
Proceedings Paper:
Patrick, Matthew, Alexander, Rob orcid.org/0000-0003-3818-0310, Oriol, Manuel et al. (1 more author) (2014) Probability-based semantic interpretation of mutants. In: Proceedings - IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops:ICSTW 2014. 7th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014, 31 Mar - 04 Apr 2014 IEEE Computer Society , GBR , pp. 186-195.

https://doi.org/10.1109/ICSTW.2014.18

Probability-Based Semantic Interpretation of Mutants

Matthew Patrick*†, Rob Alexander†, Manuel Oriol†‡ and John A. Clark*
*Department of Plant Sciences, University of Cambridge, United Kingdom, mtp33@cs.york.ac.uk
†Department of Computer Science, University of York, United Kingdom, {mtp, rda, manuel, jac}@cs.york.ac.uk
‡Industrial Software Systems, ABB Corporate Research, Baden-Dättwil, Switzerland, manuel.oriol@ch.abb.com

***Abstract*—Mutation analysis is a stringent and powerful technique for evaluating the ability of a test suite to find faults. It generates a large number of mutants and applies the test suite to them one at a time. As mutation analysis is computationally expensive, it is usually performed on a subset of mutants. The competent programmer hypothesis suggests that experienced software developers are more likely to make small mistakes. It is prudent therefore to focus on semantically small mutants that represent mistakes developers are likely to make. We previously introduced a technique to assess mutant semantics using static analysis by comparing the numerical range of their symbolic output expressions. This paper extends our previous work by considering the probability the output of a mutant is the same as the original program. We show how probability-based semantic interpretation can be used to select mutants that are semantically more similar than those selected by our previous technique. In addition to numerical outputs, it also provides support for modelling the semantics of Boolean variables, strings and composite objects.**

***Keywords*—mutation analysis; semantic interpretation; static analysis; symbolic execution; mutant selection;**

I. Introduction

Mutation analysis is more stringent than other testing techniques and a good predictor of the real fault finding capability of a test suite [1] [7]. Yet it is computationally expensive to apply and requires significant human effort to interpret the results [3]. Mutation analysis generates a large number of mutants (each with a small syntactic change from the original program), then executes the test suite against them one at a time. It is computationally expensive to apply the test suite to all the mutants that are generated [4]. Significant human effort is required to discern the expected output for each test input used to kill mutants (i.e. distinguish them from the original program) and identify semantically equivalent mutants (that cannot be killed by any test input) [5]. Mutation analysis may be made less expensive (with a small reduction in capability) by evaluating the test suite with a smaller set of mutants [4].

The veracity of mutation analysis depends upon the mutants that are applied. It is important to select mutants that are representative of mistakes programmers are likely to make and cover the potential range of faults as completely as possible. A mutant's contribution to the coverage of potential faults can be considered in terms of the test inputs that may be used to kill it. Mutants that are killed by a superset of the input values that kill another mutant (or group of mutants) can be considered less valuable that those that require a unique combination of test input values. In general, mutants that are more difficult to kill contribute more to mutation analysis, as they are less likely to be killed as a 'side effect' of killing other mutants.

Although there have only been a few investigations comparing mutants with actual faults, research suggests that mutants do represent faults programmers are likely to make [6][7]. This helps to confirm the coupling and competent programmer hypotheses that experienced programmers make small mistakes and that complex failures are linked to simple failures [8][9]. Small mistakes in semantics result in programs with behaviour that is close to being correct. These faults are less likely to be repaired by the developer, as they are difficult to detect [7]. Although small mutations of the program code can be used to represent small semantic mistakes, just because a mutation is small syntactically does not mean it represents a semantically small fault. Small changes in syntax can have a large effect on semantics [10]. Many mutants are trivially easy to kill and thus do not represent faults a programmer is likely to make.

This paper introduces a new technique for evaluating and selecting mutants according to their semantic similarity to the original program. This is useful for two reasons: firstly, because semantically similar mutants are more likely to represent mistakes that a programmer might make; secondly, because semantically similar mutants contribute more to the coverage of potential faults (i.e. other mutants are likely to be killed as a side effect of killing semantically similar mutants). Our technique evaluates the semantics of each mutant using static analysis, without reference to any particular test suite. This allows for the independent selection of semantically small mutations and forgoes the expense of evaluating all the mutants against a test suite. Only the mutants most similar to the original program are selected for use in mutation analysis.

Our technique evaluates mutants based on an approximation of the probability that (for a given range of inputs), they will produce the same output as the original program. Mutants that are predicted to have a high probability of producing the same output as the original program are likely to be similar in semantics. This work extends our previous technique [11] by incorporating a more advanced model of mutant semantics. Previously, we evaluated mutants according to the difference between their output range and that of the original program. Both our new and previous technique apply dynamic symbolic execution to each path through the program and perform semantic analysis on the resulting symbolic expressions.

The paper is organised as follows. Section II explores some background ideas and Section III describes the general principles of our technique. Section IV presents our model, with an example in Section V. Section VI describes our experiments, Section VII explains our methodology and Section VIII analyses the results. Our conclusions are presented in Section IX and Section X describes an opportunity for further work.

II. Background

Semantic interpretation uses symbolic execution to perform an abstract exploration of a program's semantics. Instead of executing the program with actual input values, symbolic execution represents each input symbolically. As the program is executed, a path condition and symbolic output expression are constructed for each path. Path conditions describe the requirements that must be met for each path to be executed with actual inputs. Symbolic expressions represent the output of each path symbolically in terms of the input variables. Symbolic execution makes it possible to reason about all possible executions of a program, not just those that have been encountered using actual inputs.

Figure II demonstrates an example of symbolic execution, as applied to the remainder algorithm. The input variables *x* and *y* are represented symbolically as X and Y. A new variable (div) is assigned the symbolic expression X/Y and subsequently used in the program. Elsewhere in the diagram, this variable is replaced by its symbolic expression (X/Y). Symbolic execution reveals that the output is ((X/Y)*Y)-X if X/Y is less than zero, otherwise it is X-((X/Y)*Y). The second expression is equal to the first multiplied by -1. This means that the output range (cardinality) is the same for both paths, but the signs and therefore the minimum and maximum values are swapped around. Symbolic execution can be used to analyse the output of each path through a program and all its mutants.

We use Java Pathfinder (JPF) to perform symbolic execution. JPF is an open source model checker and Java virtual machine, originally developed by NASA to find concurrency faults [12]. Extensions have been written for JPF to handle a variety of testing and verification tasks. JPF-symbc [13] is a symbolic execution extension for JPF. It performs symbolic execution by storing symbolic attributes along with each variable on the stack. JPF-symbc is capable of processing integer and real numeric values, Booleans, references and strings. A number of constraint solving packages are also included for finding input values to exercise each path. JPF-symbc has been used by Fujitsu to test web applications [14] and has helped find a bug in the Onboard Abort Executive for the NASA Crew Exploration Vehicle [13].

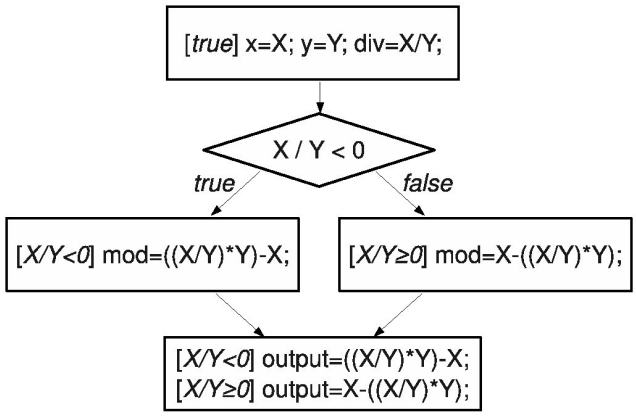

Fig. 1. An Example of Symbolic Execution

Symbolic output expressions and path conditions contain a number of entries for each input variable, connected together by operations that describe program behaviour. They can be processed to provide information about mutants and the original program. For example, the calculations in Table I are used to find the minimum and maximum numerical output values for each path from the range of the input variables. This particular model of arithmetic is used in both our difference and probability-based semantic interpretation techniques as part of their assessment of mutant similarity.

We use muJava to generate mutants for the programs under evaluation. muJava includes twelve method-level operators and twenty nine class-level operators [15]. We only use the method level operators, as the interaction between classes is outside the scope of this paper. The method-level operators are based on research into selective mutation started by Wong and Mathur [16] and continued by Offutt et. al. [17]. Mutation operators that alter or replace expressions were found to be the most effective in terms of cost reduction and mutation score. MuJava method-level mutation operators modify arithmetic, relational, logical and conditional expressions in the program code [15]. We use all the method level operators in our research.

TABLE I. The Minimum and Maximum Results of Each Numerical Operation [11]

	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))
	  return Min(L) ∗ Min(R)
	else if ((Max(L) ≤ 0) && (Max(R) ≤ 0))
	  return Max(L) ∗ Max(R)
	else return Smallest(Min(L) ∗ Max(R),
	  Max(L) ∗ Min(R))
		if((Min(L) ≥ 0) && (Max(R) ≤ 0))
		  return Min(L) ∗ Max(R)
		else if ((Max(L) ≤ 0) && (Min(R) ≥ 0))
		  return Max(L) ∗ Min(R)
		else return Biggest(Max(L) ∗ Max(R),
		  Min(L) ∗ Min(R))

L / R	if((Min(L) ≥ 0) && (Min(R) ≥ 0))
	  return Min(L) / Max(R)
	else if ((Max(L) ≤ 0) && (Max(R) ≤ 0))
	  return Max(L) / Min(R)
	else return Smallest(Max(L) / Smallest(Max(R), -1)),
	  Min(L) / Biggest(Min(R), 1))
		if((Min(L) ≥ 0) && (Max(R) ≤ 0))
		  return Min(L) / Min(R)
		else if ((Max(L) ≤ 0) && (Min(R) ≥ 0))
		  return Max(L) / Max(R)
		else return Biggest(Max(L) / Biggest(Min(R), 1)),
		  Min(L) / Smallest(Max(R), -1))

L % R	if(Min(R) ≥ 0) return 0 else return Max(-Max(L), Min(R) + 1)	if(Max(R) ≤ 0) return 0 else return Min(Max(L), Max(R) - 1)
-L	-Max(L)	-Min(L)

III. Probability-Based Semantic Interpretation

Probability-based interpretation estimates the probability that (for a value sampled uniformly at random from a given range of inputs) a mutant will have the same behaviour as the original program. This is calculated in terms of the probability that the mutant will follow the same path as the original program and produce the same output. To simplify this calculation and make the model computationally feasible, we do not consider the probability that a mutant will produce the same output as the original program if it follows a different path. Instead, we assume mutants that deviate from the path followed by the original program have a different output. The probability that a mutant behaves differently is therefore equal to the probability it follows a different path or it follows the same path but produces a different output (see Equation 1).

$$P(m = o) = \sum_{p \in Paths} \begin{array}{r} P(o_p) - P(m_p \&\& o_p) \\ + P(m_p \&\& o_p) * P(m_p = o_p) \end{array} \quad (1)$$

($P(o_p)$ is the probability the original program executes path p, $P(m = o)$ the mutant has the same output as the original program)

Probability-based interpretation is more accurate than difference-based interpretation [11]. Take for example two mutants, one with an output range of [1,10] and the other [1,100]. If the output range of the original program is [11,20], the sum of differences for the first mutant is 20 and the second 90. Our previous, difference-based, metric incorrectly assumed the first mutant is more similar to the original program because its minimum and maximum values are numerically closer. Yet, it is actually impossible for the first mutant to produce the same output as the original program. Our new, probability-based, metric reveals the second mutant is more similar because it has 0.1 probability of producing the same output as the original program, compared to the zero probability of the first mutant.

Our new metric also takes into account the likelihood that paths through a program are exercised. This is important for two reasons: firstly, the effect of a mutation may have an impact on one or more branch conditions, thus causing the mutant to follow a different path to the original program; secondly, semantic differences have a greater effect on the output if they occur on frequently exercised paths. A path through the program may guarantee that the mutant produces a different output, but unless it can be executed that path will have no effect. Modelling a program's control flow therefore allows more accurate semantic interpretation of its mutants.

We estimate the likelihood that a path is exercised by examining its branch conditions one at a time. A mutation can affect the control flow in one of two ways: directly, by changing the operations or values applied within a branch condition; or indirectly, by changing the value of a variable that is later used by a branch condition. The probability of satisfying each branch condition is estimated in the same way symbolic outputs are compared. For example, branch condition $x == 1$ has probability 0.01 of being exercised if the input domain of x is $[0, 99]$. We calculate the probability of satisfying each branch condition assuming independent probability distributions and a fixed input domain. The resulting metric is not completely accurate, but it is computationally feasible and allows reasonable approximations to be made.

IV. Modelling Semantic Probabilities

We model semantics by assuming that values are selected uniformly at random from a fixed input range, or (in the case of Boolean inputs) from a fixed probability of being true. Dynamic symbolic execution produces a symbolic output expression and path condition for each path through the program. From this, it is possible to predict the probability that a mutant will produce the same output as the original program by modelling the approximate effect of each operation in turn.

A. Numerical Expressions

Numerical expressions are compared according to their potential output range for a given range of input. We model the output range of a numerical operation using the calculations from Table I. Each operation affects the output range differently, but it is assumed (by approximation) that the distribution of output values is always uniform. Under this assumption, the probability that the output of a path through a mutant will be the same as the original program can be calculated in terms of the intersection of their output range (see Equation 2).

$$P(m_p = o_p) = \frac{|m_p^* \cap o_p^*|}{|m_p^*| * |o_p^*|} \quad (2)$$

(m_p^* is the range of outputs from mutant m along path p)

Numerical expressions are used within a Boolean expression, as part of an equality (e.g. $x == y$) or an inequality (e.g. $x > 5$). The probability that an equality is true can be predicted using Equation 2. Inequalities are evaluated by considering the range of values that appear in one output but not the other. Equation 3 describes the calculation for a greater-than operation (it is trivial to rearrange it for a less-than operation).

$$P(a_p > b_p) = \frac{\begin{array}{r} \max(\max(a_p^*) - \max(b_p^*), 0) * |b_p^*| + |a_p^* \cap b_p^*| \\ * \min(\max(a_p^*), \max(b_p^*)) - \min(b_p^*) - k)/2 \end{array}}{|m_p^*| * |o_p^*|} \quad (3)$$

($k = 1$ for discrete comparisons, $k = 0$ for continuous)

Numerical inequality satisfactions are typically dependent events. The probability that one of the inequalities is satisfied influences the probability that other inequalities in the expression are satisfied. In the simplest case, dependencies can be identified as overlapping regions (e.g. $x > 5 \&\& x > 10 \rightarrow x > 10$). More complex dependencies require the inequalities to be rewritten (e.g. $x + y > 3 \&\& x + 2y > 6 \rightarrow y > 3$).

We handle dependencies between inequalities with the simplex algorithm for linear algebra [18]. Inequalities are simplified and removed through a process of Gaussian elimination. Linear approximation is applied for any non-linear expressions. We prepare Boolean expressions for the simplex algorithm by rewriting them in canonical form (disjunction of conjunctions) using the Quine-McCluskey algorithm [19]. The simplex algorithm is applied to simplify each conjunction separately, before simplifying the disjunction as a whole using the rule $P(x \,||\, y) = P(x) + P(y) - P(x \&\& y)$.

B. Boolean Expressions

Although Boolean expressions can be represented with 1 (for true) or 0 (for false), limiting their range to just two values restricts the depth of semantic information they express; only operations involving tautologies or contradictions have any effect on the comparison of their outputs. We therefore represent Boolean expressions in terms of their probability of being true (see Table II). This allows more accurate estimation of the effect of each mutation in semantic output comparisons.

TABLE II. SEMANTIC INTERPRETATION OF BOOLEAN OPERATIONS

Operation	Probability Output True
$P(L \&\& R = T)$ (conjunction)	$P(L = T) * P(R = T)$
$P(L \mid\mid R = T)$ (disjunction)	$P(L = T) + P(R = T) - P(L \&\& R = T)$
$P(L \wedge R = T)$ (exclusive-or)	$P(L \mid\mid R = T) - P(L \&\& R = T)$

It is not possible to apply this model directly to expressions that contain repeated terms, such as $x \&\& x$, $x \mid\mid \neg x$ and $x \wedge x$. If x has a 50% chance of being true, our model predicts probabilities of 0.25, 0.75 and 0.5. The correct probabilities for these expressions are 0.5, 1.0 and 0.0. To avoid this problem, we first simplify the expressions into canonical form. For Boolean expressions to have the same output value, they must both be true or false (see Equation 4).

$$P(m_p = o_p) = \begin{array}{c} P(m_p = T) * P(o_p = T) + \\ P(m_p = F) * P(o_p = F) \end{array} \quad (4)$$

C. Bitwise Expressions

Although bit vectors are stored in numerical data types, bit operations act upon the value of each individual bit (1 or 0), rather than the numerical value as a whole. For the highest level of accuracy, it is important to consider the effect these operations have at the bit level. The probability of a bit value being 1 in the output of a conjunction, disjunction or exclusive-or can be determined by the same model as was used for Boolean values (where $1 = true$ and $0 = false$). Equation 5 gives the probability of two bit vector expressions returning the same output value.

$$P(m_p = o_p) = \prod_{i=0}^{n} P(m_{p,i} = o_{p,i}) \quad (5)$$

(n is the most significant bit set in o or m)

Bit vectors can also be included as terms of numerical operations (plus, minus, multiply etc.). These operations are modelled by transforming the probability of each bit being true into a minimum and maximum value for the bit vector (see Equation 6). The minimum value of a bit vector is determined by the bits fixed at 1 and the maximum value by the largest significant bit and the bits fixed at 0.

$$\begin{aligned} Min(x) &= \sum_{i=0}^{n} [P(x_i = 1) = 1] * 2^i \\ Max(x) &= 2^{n+1} - 1 - \sum_{i=0}^{n} [P(x_i = 1) = 0] * 2^i \end{aligned} \quad (6)$$

Iverson bracket notation: $[P] = \begin{cases} 1 \text{ if } P \text{ is true} \\ 0 \text{ otherwise} \end{cases}$

D. String Processing

We model strings on a character-by-character basis. A character is either concrete (it has a known value) or symbolic (its actual value is unknown, but it is assumed to be within a given set e.g the Latin alphabet). Each input string is represented as an array of 5 symbolic characters. As well as extracting and reordering symbolic characters, string operations can also introduce new concrete characters. By keeping track of the individual characters, it is straightforward to determine the effect of each string operation on the output (see Table III).

TABLE III. SEMANTIC INTERPRETATION OF STRING OPERATIONS

Operation	Output
concat(a, b)	$a_1 a_2 a_3 a_4 a_5 .. a_{\|a\|} b_1 b_2 b_3 b_4 b_5 .. b_{\|b\|}$
equals(a, b)	$[\|a\| = \|b\|] \prod_{i \in \|a\|} ([a_i \equiv b_i] + [a_i \not\equiv b_i \& (a_i \in S \| b_i \in S)]/N)$
length(a)	$\|a\|$
substring(a, b, c)	$a_b .. a_c$
startsWith(a, b)	$[\|a\| >= \|b\|] \prod_{i \in \|b\|} ([a_i \equiv b_i] + [a_i \not\equiv b_i \& (a_i \in S \| b_i \in S)]/N)$

(S is the set of symbolic inputs, N is the number of values assigned to each input (a..z would be 26) and $a \equiv b$ is true if a and b are identical symbolic or concrete values)

Table III describes the output of the 5 most commonly used string operations. In a survey of 38 Java applications, they were shown to be responsible for 90% of all string processing [20]. Many of the operations in the remaining 10% can be rewritten in terms of these operations. The probability of a symbolic and concrete character or two symbolic characters having the same value is $1/N$, where N is the size of the character set. If there are different symbolic or concrete characters at any point, or the length of the input strings is incompatible, these operations have a zero probability of being true. Otherwise probabilities are estimated under the assumption of their independence.

E. Objects

Ciupa et al. [21] describe a metric for measuring the distance between two objects according to their type, their field values and the field values of any other objects they reference. We have adapted this metric for static semantic analysis to calculate the probability that a mutant and the original program have the same output value (see Equation 7).

$$P(m_p = o_p) = \begin{cases} \prod_{i \in F(o)} P(m.i_p = o.i_p) & \text{if } type(m) = type(o), \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

($F(o)$ is the set of field values and objects referenced by o)

We calculate the probability that two objects output from a program have the same value by recursively comparing their types. Each object contains a collection of fields, which in turn may have primitive types or references to other objects. Two objects cannot have the same value if their primitive fields have a different type or they refer to differently typed objects. In this case, the probability of their equivalence is zero.

Pairs of objects that have the same type are compared recursively to assess the semantic similarity of their field values. We compare all the primitive field values inside these objects along with those contained in objects they refer to. Primitive values are compared symbolically, using symbolic expressions expressed in terms of the program input values.

Algorithm 1 Example ISBN class written in Java

```
public class ISBN {
   public String code = ""; public boolean valid = false;
   public int get(int pos)
      { return Integer.parseInt(code.substring(pos, pos)); }
   public ISBN(int prefix, int group, int publisher, int book)
   {
      if ((prefix <*_1 978)||(prefix >979)||(group <0)||(group >99999)||
          (publisher <0)||(publisher >9999999)||(book <0)||(book >999999))
            { valid = false; }
      else
      {
         code+=prefix+group+((publisher<10†)?"0":"")+publisher+book;
         valid = (code.length() == 12);
         if (valid)
         {
            int sum=(get(0)+3*get(1)+get(2)+3*get(3)+get(4)+3*get(5)+
                  get(6)+3*get(7)+get(8)+3*get(9)+get(10)+3*get(11));
            code += ((sum%10==0)?"0":(10 −‡ sum%10));
         }
      }
   }
}
```

*1 M1 changes < into >

† M2 changes 10 into 5

‡ M3 changes − into +

V. An Example of Semantic Analysis

As an example of probability-based semantic interpretation, Algorithm 1 describes a Java class for constructing ISBN (International Standard Book Number) codes. The class constructor inputs a prefix, along with the group, publisher and book code. It then checks they are within the range of the ISBN standard, then sets the value of the ISBN code accordingly.

Mutants produce a different output to the original program if they change the validation conditions, alter the circumstances under which a zero is added to the publisher code or calculate the check digit differently. Three mutations are annotated on the program code: M1 replaces a less-than sign with a greater-than sign in the validation conditions; M2 moves the zero addition threshold from 10 to 5; M3 replaces a minus sign in the check digit calculation with a plus. These mutations demonstrate various semantic changes that probability-based interpretation must take into account.

The probability that M1 has an effect on the output can be calculated as per the steps in Table IV. It produces a different output if the mutated branch condition evaluates true in the mutant, but not the original program, or vice versa (Step 1). This can be rewritten (Step 2) and simplified (Step 3) to show these events are independent. All other inequalities cancel out (Step 4), leaving only the one that has changed. Domain reduction reveals the mutant has an effect if *prefix* is not equal to 978 (Step 5). Assuming an input range of [0,999], the probability that the mutant has an effect is $977/1000 + 22/1000 = 0.999$.

The probability that M2 has an effect on the output can be calculated in a similar way. The probability that $publisher$ is less than 10 but not less than 5 is equal to $(10-5)/1000$ or 0.005. The probability that $publisher$ is less than 5 but not less than 10 is equal to $zero$. M2 will only be exercised if the first branch condition evaluates false. The probability of this is $(21 * 980)/(1000^2)$, assuming an input range of [1,1000] for each parameter. The probability that M2 has an effect on the output is therefore $0.02058 * 0.005 = 1.029e-4$.

The probability that M3 has an effect on the output can be calculated by considering the likelihood that it is executed and changes the value of *code*. M3 is exercised if the first branch evaluates false ($P = 0.02058$), the length of *code* is 12 ($P = 0.9^4$) and $sum\%10$ is not equal to zero ($P = 0.9$). M3 just affects the check digit. In the original program, its output range is $[10,10] - [0,9] = [1,10]$. In the mutant, this becomes $[10,10] + [0,9] = [10,19]$. These ranges only overlap once (with 10). The probability that M3 has an effect on the output is therefore $0.02058 * 0.9^5 * 0.99 = 0.012031$. Assuming every input parameter has the range $[0,999]$, M1 is predicted to have the greatest semantic effect and M2 the smallest.

TABLE IV. Calculating Probability of Differently Exercising First Branch

	Probability	Rule
1	P(m&&!o ||!m&&o)	Statement extraction
2	P(m&&!o)+P(!m&&o)-P(m&&!o&&!m&&o)	Equivalent expression
3	P(m&&!o)+P(!m&&o)	Contradiction
4	P($(prefix < 978)$&&!$(prefix > 978)$) +P(!$(prefix < 978)$&&$(prefix > 978)$)	Rewrite and contradiction
5	P($prefix < 978$)+P($prefix > 978$)	Domain reduction

(m and o are path conditions of the mutant and original program respectively)

VI. Experiments

Experiments were set up to answer the following two research questions regarding the potential for probability-based static semantic interpretation to be used as a metric for deciding which mutants to include in mutation analysis:

RQ1: Can our new approach to probability-based semantic interpretation be used to select mutants that are more semantically similar to the original program than our previous approach?

Previously, we interpreted mutant semantics in terms of the differences between their output range and that of the original program [11]. In general, we found there to be a positive correlation between the number of mutants selected using our previous metric and the mutation score / killing frequency achieved by random testing. There were however some exceptions in that smaller selections of mutants are not always on average more difficult to kill. Our new technique for probability-based selection aims to model mutant semantics more accurately. We consider whether the correlations achieved by our new technique are stronger than those previously observed.

We address this research question by selecting mutants that are predicted to be similar to the original program in different set sizes. We make two separate selections for each set size, one according to predictions made by difference-based semantic interpretation and the other according to predictions made by probability-based semantic interpretation. Probability-based interpretation is likely to be more effective than difference-based interpretation because it applies a more detailed model of semantics. We will consider probability-based interpretation to be more effective if it identifies mutants that are more difficult to kill on average than difference-based interpretation.

RQ2: Is probability-based interpretation capable of comparing the semantic similarity of more complex programs with greater accuracy than difference-based semantic interpretation?

We define a program to be complex if it has more lines of code and produces a greater number of mutants. Although our previous technique (for difference-based interpretation) was evaluated on real methods from the Java Standard Library, the largest of these methods had only 18 lines of code and the most number of mutants generated for a method was 70 [11]. It is likely that the effort required to perform unit testing and evaluate the output of each test case for these programs will not be overwhelming, since they are object-oriented [22]. It is therefore also useful to compare the accuracy of difference-based and probability-based interpretation on procedural programs that are slightly larger and produce more mutants.

We address this research question by applying difference-based and probability-based interpretation to three programs that are often used in unit testing research: FourBalls, TriTyp and Tcas (see Table IX). Our new probability-based metric can be used to interpret the semantic similarity of mutants that involve Booleans, bit vectors, strings and objects. Difference-based interpretation can only be used on numerical data types (i.e. integers or floating point values). We chose programs that make use of some other data types (Booleans and array objects), but output must be numerical and well-formed (i.e. object structure is not changed by mutation).

VII. Methodology

We applied our techniques to 19 methods from the Java Standard Library (see Table V) and three programs that are often used in research (see Table IX). The Java Standard Library methods were also used in our previous research [11]. They come from six different classes and process numerical values. The other three programs are slightly larger and more complex. This should help make semantic analysis more challenging. TriTyp only processes numerical values, but Tcas also works with Booleans and FourBalls outputs an array.

TABLE V. Methods from the Java Standard Library [23]

	LOC	Mutants
java.math.BigDecimal		
int checkScale(long)	16	60
int longCompareMagnitude(**long**,**long**)	18	44
long longMultiplyPowerTen(**long**,**int**)	18	98
java.math.BigInteger		
int getInt(**int**)	18	46
javax.swing.JTable		
int limit(**int**,**int**,**int**)	5	36
javax.swing.plaf.basic.BasicTabbedPaneUI		
int calculateMaxTabHeight(**int**)	8	42
int calculateMaxTabWidth(**int**)	8	42
int calculateTabAreaHeight(**int**,**int**,**int**)	8	51
int calculateTabAreaWidth(**int**,**int**,**int**)	8	51
int getNextTabIndex(**int**)	4	17
int getNextTabIndexInRun(**int**,**int**)	12	63
int getPreviousTabIndex(**int**)	4	50
int getPreviousTabIndexInRun(**int**,**int**)	12	91
int getRunForTab(**int**,**int**)	9	43
int lastTabInRun(**int**,**int**)	11	81
javax.swing.plaf.basic.BasicTreeUI		
int findCenteredX(**int**,**int**)	5	49
int getRowX(**int**,**int**)	3	25
javax.swing.text.AsyncBoxView		
float getInsetSpan(**int**)	5	27
float getSpanOnAxis(**int**)	7	17

TABLE VI. Benchmark Programs From Testing Research

Program	Mutants	LOC	Function
FourBalls	189	40	Ratio calculation [24]
TriTyp	310	61	Triangle classification [24]
Tcas	267	120	Air traffic control [25]

We select 10%, 25%, 50% and 100% of the mutants according to their predicted semantic similarity (the 10% selection should be most similar to the original program), then generate one million random test cases and count the number of times each mutant is killed. Mutants that are similar to the original program are likely to be killed less often. We evaluate our techniques using two metrics: mutation score (see Equation 8) represents the proportion of mutants killed at least once; killing frequency (see Equation 9) the proportion of tests that kill each mutant. If our techniques are effective, there should be a high correlation between these metrics and selection size.

$$\text{Mutation score} = \frac{\text{number of mutants killed}}{|M|} \tag{8}$$

$$\text{Killing frequency} = \frac{\sum_{m \in M} \text{number of test cases that kill } m}{\text{total number of test cases} * |M|} \tag{9}$$

(M is the set of non-equivalent mutants, i.e. those that can be killed)

VIII. Results and Analyses

A. Results for RQ1

RQ1 applies semantic interpretation to mutants generated from the Java Standard Library. We compare the mutation score and killing frequency for each selection size. The mean results for these measurements are presented in Tables VII and VIII respectively. Figures 2 and 3 are frequency density graphs of the Pearson correlation coefficients between selection size and mutation score / killing frequency. Examining these results will help us to anticipate the accuracy of our techniques in modelling mutant semantics for other similar programs.

Probability and difference based semantic interpretation were both able to select difficult to kill mutants (as determined by random testing) for most methods evaluated from the Java standard library. The mutants selected by probability-based interpretation were slightly harder to kill on average. Of the first ten percent of mutants selected by this technique, 42.3% were killed, compared to 42.6% for those selected by difference-based interpretation. This result is confirmed by the frequency curves in Figure 2, which have a lower density for small coefficients of probability-based interpretation and a slightly larger density for high correlations. The curve is also smoother, with smaller deviations from the trend, but overall there is little difference between the two techniques.

The results for killing frequency are similar. Both probability and difference-based interpretation select mutants that are more difficult to kill than average (see Table VIII). The killing frequency for mutants in the first quarter is 21.4% for difference-based interpretation and 20.8% for probability-based interpretation, compared to 38.4% for the complete set. This means the average mutant in the top quarter of those selected is less than half as likely to be killed by a random test case than the average mutant in the remaining three quarters. Probability-based interpretation also has a slightly higher density for high correlation coefficients and a lower density for small correlation coefficients (see Figure 3).

TABLE VII. Mean Results for Mutation Score

	Difference-Based	Probability-Based
10%	0.426	0.423
25%	0.455	0.435
50%	0.541	0.548
100%	0.694	0.694

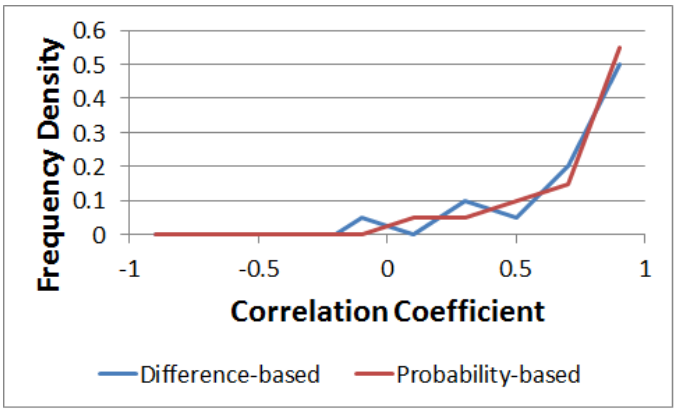

Fig. 2. Correlation Between Selection Size and Mutation Score

TABLE VIII. Mean Results for Killing Frequency

	Difference-Based	Probability-Based
10%	0.219	0.215
25%	0.214	0.208
50%	0.267	0.282
100%	0.384	0.384

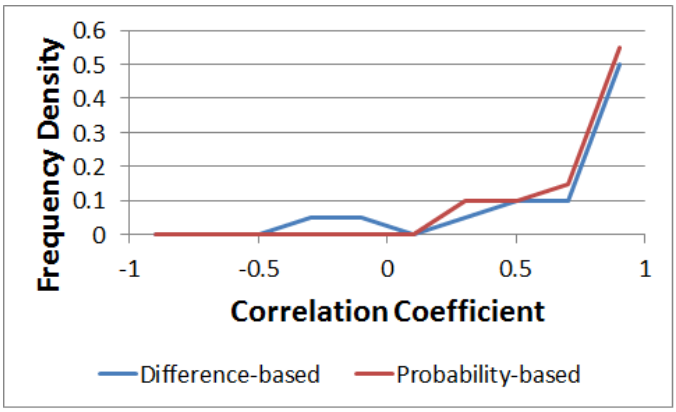

Fig. 3. Correlation Between Selection Size and Killing Frequency

One of the reasons why there is little difference in the results for probability and difference-based interpretation is that none of the methods evaluated in this research question are overly complicated. They have little branching structure and only process numerical values. This means that significant parts of our new semantic model are not needed for these programs. The effective difference between our new and old model is therefore very small. The main difference is that, instead of comparing output ranges by their numerical difference, we consider the probability that the same value occurs in both. In Section III we showed that this has a significant difference in some case, but this is not always the case. Probability-based interpretation is therefore only slightly more effective that difference-based interpretation on simple programs.

There are mutants for which difference-based and probability-based interpretation both fail to successfully compare semantic similarity. This is sometimes an artefact of the way we have conducted our experiments. For example, no decrease in mutation score is observed for any selection size of the *calculateMaxTabHeight*, because all of its mutants are killed by the random test suite (in this case, the killing frequency is a more effective metric). In other cases, there is a weakness in our techniques. For example, the lowest killing frequency for *lastTabInRun* occurred when using the complete set of mutants - all selections resulted in a significantly higher killing frequency. One significant weakness of our techniques is that we assume each output of a program is uniformly distributed. If outputs have a significantly different distribution, then our techniques become less accurate. It can therefore be said that, although difference-based and probability-based interpretation *do* identify more difficult to kill mutants, there are specific instances where they fail to achieve this goal.

Summary for RQ1: Difference and probability-based interpretation can both be used to select mutants that are semantically similar to the original program. Probability-based interpretation was slightly more effective on the simple methods we evaluated from the Java Standard Library, but both techniques can be used to good effect on these methods.

B. Results for RQ2

RQ2 applies semantic interpretation to programs that are often used in software testing research (FourBalls, TriTyp and Tcas). Table IX shows the mutation score for each selection made by difference and probability-based interpretation; Table X shows the killing frequency. Figures 4, 5, 7 and 8 present the mutation score and killing frequency of each selection relative to the complete set of mutants. Finally, Figures 6 and 9 shows the Pearson correlation coefficients between mutation score / killing frequency and selection size for each program. By considering these results, we will be able to make conclusions about the effectiveness of difference and probability-based semantic interpretation on these slightly more complex programs.

Both techniques can both be used to select mutants that have a lower mutation score (and are hence considered more difficult to kill), but with different degrees of success (see Table IX). The 25% selection for difference-based interpretation had a mean mutation score of 25.4% (compared to 27.6% for the complete set), whereas it was 19.7% for probability-based interpretation. The mutation score for the probability-based technique decreased further to 18.5% for the 10% selection, whereas it actually increased to 26.7% for difference-based interpretation. This distinction between the two techniques is significantly greater than that seen in results for the Java Standard Library. This suggests the choice of interpretation technique is more important for these programs.

At first sight, the difference in killing frequency between the two selection techniques seems quite small by comparison. Difference-based interpretation achieves a mean killing frequency of 14.5% for the 10% selection size (compared to 17.2% for the complete set), whereas probability-based interpretation achieves 12.5%. Yet, the probability-based technique is much more consistent with regards to reducing the mutation score as the selection size gets smaller. When a quarter of the mutants are selected, probability-based interpretation achieves a mean killing frequency of 12.6%, whereas the difference-based technique actually increases the killing frequency to 18.0%. Overall, it is clear for the programs under evaluation that probability-based interpretation selects more difficult to kill mutants on average than difference-based interpretation.

Figures 4 and 5, 7 and 8 corroborate this finding with a stronger trend for probability-based interpretation than difference-based interpretation - selecting fewer (more semantically similar) mutants produces a lower mutation score. Probability-based interpretation reduces the mutation score and killing frequency for all three programs, whereas difference-based interpretation actually increases the frequency for certain selection sizes. Probability-based interpretation provides stronger correlations between selection size and mutation score/killing frequency. This indicates that it is more effective at determining which mutants are difficult to kill.

Probability-based interpretation achieves success partly due to its superior handling of branch conditions. Rather than assuming paths contribute equally towards a program's semantics, probability-based interpretation weights each path according to how likely it is to be exercised. Most paths through FourBalls have the same probability, whereas triangle types in TriTyp have different probabilities of occurrence. Tcas contains paths that are only exercised when action needs to be taken to avoid a collision. As a result, the difference between Pearson correlation coefficients in Figure 6 and 9 is much higher for Tcas than TriTyp or FourBalls.

Probability-based interpretation has six times the correlation coefficient for Tcas (between mutation score and selection size) than difference based-interpretation (see Figure 6). Difference-based interpretation has a negative correlation (between killing frequency and selection size) for Tcas, whereas probability-based interpretation has a strong positive correlation (see Figure 9). Probability-based interpretation performs better than difference-based interpretation on programs that rely heavily on branch conditions.

Summary for RQ2: Probability-based interpretation has been shown to be a more reliable metric of the difficulty involved with killing mutants generated from more complex programs than difference-based interpretation. It can can be used to select mutants, such that random testing achieves a lower mutation score on average than with difference-based interpretation and it is effective for all sizes of selection. Probability-based selection is particularly effective for programs that have complex branch structure (e.g. Tcas) because it takes into account the probability of executing each branch.

TABLE IX. MUTATION SCORE RESULTS

Programs	Difference-based metric				Probability-based metric			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
FourBalls	0.434	0.423	0.492	0.463	0.345	0.321	0.324	0.463
TriTyp	0.274	0.225	0.235	0.267	0.155	0.206	0.223	0.267
Tcas	0.094	0.115	0.093	0.097	0.054	0.063	0.105	0.097
Mean	**0.267**	**0.254**	**0.273**	**0.276**	**0.185**	**0.197**	**0.217**	**0.276**

TABLE X. KILLING FREQUENCY RESULTS

Programs	Difference-based metric				Probability-based metric			
	10%	**25%**	**50%**	**100%**	**10%**	**25%**	**50%**	**100%**
FourBalls	0.243	0.318	0.363	0.317	0.210	0.207	0.226	0.317
TriTyp	0.157	0.190	0.135	0.168	0.147	0.151	0.162	0.168
Tcas	0.037	0.033	0.032	0.031	0.017	0.020	0.031	0.031
Mean	**0.145**	**0.180**	**0.177**	**0.172**	**0.125**	**0.126**	**0.140**	**0.172**

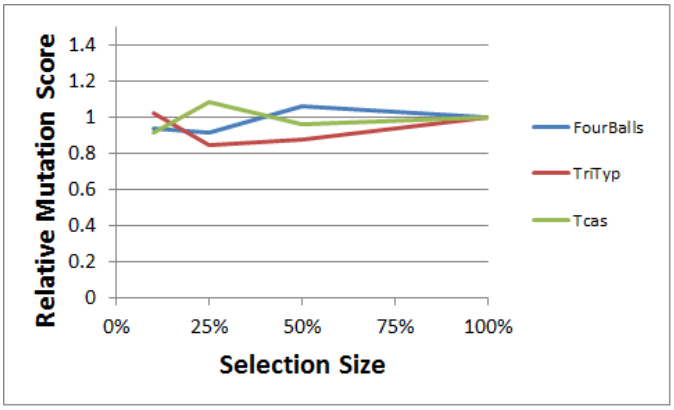

Fig. 4. Mutation Score Achieved by Difference-Based Selection

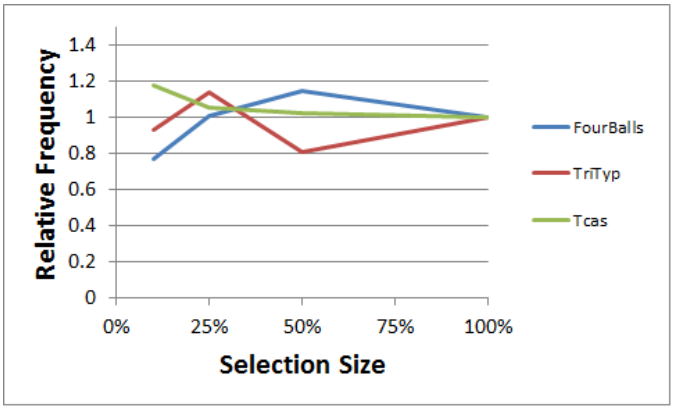

Fig. 7. Killing Frequency Achieved by Difference-Based Selection

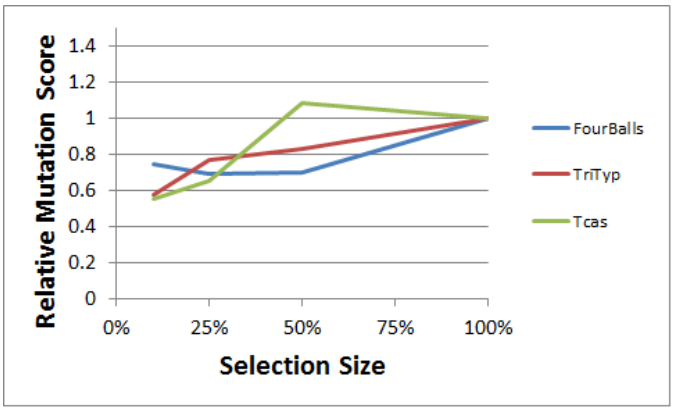

Fig. 5. Mutation Score Achieved by Probability-Based Selection

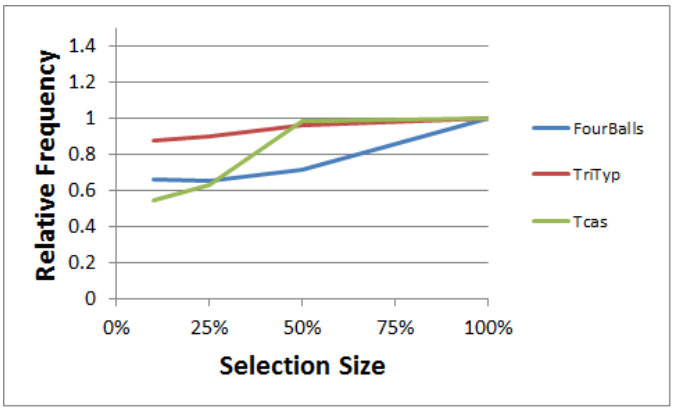

Fig. 8. Killing Frequency Achieved by Probability-Based Selection

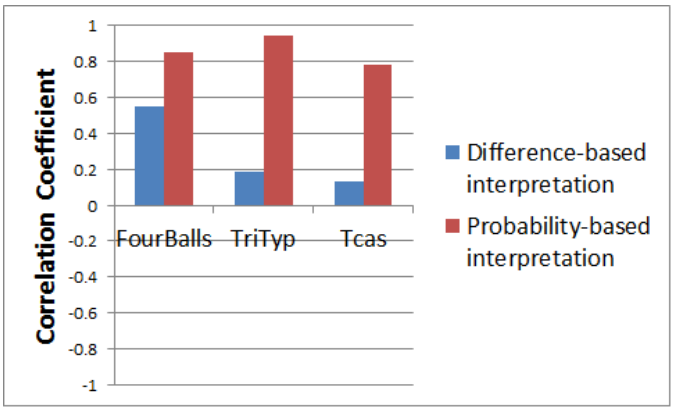

Fig. 6. Mutation Score Correlations for Each Selection Technique

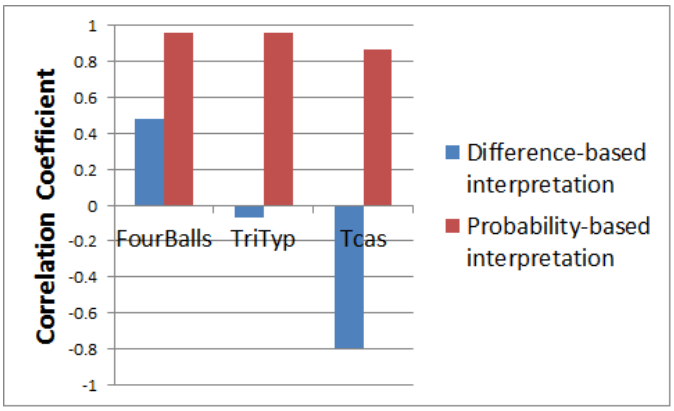

Fig. 9. Killing Frequency Correlations for Each Selection Technique

IX. Conclusions

Semantic interpretation uses static analysis to select mutants that are difficult to kill. It executes the program under test symbolically in order to determine the effect that mutations of the program code have on the relationship between inputs and outputs. The competent programmer hypothesis suggests these mutants are particularly useful for testing because they resemble faults that a programmer is more likely to make.

In our previous research, we predicted semantic similarity in terms of the differences between the range of outputs from each mutant and the original program. Our difference-based technique does not take path conditions into account and can only be applied to numerical values. In this paper, we introduced a new technique for probability-based interpretation that works by predicting the likelihood the output from a mutant is the same as the original program. Probability-based interpretation can be used on strings, Booleans, bitwise values and compound objects. It is more effective than our previous technique, particularly for more complex programs.

Difference and probability-based interpretation were both capable of selecting difficult to kill mutants for most (not all) methods evaluated from the Java standard library. Mutants in the top quarter of those selected were less than half as likely to be killed than the average mutant in the remaining three quarters. Probability-based interpretation was slightly more effective on these methods, but the difference is very small since they do not require many features from the new model.

Probability-based interpretation was much more effective at selecting difficult to kill mutants from the three slightly more complex Java programs that are used in research. For example, probability-based interpretation achieved an average 0.78 correlation between selection size and mutation score (compared to 0.13 for difference-based interpretation). Probability-based interpretation is therefore a suitable technique for selecting semantically similar mutants from programs with more complex branching structure or non-numerical types.

X. Further Work

If the output distribution for a program is sparse (as in multiplication) or skewed (as in division), semantic interpretation will underestimate the likelihood that a mutant outputs the same value as the original program. This is because there are less distinct values in the output domain, so it is more likely that two values sampled at random will be the same. This may be addressed by taking into account the shape and sparsity of the output distribution in semantic interpretation.

It is difficult to estimate the proportion of distinct outputs for a given multiplication due to its relationship with prime numbers. The best we can do is to use statistical trials and curve fitting to construct an approximate model over a particular range of input. We conducted a preliminary experiment by multiplying all pairs of numbers from ranges chosen randomly between 1 and 100. We found that the proportion of distinct outputs can be curve-fit using a logarithmic equation. The resulting model only works for the ranges included in our experiments. As it is not possible to know the range of outputs from a mutant until semantic interpretation is applied, a distribution-based approach would be expensive.

References

[1] P. G. Frankl *et al.*, "All-uses versus mutation testing: an experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, no. 3, pp. 235-253, June 1996.

[2] J. H. Andrews *et al.*, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, 2005, pp. 402-411.

[3] Y. Jia and H. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649-678, Sept. 2011.

[4] A. J Offutt and R. H Untch, "Mutation 2000: Uniting the orthogonal," in *Proc. 1st Int. Works. Mutation Analysis*, San Jose, CA, 2000, pp. 34-44.

[5] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *Proc. 3rd IEEE Int. Conf. Softw. Testing*, Paris, France, 2010, pp. 45-54.

[6] M. Daran and P. Thévenod-Fosse, "Software error analysis: a real case study involving real faults and mutations", in *Proc. 1st Int. Symp. Softw. Testing Analysis*, San Diego, CA, 1996, pp. 158-171.

[7] J. H. Andrews *et al.*, "Is mutation an appropriate tool for testing experiments?", in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, 2005, pp. 402-411.

[8] R. A. DeMillo *et al.*, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, vol. 11, no.4, pp. 34-41, Apr. 1978.

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

[10] A. J. Offutt and J. H. Hayes, "A Semantic Model of Program Faults", in *Proc. 1st Int. Symp. Softw. Testing Analysis*, San Diego, CA, 1996, pp. 195-200.

[11] M. Patrick *et al.*, "MESSI: Mutant Evaluation by Static Semantic Interpretation", in *Proc. 7th Int. Works. Mutation Analysis*, Montreal, Canada, 2012, pp. 711-719.

[12] W. Visser *et al.*, "Model checking programs", *J. Aut. Softw. Eng.*, vol. 10, no. 2, pp. 3-12, Apr. 2003.

[13] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode", in *Proc. 25th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Antwerp, Belgium, 2010, pp. 179-180.

[14] (2010, Jan.) Fujitsu Develops Technology to Enhance Comprehensive Testing of Java Programs. [Online]. Available: http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html.

[15] Y.-S. Ma *et al.*, "MuJava : an automated class mutation system", *J. Softw. Test. Verif. Rel.*, vol. 15, no. 2, pp. 97-133, Nov. 2004.

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

[17] A. J. Offutt *et al.*, "An experimental determination of sufficient mutant operators", *ACM Trans. Softw. Eng. Methodology*, vol. 5, no. 2, pp. 99-118, Apr. 1996.

[18] S. B. Dantzig, "Maximization of a linear function of variables subject to linear inequalities", in *Activity Analysis of Production and Allocation*. T. C. Koopman, Ed. New York, NY: Wiley, 1951, pp. 339-347.

[19] E. J. McCluskey, "Minimization of Boolean Functions", *Bell Syst. Tech. J.*, vol. 35, no. 6, pp. 1417-1444, June 1956.

[20] G. Redelinghuys, "Symbolic string Execution", M.S. thesis, Dept. Comp. Sci., Stellenbosch Univ., Matieland, South Africa, 2012.

[21] I. Ciupa *et al.*, "ARTOO: Adaptive Random Testing for Object-Oriented Software", in *Proc. 30th Int. Conf. Softw. Eng.*, Leipzig, Germany, 2008, pp. 71-80.

[22] A. Orso, "Integration testing of object-oriented software", Ph.D. dissertation, Scuola di Ingegneria dell'Informazione, Politecnico di Milano, Milan, Italy 1998.

[23] (2013) Java™ Platform, Standard Edition 7 API Specification. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/.

[24] M. P. Usaola *et al.*, "Reduction of test suites using mutation", in *Proc. 15th Int. Conf. Fundamental Approaches Softw. Eng.*, Tallinn, Estonia, 2012, pp. 425-438.

[25] M. Papadakis and N. Malevris, "Automatic Mutation Test Case Generation via Dynamic Symbolic Execution", in *Proc. 19th Int. Symp. Softw. Testing Analysis*, Trento, Italy, 2010, pp. 121-130.