Measuring Component Specification-Implementation Concordance with Semantic Mutation Testing

Kanat Bolazar, James W. Fawcett

Department of Electrical Engineering and Computer Science,
Syracuse University, Syracuse, NY 13244, USA
kanat2@yahoo.com, jfawcett@twcny.rr.com

Abstract

Documented specifications can aid in software verification, comprehension, maintenance and reuse. Often, specifications fall out of sync with evolving implementation. Both discovery of missing specifications and corrective maintenance of incorrect specifications require measuring the quality of specifications, not in terms of accurately representing end user needs, but rather, to answer the question of concordance "How faithfully/accurately does this specification represent the behavior of this implementation?" We implemented and tested our prototype system using Design by Contract (DBC) specifications and intermediate language (Java bytecode) manipulation. We consider the traditional source code mutation testing "syntactic mutation testing", and examine its shortcomings. We propose a different failure-injection approach, "semantic mutation testing," to measure component specification-implementation concordance. We demonstrate and compare our approach to traditional mutation testing, using a small function with multiple alternative specifications.

Keywords: software maintenance, software verification, software comprehension, Design by Contract, mutation testing, fault injection

1 INTRODUCTION

We propose a component specification quality (adequacy) criterion that measures how completely a specification represents the behavior of an implementation. Our "semantic mutation testing" approach borrows ideas from traditional ("syntactic") mutation testing [1], but instead of causing static change in code, we inject random small errors in the values used throughout processing, to perturb the dynamic state of the program.

In our earlier tests, we quickly discovered that mutants (mutated versions of the programs) often crash due to "array index out of bounds" exception. We use automated data flow analysis to avoid most cases of array index out of bounds by not mutating values used in indexing. In a correctly implemented sort function with multiple alternative specifications, our black-box and white-box approaches give better measures of specification quality compared to traditional (syntactic) mutation testing.

Section II states our central question of specification-implementation concordance and introduces DBC. Section III defines traditional (syntactic) mutation testing and its documented shortcomings. Section IV introduces semantic mutation testing. Sections V and VI explain the experiments conducted and analyze their results.

2 SPECIFICATIONS

2.1 Specification-Implementation Concordance

Software development is a structured creative process of human-machine communication and control. Often, the contract between the end user and the software developers is the requirements specification document, without which, end users would have to manually verify the implementation. Documented specifications are central to software verification and validation, and are of great importance to the tasks of software comprehension, maintenance and reuse. Human language requirements specifications have traditionally caused specification quality research to put more emphasis on human-centric criteria, mainly for comprehension and implementation of specifications [2].

Often, requirements specifications fall out of sync with evolving implementation. Some features added to the implementation may be missing in the specification, causing incompleteness of specifications, even with executable specifications that continue to validate the implementation. Maximally incomplete, an empty specification accepts (verifies as correct) any implementation. Concordance of specifications and implementation cannot be measured merely by using specifications to verify an implementation of the program. Although necessary, this is not sufficient.

If specifications are missing and need to be discovered, alternative versions of (hypotheses for) specifications may have to be compared, by measuring specification-implementation concordance to answer the question:

> *"How faithfully/accurately does this specification represent the behavior of this implementation?"*

This quality is measured against the implementation, and can only represent actual end-user need to the degree that the program is already known to perform correctly as judged by the end user. For component reuse, this is exactly the quality that we need to measure and improve.

2.2 Design by Contract (DBC)

Meyer demonstrates in [3] that human-language specifications are often incomplete, ambiguous, and inconsistent. From a six-line requirements specification by Naur that at first sight appears to be clear, unambiguous and complete, even after two published versions of supposedly corrected and complete (and four times larger) specifications by Goodenough and Gerhart [4][5], Meyer still shows errors of incompleteness, ambiguity, inconsistency and incorrectness.

Meyer's DBC specifications (called "contracts") add executable side-effect-free checks before each method to require valid inputs ("preconditions") and after, to ensure correct functional operation ("postconditions"). Every method must also preserve, upon termination, the object internal consistency conditions ("object invariants").

Consider this div method, with a human-readable but not automatically testable API documentation:

```
/** Divides n by m; m should not be 0. */
int div(int n, int m) { return n/m; }
```

A *partial contract* for div may only allow operation with nonnegative n and positive m values. This is a contract for a partial domain (input value space). We mark preconditions with @pre and postconditions with @post:

```
@pre   n >= 0  &&  m > 0
@post  $result * m >= n
@post  ($result - 1) * m < n
```

A more general *complete contract* can easily be derived by requiring these conditions on absolute values of n and m (leaving only `m != 0` as the precondition), as well as requiring consistency of sign in $result. Different syntaxes and languages can be used for DBC; for example, UML diagrams use OCL (Object Constraint Language) for such specifications. We use DBC in our experiments, but this can easily be switched with any other form of automatically verifiable specification.

Given alternative specifications, we want to measure how closely these specifications match against existing function implementation. Our specification adequacy measure is similar to (syntactic) mutation testing.

3 SYNTACTIC MUTATION TESTING

3.1 Definition

We consider the traditional mutation testing as per [1] to be syntactic mutation testing. More appropriately called "mutation analysis" (of test suites), this method does not test software, but rather provides a test criterion to evaluate test suite adequacy (quality), similar to various code coverage criteria.

For an original program, P which passes test cases T_1, T_2, ..., mutation testing uses a predefined set of mutation operators (such as '/' → '+') to create "mutants" M_1, M_2, ... which are all possible single-mutation versions of P. Mutants are compiled and checked against the test suite. If mutant M_i doesn't pass the test suite (if any test case T_j fails), we consider mutant M_i killed. Otherwise mutant remains "live".

A mutant is semantically equivalent to P if it always behaves the same way as P for any input. No test case (that passes P) could kill an equivalent mutant. *Mutation adequacy score* is the number of killed mutants divided by the number of non-equivalent mutants. If this score is 1.0, all non-equivalent mutants are killed by our test suite, and the test suite is called "mutation adequate".

3.2 Semantic Equivalence of Mutants

As an example of semantic equivalence, these two loops behave the same way so long as i is not varied unpredictably from within the loop:

```
for(i=0; i  < 10; i++) …
for(i=0; i != 10; i++) …
```

Semantic equivalence of arbitrarily complex programs is undecidable; we cannot generally know if $M_i \equiv P$. Offutt [1] suggests skipping mutant equivalence testing for some hard-to-analyze mutants to automate mutation testing. But inadequacy of test suite is discovered by mutants M_i (suspected to be $\equiv$ P) that pass all tests. Without equivalence testing, any test suite is "mutation adequate".

Actually, reading between the lines of DeMillo's seminal early work on mutation testing in 1978 [6], even in the short example of Hoare's `FIND` function, DeMillo did not check for semantic equivalence of live (unkilled) mutants. He postulates that the 14 live mutants for the final reduced test set may be equivalent, even though there can't be more than 10 equivalent mutants because an earlier larger test set had only 10 live mutants left.

As equivalence is in general undecidable, and prohibitive in practice even for short programs, the actual number of non-equivalent mutants is never truly known. Therefore the mutation adequacy score can only be approximated for nontrivial programs; it cannot be calculated.

3.3 Myths of Mutation Testing

Recently, some faulty assumptions ("myths") of mutation testing were listed and dispelled by experiments conducted in a paper on higher order (multiple mutations per mutant) mutation testing [7]. These myths are rarely stated but often presumed to be correct in mutation testing research.

"All Mutants Are Equal (AME) Myth" states that all mutated versions of a program are equally useful. This is easy to disprove: In [7], 41% of all mutants were killed by 100% of the test cases, and were called *"dumb mutants"*. For dumb mutants, any one test case is a mutation-adequate test suite all by itself! By appearing in both the numerator and the denominator, dumb mutants inflate the mutation adequacy score and give a false sense of test suite adequacy.

"Syntactic Semantic Size (SSS) Myth" states that programs generally have relatively few minor *syntactic* differences from a correct version. This is falsely derived from DeMillo's "Competent Programmer Hypothesis" [6] which states that a program written by a competent programmer will differ from a correct version by relatively few faults. But semantic differences from correct behavior (faults) are not necessarily caused by few syntactic differences; different algorithms, data structures and much refactoring may be needed for a small change in behavior.

3.4 Beyond "Dumb" Mutants

SSS myth can be dispelled with a simple example. Consider this method and its partial (incomplete) contract:

```
/*         int n0 = abs(n);
 * @post  $result <= 2*n0
 * @post  (abs(n) > 10 ||
 *         $result >= max(n0-2, 4*n0-20))
 */
int sqr5(int n) {
    if (abs(n) < 10) return n * n / 5;
    else return 20;
}
```

This is a somewhat strict specification that only allows values within `[max(|n|-2, 4|n|-20), 2|n|]` while `|n|` <= 10. Here are the actual outputs from this function compared to our contract's min and max requirements:

TABLE I. Input, Output and Contract Limits for sqr5(int n)

|n|	0	1	2	3	4	5	6	7	8	9	10
sqr5(n)	0	0	0	1	3	5	7	9	12	16	20
max	0	2	4	6	8	10	12	14	16	18	20
min	-2	-1	0	1	2	3	4	8	12	16	20

Consider the common mutation operator of replacing an arithmetic operator in {*, /, %, +, -} with another. Fig 1 shows original function and eight mutants, of which only one (12.5%), `n*n+5`, fails all tests and is a dumb mutant. Even the two mutants that are always 0 do not fail all tests.

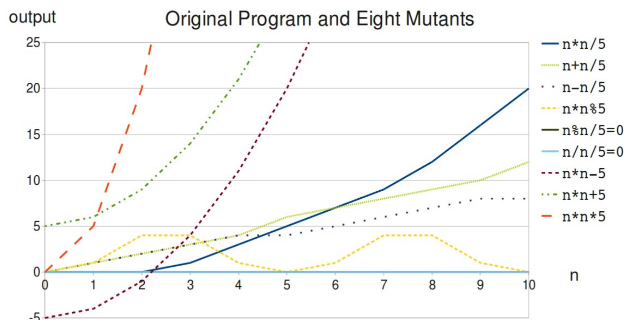

Figure 1. sqr5(int n) method (n*n/5) and eight mutants

For nontrivial test cases with n in 3..10, our mutants, on average, pass only 17.2% of the tests each. For n in {8, 9, 10}, there is no mutant that passes our contract. *Any one of these three test cases kills all mutants*. For n=10 where the correct output is 20, the output error for eight mutants are {-8, -12, -20, -20, -20, +75, +85, +480}.

Averaged over all experiments, Jia and Harman had 41% dumb mutants [7]. We only have 12.5%, even though any mutant, on average, fails 82.8% of nontrivial tests. Two-mutation mutants are worse (10 of 16 are constant-valued). Clearly, few syntactic differences do not produce programs that behave similarly. Few syntactic differences on a correct program does not produce a program with only a few faults.

4 SEMANTIC MUTATION TESTING

4.1 Introduction

Semantic mutation testing (SMT) is a fault injection method that introduces input/output (black-box) or internal (white-box) semantic state faults while a program P is being tested or verified against specifications/contracts. Analysis of syntactic mutation testing examines effects of mutation on dynamic behavior of the program. Offutt's "necessity" condition [8] states that mutation must cause a program state change to be of value. In SMT, we focus on necessity, but also attempt to limit the magnitude of program state change to have small semantic defects in our mutants.

SMT adequacy score of a contract that passes P is the number of mutant test cases not killed by the contract divided by the number of non-equivalent mutant test cases. Contracts that fail P are disqualified, and get a score of 0.0.

Even though equivalence of arbitrary programs for all possible inputs is undecidable, equivalence for one test case is easy to check: If the output of mutant M_i for the test case T_j is the same as the output of P, test case is equivalent, and could not possibly be killed by any specification/contract. This means, unlike syntactic mutation testing, SMT adequacy score can always be calculated.

4.2 Black-Box Semantic Mutation Testing

In black-box SMT, errors are introduced only to the inputs and outputs of the method. To make sure this introduces a semantic change, the ideal approach is to mutate inputs, run program P with some form of crash-handling, and mutate outputs. The mutated inputs and outputs define a mutant. Specifications with higher degree of concordance with implementation should kill (fail) more mutants.

Black-box error injection is also used in fuzz testing [9]. Fuzz testing focuses only on stress-testing the implementation for robustness (avoiding crashes) whereas black-box SMT does not test the target program itself, but rather attempts to measure functional specification adequacy. We want to avoid, not discover, cases that cause crash, so that we can conduct functional tests.

4.3 Data-Flow Analysis for Crash Prevention

Our experiments revealed that program crash caused by array index out of bounds exception is a common problem when random errors are introduced in programs. Similar to deadlock handling, we can logically prevent, dynamically avoid, or detect and recover from such critical faults. For prevention, we need white-box analysis, even for black-box fault injection. Alternatively, to dynamically avoid or recover properly, we need to instrument the implementation.

Our prototype uses a prevention approach. We start with a data flow analysis of the Java implementation using a Java bytecode manipulation library, ASM, to discover when an int type data source (variable or value) is used directly or indirectly for indexing in an array. This requires discovering possible execution paths and tracking definition-use paths that may pass through various locations in the JVM expression stack. We refrain from mutating values from such data sources, to prevent array index crashes. In the example below, if n and k are method arguments, we will not mutate their incoming values in black-box SMT; in this case, a single mutation of k → k+1 could cause a crash:

```
for(int i = 0; i < n + k; i++)
       sum += a[i];
```

Prevention is a static analysis approach, keeping execution safe without dependence on actual runtime values. This comes at a cost: While preventing crash, this also eliminates potentially safe array size and cross-element mutations within the array. Such mutations can separately be added with (array-size-aware) array mutation operators (see section 5.3). Our approach can also be extended to handle other less-frequent causes of crash and state corruption, such as division by zero and using a type beyond its valid range.

4.4 White-Box Semantic Mutation Testing

In white-box semantic mutation testing, errors can be introduced at any place where Java stack machine has values in its stack (expression stack). For example, the simple expression `a + b – c` has five spots for mutation: When local variables `a`, `b`, and `c` are read, and when expressions `(a+b)` and `(a+b–c)` are calculated. All five of these values may be mutated at the same time.

In white-box SMT, a single composite randomized mutant is generated to introduce random errors in program dynamic state. Each execution with the same inputs gives us a slightly different test case. A mutant test case is defined by inputs *and* the mutation decisions made in one execution of the program.

As there are many sites for mutation, if the probability of state change per mutation site is not very small, even a short program can quickly accumulate a large number of mutations (especially within loops) and deviate significantly from the original program's semantics. We control the standard deviation of our random value generator to limit the deviation in program state space (see third paragraph of section 5.4).

5 EXPERIMENTS

5.1 Sorting, Alternative Specifications

We tested traditional syntactic mutation testing, white-box and black-box semantic mutation testing on a sorting function with seven alternative specifications. The sort method, `sort(int[] ar)`, takes an integer array, and returns a nondecreasing sorted version of the integer array.

Seven alternative contracts (our DBC specifications) are ordered in our approximate completeness/strictness order in table II; actual order depends on input pattern frequencies. Here, bag(.) converts the collection (array) to a bag/multiset.

TABLE II. Seven Alternative Contracts for Sort Method

Contract	Postcondition
C0. NO_TEST	TRUE
C1. SAME_LEN	$result.length == ar.length
C2. SORTED_NOTEQ	∀i: $result[i] < $result[i+1]
C3. SORTED	∀i: $result[i] <= $result[i+1]
C4. SAME_VALUES	bag($result).equals(bag(ar))
C5. SORTED_LEN	SAME_LEN && SORTED
C6. SORTED_VALUES	SAME_VALUES && SORTED

C2 is too strict: It won't allow duplicate elements in the returned array, so it fails even correctly implemented sort method for 54% of our test cases. Due to this, its killed mutants score is erroneously inflated. Mutation testing disqualifies C2 for failing unmutated program.

5.2 Traditional (Syntactic) Mutation Testing

We used Jumble [10] to run traditional mutation tests on this method. Jumble uses Java bytecode rather than source code manipulation to eliminate the need for recompilation per mutation. Jumble was being used in 2007 in a continuous-test of a 370,000-line Java software every 15 minutes [10]. We ran Jumble with standard settings.

Unfortunately, Jumble did very poorly on the original code, as seen in Table III, due to finding many mutation points in the unrelated debugging/logging print statements. Even though debug option was turned off and these statements never ran in unmutated original program, Jumble gave poor adequacy scores due to our unit test not discovering changes in the debug/log code. Of the 17 mutation points Jumble found, 11 mutation points were in debugging code that never got executed. Strangely, two mutation points were supposedly even caught by no-test unit test (not theoretically possible), leaving a compressed

and misleading range of adequacy score values that gave 0.35 to the best contract and 0.11 to the no-contract case.

After we edited our code to manually remove all debugging/logging code, we were left with only 6 mutation points, of which two again were supposedly caught by the no-test contract. The adequacy score is erroneously nonzero for C0, and contracts are not distinguished by their scores.

TABLE III. Traditional (Syntactic) Mutation Test Results

Contract	Mutant Adequacy Score	
	Original Code	Edited Code
C0. NO_TEST	0.11	0.33
C1. SAME_LEN	0.11	0.33
C2. SORTED_NOTEQ	-- (error: "test class is broken")	
C3. SORTED	0.35	1.00
C4. SAME_VALUES	0.11	0.33
C5. SORTED_LEN	0.35	1.00
C6. SORTED_VALUES	0.35	1.00

In traditional syntactic mutation testing, any mutation is considered equivalent, regardless of how many times it gets executed. These tests show that considering every mutation point equal gives misleading adequacy scores.

5.3 Black-Box Tests

We randomly produced 1000 input-output sets using our correct implementation of sort method. After skipping trivial cases of arrays of size 0 and 1, we were left with 803 input-output sets. We mutated the returned sorted array using four array mutation operations:

`swap`: swaps two elements of array ar

`replace`: replaces an element of array ar with a randomly picked element from another array of similar values

`resize`: either duplicates an element of array to grow the array by one element, or removes one element to shrink the array by one element.

`random`: replaces an element of array ar with a random int value

For each of the 803 input-output sets, we produced 200 mutated versions of the output array using 1 to 30 mutations per mutant, for a total of 160,600 mutant test cases. Comparing with correct output from unmutated program, we found 146,179 of these to be non-equivalent mutant test cases. Table IV shows our test results.

Recall that C2 is too strict to even accept correct implementation; this behavior inflates the number of mutants killed by C2, as it even kills some equivalent mutants. In this case, the ordering of semantic mutation adequacy scores corresponds to our prior belief of how complete each specification is. As C6 is a complete specification (modulo invariants), its mutation adequacy score is 1.0.

TABLE IV. Black-Box Semantic Mutation Test Results

Contract	Mutants Killed	Adequacy Score (Proportion)
C0. NO_TEST	0	0.00
C1. SAME_LEN	82,245	0.56
C2. SORTED_NOTEQ	141,838	-- (0.00)
C3. SORTED	103,385	0.71
C4. SAME_VALUES	125,597	0.86
C5. SORTED_LEN	132,531	0.91
C6. SORTED_VALUES	146,179	1.00

5.4 White-Box Tests

We ran our data flow analysis to prevent mutation of values that may cause array index out of bounds exception, using any array access and new array creation operations as our targets. For example, whatever value or variable reaches top of the JVM expression stack for an IALOAD (int array load) instruction should never be mutated. At all remaining sites of integer value operation in the JVM stack machine, we modified the compiled Java bytecode to insert a call to our stateless mutater method to inject error to the int value on top of the expression stack. We didn't need access to source code or recompilation to produce this composite randomized mutant.

The amplitude and frequency of mutations can be numerically adjusted in semantic mutation testing. We used integer-rounded Gaussian distributions as our additive error terms, with two different standard deviations; $\sigma = 0.2$ and $\sigma = 0.5$. Values in (-0.5, 0.5) get rounded to 0 and do not cause any state change. This happens 98.76% of the time with $\sigma = 0.2$, and 68.27% of the time with $\sigma = 0.5$. These two tests are significantly different; the $\sigma = 0.5$ case has nonzero error added about 25.6 times more often than the $\sigma = 0.2$ case.

The results for both values of standard deviations are shown in table V. For tests with $\sigma = 0.2$, 812 non-trivial cases with 200 mutant runs each gave us 162,400 test cases, of which only 7,095 were non-equivalent. For tests with $\sigma = 0.5$, 787 non-trivial input-output sets each with 200 mutant runs each gave us 157,400 test cases, of which 102,868 were non-equivalent. In both cases, the faulty contract C2 failed more mutants than there were non-equivalent mutants.

As we do not mutate int values that are directly or indirectly used in array indexing, the length of the array never changes, and two pairs of contracts that differ only in checking array length produce exactly the same values:

C0 and C1 both kill (fail) no mutants, and C3 and C5 always kill the same number of mutants.

In this case, the limitations of our mutation operators cause sortedness to be considered more easily satisfiable then preserving the bag of values from the input array. This is understandable, as any one mutation to any of the values during the execution will always change the bag of values, but may not change their order (and sortedness).

TABLE V. White-Box Semantic Mutation Test Results

Contract	Mutant Adequacy Score	
	σ = 0.2 (7,095 $M_i \not\equiv P$)	**σ = 0.5 (102,868 $M_i \not\equiv P$)**
C0. NO_TEST	0.00	0.00
C1. SAME_LEN	0.00	0.00
C2. SORTED_NOTEQ	-- (117,442 killed)	-- (109,729 killed)
C3. SORTED	0.37	0.47
C4. SAME_VALUES	0.86	0.92
C5. SORTED_LEN	0.37	0.47
C6. SORTED_VALUES	1.00	1.00

6 DISCUSSION AND FUTURE WORK

We have shown the feasibility of measuring specification quality by how often specification fails programs with small semantic errors. Compared to traditional (syntactic) mutation testing, both our white-box and black-box semantic mutation testing approaches produce adequacy scores that better represent the concordance of the specification with our correct implementation. Our data-structure-aware black-box mutation operations gave better results compared to primitive-value-aware single mutation operator for integer values injected into method implementation (white-box).

Our crash prevention avoids any mutations that could have an effect in array sizes and which element is accessed. Other mutation operators such as the array-mutation operators used for black box testing can also be introduced to accessed arrays, after proper data flow analysis to prevent array index crashes. As always, the mutation operators will be most useful if they represent common types and patterns of faults.

As an alternative to randomly created test cases as seen in our experiments, we can use any existing test suites, or consider gathering test data in situ, by saving input-output sets (using serialization for reference types, objects) from a component while the software system is running.

To the degree that the implementation itself is tested and known to conform to user's needs, our semantic mutation specification adequacy score also measures how well the specifications match actual user requirements.

We believe this simple-to-compute measure of specification-implementation concordance can help automate measuring quality of suspected-to-be-outdated as well as rediscovered or competing specifications. We plan to use this method in evolving component specifications, in a tool that assists program comprehension and discovery of missing specifications.

Source code and examples from this paper are available at our Semantic Mutation Testing page [11].

7 REFERENCES

[1] A. J. Offutt, R. H. Untch, "Mutation 2000: Uniting the Orthogonal", in Mutation 2000: Mutation Testing For the New Century, W. E. Wong, Ed, Kluwer International Series on Advances in Database Systems, vol. 24. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 34-44.

[2] A. Davis, S. Overmyer, K. Jordan, et al, “Identifying and Measuring Quality in a Software Requirements Specification,” pp. 141–152 in Proceedings of METRICS ’93, Baltimore, MD, May 1993.

[3] B. Meyer, "On Formalism in Specifications," IEEE Software, vol. 2, no. 1, pp. 6-26, January/February 1985.

[4] J. B. Goodenough, S. Gerhart, "Towards a Theory of Test Data Selection," IEEE Trans. Software Engineering, vol. SE-1, no. 2, pp. 156-173, June 1975.

[5] J. B. Goodenough, S. Gerhart, "Towards a Theory of Test: Data Selection Criteria," in Current Trends in Programming Methodology, vol. 2, R. T. Yeh, Ed, Prentice-Hall, Englewood Cliffs, NJ, 1977, pp. 44-79.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” Computer, vol. 11, no. 4, pp. 34–41, April 1978.

[7] Y. Jia, M. Harman, "Higher Order Mutation Testing," Information and Software Technology, vol. 51, no. 10, pp. 1379-1393, October 2009.

[8] R. A. DeMillo, A. J. Offutt, "Constraint-Based Automatic Test Data Generation," IEEE Trans. Software Engineering, vol. 17, pp. 900-910, September 1991.

[9] B. P. Miller, L. Fredriksen, B. So, "An empirical Study of the Reliability of UNIX Utilities," Communications of the ACM, vol.33, no.12, pp.32-44, December 1990.

[10] S. A. Irvine, T. Pavlinic, L. Trigg, J.G. Cleary, S. Inglis, M. Utting, "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests ," pp. 169-175 in Proceedings of TAICPART - MUTATION 2007.

[11] http://www.ecs.syr.edu/faculty/fawcett/handouts/webpages/research/Bolazar/SMT/