and A. Zeller, "(Un-)covering equivalent mutants," in *Proc. ICST*, 2010, pp. 45-54.

[14] A. P. Mathur, and W. E. Wong, "Reducing the cost of mutation testing: an empirical study," in *J. Syst. Softw.*, vol. 31, no. 3, pp. 185-196, Dec. 1995.

[15] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava : an automated class mutation system," in *Softw. Test. Verif. Rel.*, vol. 15, no. 2, Jun. 2005, pp. 97-133.

[16] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. "Model checking programs," in *J. Aut. Softw. Eng.*, vol. 10, no. 2, Apr. 2003, pp. 3-12.

[17] C. S. Păsăreanu, and N. Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode," in *Proc. ASE*, Sep. 2010, pp. 179-180.

[18] Oracle. "Java$^{TM}$Platform, Standard Edition 7 API Specification," http://docs.oracle.com/javase/7/docs/api/, 1993-2011 [Jan. 17, 2012].

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. ICSE*, May 1994, pp. 191-200.

[20] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," in *Softw. Test. Verif. Rel.*, vol. 11, no. 2, pp. 113-136, May 2001.

[21] S. Hussain, "Mutation clustering," MSc dissertation, Dept. Comp. Sci., KCL, London, UK, Sep. 2008.

[22] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proc. GECCO*, June 2004, pp. 1338-1349.

[23] Y. Jia, and M. Harman, "Higher order mutation testing," in *J. Inf. Softw. Tech.*, vol. 51, no. 10, Oct. 2009, pp. 1379-1393.

[24] M. Ellims, D. Ince, and M. Petre, "The csaw c mutation tool: initial results," in *Proc. TAICPART*, 2007, pp. 185-192.