Efficient Mutation Testing by Checking Invariant Violations

David Schuler, Valentin Dallmeier, and Andreas Zeller
Saarland University, Saarbrücken, Germany
{ds, dallmeier, zeller}@cs.uni-saarland.de

ABSTRACT

Mutation testing measures the adequacy of a test suite by seeding artificial defects (mutations) into a program. If a mutation is not detected by the test suite, this usually means that the test suite is not adequate. However, it may also be that the mutant keeps the program's semantics unchanged—and thus cannot be detected by any test. Such *equivalent mutants* have to be eliminated manually, which is tedious.

We assess the impact of mutations by checking dynamic invariants. In an evaluation of our JAVALANCHE framework on seven industrial-size programs, we found that mutations that violate invariants are significantly more likely to be detectable by a test suite. As a consequence, mutations with impact on invariants should be focused upon when improving test suites. With less than 3% of equivalent mutants, our approach provides an efficient, precise, and fully automatic measure of the adequacy of a test suite.

Categories and Subject Descriptors

D.2.5 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Experimentation

Keywords

Dynamic Invariants, Mutation Testing

1. INTRODUCTION

How do we know a test suite is adequate in finding defects? Among the best ways is *mutation testing:* seeding defects into a program and checking whether the test suite finds them. Such defects can be created automatically, using a set of *mutation operators* to change ("mutate") random program parts. A mutation that is not detected ("killed") by the test suite indicates that the test suite was unable to detect the seeded defect—and therefore is likely to miss similar, true defects in the code. Test managers can use such results to improve their test suites such that they detect these mutants.

Mutation testing has been shown to be an effective measurement for the quality of a test suite [2] and superior to commonplace assessments such as coverage metrics [27, 8]. However, mutation testing is known to be very *expensive.* A well-known issue is its large usage of computing resources. A less known, but far more significant cost, though, comes from the problem of *equivalent mutants.* These are mutants that leave the program's overall semantics unchanged—and therefore cannot be caught by any test suite: The result of mutation testing—"surviving" mutations not found by the test suite—thus mixes the most valuable and the least valuable mutations in one set. Therefore, when one assesses surviving mutants, one must first eliminate equivalent mutants. This problem is widespread; its solution is tedious. In an experiment on a sample of 20 random mutations on the 12,000-line JAXEN program [9], we found 40% of the non-detected mutations to be equivalent. On average, it took us 30 minutes to assess one single mutation for equivalence. If we had assessed *all* 4,110 non-detected mutations, this task would have cost us 2,055 hours, or two person-years; actually, it can be assumed that the ratio of equivalent mutants *increases further* as we improve the test suite.

The problem of assessing mutation equivalence has been noted before, but rarely quantified. Frankl et al. [8] state:

> *Although our experiments were designed to measure effectiveness, we also observed that using these criteria, particularly mutation testing, was costly. Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was almost prohibitive.*

In the past, a number of efforts have been attempted to reduce the cost of equivalent mutants. Various *heuristics* based on mutant similarity have been suggested [3]. Static program analysis, in particular *path constraints,* can detect many cases of equivalent behavior [20]. *Program slicing* can assist in narrowing down the impact of a mutation [11]. *Genetic algorithms* have been suggested to specifically evolve mutants detected by at least one test case [1]. None of these techniques has yet been shown to scale to large programs.

In this work, we suggest an alternate, novel way to eliminate equivalent mutants. Our approach is based on the assumption that a non-equivalent mutant must impact not only the program code, but also the program behavior—just

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-338-9/09/07 ...$5.00.

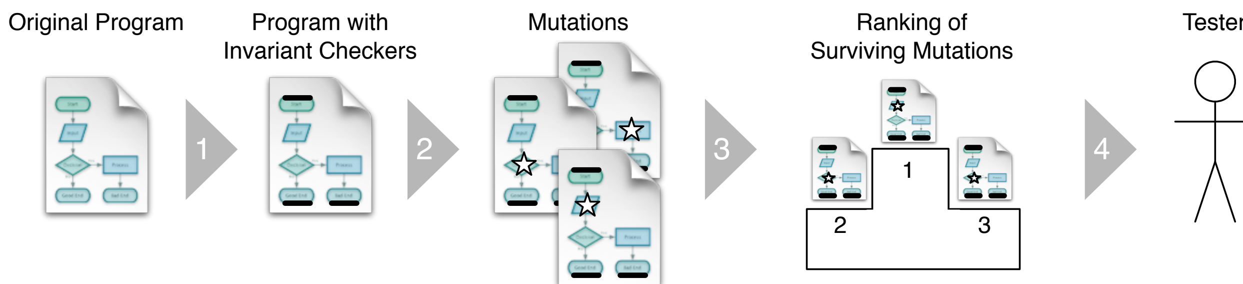


Figure 1: The JAVALANCHE process. After learning dynamic invariants from test suite execution, JAVALANCHE instruments the program with invariant checkers (Step 1), generates mutations (Step 2), runs the test suite on each mutation (Step 3), and ranks mutations by the number of violated invariants. Finally, the tester (Step 4) improves the test suite to detect the top-ranked mutations.

like a defect must impact program execution to produce a failure. To characterize the "normal" behavior, we use *dynamic invariants,* learning pre- and postconditions for every function as observed during its executions. Our hypothesis is that a mutation which violates such invariants causes different behavior—and therefore is much more likely to also violate the program's semantics than a mutation which satisfies all learned invariants. We actually propose that testers focus on those mutations that violate the *most* invariants (i.e. have the greatest impact on program behavior), and still are not caught by the test suite. Our framework, named JAVALANCHE, is summarized in Figure 1.

In this paper, we explain how to assess the impact of mutations using dynamic invariants, and we demonstrate the usefulness of our approach by empirically showing that mutations that violate invariants are much less likely to be equivalent. After giving a motivating example (Section 2) and introducing our mutation framework (Section 3), the paper makes the following contributions:

1. We present a scalable, effective, and efficient method to learn the most common invariants (Section 4) and to check them (Section 5).

2. We demonstrate a novel method for classifying and ranking the impact of changes (mutations) on invariants (Section 6).

3. Our evaluation shows that focusing on mutations with the highest impact has a low ratio of equivalent mutants (Section 7)—making mutation testing efficient and applicable in practice.

After discussing the related work (Section 8), Section 9 closes with conclusion and consequences.

2. EXAMPLE: MUTATING ASPECTJ

When conducting mutation testing, one eventually wants the test suite to detect all mutants. There are multiple reasons why a mutant may not be detected, though:

1. It may *not be executed.* This problem can be addressed by standard test coverage criteria.

2. It does not *impact the program semantics.* These are the ones that are hard to assess.

3. The test suite is *unable to detect it.* These are the mutants that help us in improving the test suite, and the ones we actually look for.

```
public int compareTo(Object other) {
  return 0;       0 ⇒ 1
}
```

Figure 2: A mutation that is not executed. The defect induced by the change from 0 to 1 is never executed (or tested) in ASPECTJ.

Let us present a few mutations to the ASPECTJ compiler, as applied and assessed by our JAVALANCHE framework. None of these mutations are caught by the test suite.

Most Java objects implement a `compareTo()` method, comparing individual objects. However, the `Checker.compareTo()` method in Figure 2 always returns zero. When JAVALANCHE mutates it to return a value of one instead—how will this impact the ASPECTJ compiler? In the entire ASPECTJ source code, there are no less than 165 occurrences of "`compareTo`"; also, `compareTo()` could be called from within built-in Java classes such as sorted containers.

Fortunately, we do not have to check all these occurrences—simply because `Checker.compareTo()` *is never executed* (neither by the test suite nor the whole program) and therefore, no mutation will ever have any impact on any execution. Non-executed code can be uncovered by mutation testing, as in this example, but simple statement coverage does so much more efficiently. In the remainder of this paper, we assume that mutations are applied only to code executed during test.

In Figure 2, one might have thought that changing the return value would always be a non-equivalent mutant, but this is not the case. Figure 3 shows the `compareTo()` method of the ASPECTJ `BcelAdvice` class, where JAVALANCHE also increments one return value. This code is executed, so we actually need to check all explicit (and implicit) call sites.

Callers of Java `compareTo()` methods are required to only check for the sign of the return value. It turns out that the usages of `compareTo()` in ASPECTJ are no exception. Therefore, the mutation from `+1` to `+2` does not change the semantics of the program—this is a "classic" equivalent mutant. However, a single equality check for the return value `+1` rather than for an arbitrary positive value would make `compareTo()` non-equivalent. This is why all call sites have to be evaluated.[1]

[1]Obviously, the problem could be alleviated by precisely specifying the contract of `BcelAdvice.compareTo()`—does it guarantee to return `+1`, or just a positive number?

```
public int compareTo(Object other) {
  if (!(other instanceof BcelAdvice))
    return 0;
  BcelAdvice o = (BcelAdvice)other;

  if (kind.getPrecedence() !=
      o.kind.getPrecedence()) {
    if (kind.getPrecedence() >
        o.kind.getPrecedence())
      return +1;    +1 ⇒ +2
    else
      return -1;
  }
  // More comparisons...
}
```

Figure 3: A mutation that has no impact. All users of `BcelAdvice.compareTo()` only check for the arithmetic sign of the return value, making this an equivalent mutant.

```
public LazyMethodGen getLazyMethodGen(String name,
    String signature, boolean allowMissing) {
  for (Iterator i = methodGens.iterator(); i.hasNext();){
    LazyMethodGen gen = (LazyMethodGen) i.next();
    if (gen.getName().equals(name) &&
          ⇒ !  (gen.getSignature().equals(signature)))
      return gen;
  }
  if (!allowMissing)
    throw new BCException("Class " + this.getName() +
        " does not have a method " + name +
        " with signature " + signature);
  return null;
}
```

Figure 4: A mutation that is not detected. This mutation in the `LazyClassGen` class changes 39 invariants of 18 methods, but is not caught by the ASPECTJ test suite.

Finally, a trickier example. In Figure 4, we see the method `getLazyMethodGen()` of the `LazyClassGen` class in ASPECTJ. This method takes a method name and signature and returns a `LazyMethodGen` object with the same name and signature, or null. The mutation negates a condition and thus changes the behavior such that a `LazyMethodGen` is only returned when the names match but the signatures do not.

This change impacts not only the behavior of the method itself, but also violates the inferred pre- and postconditions of at least 18 other methods scattered all throughout the program. Yet, this mutation is not detected by any test, implying that the ASPECTJ test suite is not adequate with respect to such defects. Although a test triggers this behavior, it fails to check for the caused error. As a consequence, a test manager should extend the test suite such that this mutation would be detected, too.

3. GENERATING MUTANTS

Our original motivation for mutation testing was assessing the adequacy of test suites of large-scale programs. Of the existing mutation tools such as μJava [14], none met our requirements in terms of *automation* and *scalability*. We therefore decided to implement our own mutation engine as the core of JAVALANCHE.

The key features of our implementation are:

Table 1: JAVALANCHE mutation operators

Operator	Description
Replace numerical constant.	Replace a numerical constant X by $X+1$, $X-1$, or 0.
Negate jump condition.	Replace a conditional jump by its counterpart. This is equivalent to negating a conditional statement or subexpression in the source code.
Replace arithmetic operator.	Replace an arithmetic operator by another one, e.g. $+$ by $-$.

Focus on sufficient mutation operators. The idea of *selective Mutation* is to use a small set of mutation operators that is a sufficiently accurate approximation of the results obtained by using all possible operators [19]. JAVALANCHE therefore uses the same small set of operators as proposed by Offutt [19] and later adapted by Andrews et al. [2], listed in Table 1.

Use mutant schemata. Traditional mutation testing tools produce a new mutated program version for every applicable mutation possibility. For a system like ASPECTJ, this would result in 47,146 different mutated versions, which are too many to be handled effectively. To reduce the number of generated versions, we use *mutant schema generation* [26]. Mutant schema generation produces a *metaprogram* that is derived from the program under study and contains multiple mutations. Each mutation is guarded by a conditional statement that can be switched on and off at runtime.

Use coverage data. Not all tests in the test suite execute every mutant. In order to avoid executing those tests, we collect *coverage information* for each test. When checking mutants, we execute only those tests that are known to cover the mutated statement.

Since JAVALANCHE works directly on Java byte code, it also avoids costly recompilation. As shown in Section 7, JAVALANCHE easily scales up to large-scale programs like ASPECTJ. The overall development effort into JAVALANCHE (including learning and checking invariants—see Sections 4 and 5) was 12 person-months.

4. LEARNING INVARIANTS

To deduce invariants, JAVALANCHE relies on the invariant detection engine of DAIKON [7]. Invariants are learned in three steps: First, we run an instrumented version of the program and collect a trace of all parameter and field accesses. Second, we analyze the trace and generate input files for DAIKON. Finally, we feed those files into DAIKON to deduce invariants.

Tracing. JAVALANCHE uses ASM [13] to instrument Java classes. It injects code that writes interesting events such as field accesses, beginning and end of methods into a compact trace file. On average, our current JAVALANCHE implementation produces ~30 MB/s of trace data.

DAIKON analyzes Java programs through CHICORY, a front-end for tracing Java programs. CHICORY uses *all* variables in the scope of a method, regardless of

Table 2: Invariants used by JAVALANCHE.

Unary invariants. Compare a one-word variable (any Java type other than `long`, `float`, or `double`) X against $X \neq 0$, $X \neq$ `null`, $X \leq c$, $c \leq X$, $c_1 \leq X \leq c_2$, where c, c_1, c_2 are constants.

Binary invariants. Compare two one-word variables X_1 and X_2 against $X_1 = X_2$, $X_1 > X_2$, $X_1 < X_2$.

Strings and other objects are only checked for being `null`.

whether they were accessed or not. Consequently, the invariant detection engine has to deal with a lot more data, which lead to out-of-memory errors for some of the larger subjects in our case study. In contrast, JAVALANCHE uses only those variables that were *actually accessed by a method,* which greatly reduces the amount of data to be analyzed and allows for learning invariants even from very large programs.

Generating Daikon Files. In the second phase, for every method that is invoked at least once, JAVALANCHE generates program point declarations for the beginning (*ENTER*) and the end (*EXIT*) of the method. For every method invocation, JAVALANCHE generates DAIKON trace entries for the corresponding *ENTER* and *EXIT* program points.

Running Daikon. The generated files are then fed into DAIKON, which supports over 85 different types of invariants. For performance reasons, we decided to limit DAIKON to those types of invariants that occur most frequently in practice. In a small experiment using a total of 94 different runs of our subject programs, we detected those DAIKON invariants that occurred most frequently. Our current configuration uses 28 different types of invariants, summarized in Table 2, accounting for over 95% of all invariants found in our sample.

As a consequence, JAVALANCHE easily handles large programs like ASPECTJ with reasonable efficiency.

5. CHECKING INVARIANTS

JAVALANCHE learns invariants from the *unmutated* program, and checks for violations in the *mutated* program. To check for invariant violations of the mutations, we use a runtime checking approach that is similar to the approach pioneered by DIDUCE, a tool to learn invariants and detect violations at run-time [10].

For each learned invariant, we insert statements into the bytecode that check for invariant violations before and after a method. If an invariant is violated, this is reported and the run resumes. All this allows for efficient and scalable checking of invariants.

6. CLASSIFYING MUTATIONS

The output of JAVALANCHE is a ranked list of the mutations applied to a program. For each mutation, the output contains the following information:

Detectability. A flag indicates if the mutant was detected ("killed") by the test suite. A mutant is considered detected if at least one test fails, runs into a timeout, or throws an exception.

Impact. Each invariant represents a different property of "normal" program runs. The more properties violated, the higher the impact of the mutation on the program execution. We therefore use *the number of invariants violated* by a mutation to measure impact; the greater the impact of a mutation, the higher the ranking.

In this work, we are only interested in mutants that are not detected by the test suite. Using JAVALANCHE, we split this set of "surviving" mutants into two:

Non-Violating mutants (NVM) Mutants that do not violate any invariant. These mutants do not impact the program with respect to its dynamic invariants.

Violating mutants (VM) Mutants that violate at least one invariant.

The idea behind our approach is that violating mutants are less likely to be equivalent since they violate the typical behavior of the program as captured by invariants. This assumption will be investigated in the following section.

7. EVALUATION

In order to evaluate our approach, we have conducted three different experiments. In Section 7.2, we manually assessed a sample result for equivalent mutants. In Section 7.3, we compare the detection rates of mature test suites for invariant-violating mutants (VM) and non-violating mutants (NVM). Section 7.4 investigates whether the mutants with the highest impact have the highest detection rate, implying a low number of equivalent mutants.

7.1 Evaluation Subjects

For the evaluation of the JAVALANCHE framework, we used the seven open-source projects listed in Table 3. Each program comes with a regression test suite. We removed all tests that fail on the original version, as well as tests whose outcome depends on the order of test execution. The number given in Table 3 is the number of tests we conducted.

Table 4 gives the results for mutation testing the subject programs. The second column shows the *total number of mutations,* ranging from 1,533 for JTOPAS to 47,146 for ASPECTJ. The *coverage rate* of mutations, given in the third column, indicates how many of the mutations actually were executed. For most projects, it varies between 60–80%.[2]

The *mutation score* is the percentage of mutations detected by the test suite, as given in the rightmost column. A low mutation coverage implies a low mutation score; as discussed in Section 2, a mutation not executed cannot be detected by a test suite. Thus, we also give the *mutation score for the covered mutations.* In ASPECTJ, 53% of the executed mutations are detected, with the rate going up to 92% for XSTREAM.

Table 5 shows the time required by JAVALANCHE to perform the steps discussed in the previous sections. Columns 2–4 list the steps required to learn invariants. The dominating step in terms of runtime is usually mining invariants

[2]For ASPECTJ, this rate is lower, since we only executed the tests for the `core` package. The low coverage rate for BARBECUE is due to the fact that one class accounts for 11,759 mutations alone, consisting mainly of the static initializations of maps for bar code values.

Table 3: Description of subject programs.

Project Name	Description	Version	Program size (LOC)	Test code size (LOC)	Number of tests	Test suite runtime (s)	Statement coverage (%)
ASPECTJ	AOP extension to Java	1.6.1	94,902	14,736	321	21	33,15
BARBECUE	Bar code creator	1.5b1	4,837	3,136	137	3	47.26
COMMONS	Helper utilities	2.5-S	18,782	31,940	1,590	19	85.08
JAXEN	XPath engine	1.1.1	12,449	8,371	680	10	66.79
JODA-TIME	Date and time library	1.5.2	25,861	47,227	3,447	12	51.33
JTOPAS	Parser tools	1.0(SIR)	2,031	3,185	128	3	80.68
XSTREAM	XML object serialization	1.3.1	14,480	13,505	838	10	75.78

Lines of Code (LOC) are non-comment, non-blank lines as reported by `sloccount`. For ASPECTJ, we only considered the core package tests.

Table 4: Mutation statistics for the subjects of our evaluation.

Project Name	Number of Mutations	Mutation Coverage (%)	Mutation Score (%)	Mutation Score for covered mutations (%)
ASPECTJ	47,146	30.31	16.02	52.95
BARBECUE	17,178	4.73	3.18	67.28
COMMONS	15,125	73.93	61.22	82.81
JAXEN	6,712	61.23	37.17	60.71
JODA-TIME	13,859	69.59	54.81	78.75
JTOPAS	1,533	76.65	54.99	71.74
XSTREAM	5,186	69.42	63.84	91.97

with DAIKON. Columns 5–8 list the steps required to instrument and check mutated versions of the program. The rightmost column gives the total time needed to evaluate each subject. Our record holder is again ASPECTJ, with a one-time effort of 32 CPU-hours of learning invariants and creating checkers, and another 14 CPU-hours for running the mutation test.[3] To our knowledge, this is the first time mutation testing has been applied to a program of this size.

7.2 Are mutants that violate invariants less likely to be equivalent?

We started our experiments with a manual assessment of invariant-violating mutants vs. non-violating mutants.

7.2.1 Hypothesis

Our hypothesis was:

H1 *Mutants that violate invariants are less likely to be equivalent than mutants that do not violate invariants.*

7.2.2 Experimental Setup

As discussed in Section 1, manually checking if a mutant is equivalent requires a lot of effort. We therefore restricted our evaluation to one project (JAXEN) and twelve samples for each group. The non-violating mutants were chosen randomly; the twelve violating mutants were those that had the highest number of violations.

For each mutant, we first examined the source code around the mutant; we then tried our best to come up with a test case to trigger the mutant. If it was not possible to come up with such a test, we considered the mutant to be equivalent. Assessing the 24 mutants took us 12 person-hours (i.e., 30 minutes per mutant on average), plus additional time to write test cases for non-equivalent mutants.

[3]All times were measured on a 16-core 2.0GHz AMD Opteron 870 machine with 32 GB RAM; we used up to 7 cores and 2 GB RAM per core. Note that CPU time does not depend on the number of cores.

7.2.3 Results

The results of this manual inspection indicate that mutants that violate invariants are more than *twice as likely to be non-equivalent:*

Violating Mutants. For 10 out of the 12 inspected mutants ($10/12 = 83\%$), we could write a test that triggers the mutant. Most of these mutations impacted tracking input line numbers, a feature not covered by the JAXEN test suite. We failed to write tests for the remaining two mutants and therefore consider them to be equivalent.

Non-Violating Mutants. We were able to write tests for only 4 out of the 12 mutants ($4/12 = 33\%$), and failed to do so for 7 of the remaining mutants. We did not categorize one mutant, since the behavior of the mutant depends on the system locale.

A Fisher Exact Value test confirms the statistical significance of the difference at $p = 0.036$. We therefore accept our hypothesis **H1**—with a grain of salt, as the size of our sample set was small.

> *In our sample,* ***mutants that violate several invariants are less likely to be equivalent.***

7.2.4 Qualitative Analysis

In Figure 4, we already had seen an undetected ASPECTJ mutation that violates several invariants; as discussed in Section 2, we actually found comments that indicate a lack of testing for this ASPECTJ class. Inspired by this example, we also did a qualitative analysis of the JAXEN mutations, checking how addressing them would improve the JAXEN test suite.

The mutation with most impact on invariants is in line 102 of class `org.jaxen.Context` (Figure 5). The mutation sets the initial size of the context to -1 instead of 0. This violates several preconditions for methods that use this size information during their computation, e.g. that the size is

Table 5: JAVALANCHE runtime (in CPU time) for the individual steps.

Project Name	Run instrumented test suite	Create Daikon files	Learn invariants (Daikon)	Create and test checkers	Scan mutants and collect coverage data	Check mutated versions	Total time
ASPECTJ	197s	3,716s	81,416s	34,477s	271s	51,730s	47h 43m
BARBECUE	17s	20s	985s	25s	142s	2,535s	1h 02m
COMMONS	183s	752s	12,200s	388s	186s	8,765s	6h 14m
JAXEN	288s	7,274s	19,854s	789s	2,980s	7,514s	10h 44m
JODA-TIME	276s	769s	49,068s	1,132s	2,745s	11,002s	18h 03m
JTOPAS	164s	3,134s	6,086s	81s	931s	2,649s	3h 37m
XSTREAM	172s	1,561s	18,160s	1,444s	6,445s	9,811s	10h 26m

Time reported is the time used to execute the steps ("user time") measured using `time`;
System overhead ("system time") is not included.

```
public Context(ContextSupport contextSupport)
{
     this.contextSupport = contextSupport;
     this.nodeSet = Collections.EMPTY_LIST;
     this.size = 0;    0 ⇒ -1
     this.position = 0;
}
```

Figure 5: A non-detected JAXEN mutation that violates the most invariants.

```
private Token plus()
{
  Token token = new Token(TokenTypes.PLUS,
      getXPath(),
      currentPosition(),
      currentPosition()+1 );    1 ⇒ 0
  consume();
  return token;
}
```

Figure 6: A non-detected JAXEN mutation that violates the second most invariants.

greater than zero or in a certain range. However, while the test suite checks the size of the underlying node set (whose size seems to be tracked by `Context`'s size field) after creation of a `Context` object, it does not check the size of the `Context` itself (in Test `ContextTest`). Thus, this is either an insufficiency in the test suite, or a *code smell,* since the size may always be computed from the underlying node set.

The mutation with the second largest invariant impact can be found in line 615 of class `XPathLexer` (Figure 6). It sets the end index of a plus token to the same value as the begin index, making it a token of size 0. This leads to several invariant violations that involve the `tokenEnd` field of the `Token` class. The mutation, for example, has an effect on the program whenever a string representation of this token is requested, e.g. in error messages for wrong XPATH expressions. While the test suite checks for the *errors,* it does not check *the expressions enclosed in error messages.* Such checks take place for many other JAXEN messages, though; thus, the above mutation again indicates ways to improve the test suite.

7.3 Are mutations that violate invariants more likely to be detected by actual tests?

The manual effort required for assessing mutations is not only a problem in mutation testing itself; it is also a problem when *evaluating* mutation testing approaches. In particular, the precise rate of equivalent mutants can only be measured by assessing all mutations manually, which is precisely the problem we want to overcome. Therefore, for our second experiment, we wanted to have an objective classification that could be more easily automated.

7.3.1 An Indirect Evaluation Scheme

How do we detect that a mutation is non-equivalent? In practice of mutation testing, this is done all the time: By having the *test suite* detect the mutation. Any mutation detected by the test suite, by definition, alters the program semantics—and thus is non-equivalent. This fact that "detected" implies "non-equivalent" leads to our key idea:

> **A mutant generation scheme whose mutants are *more frequently detected* by the test suite also produces *fewer equivalent mutants.***

For mutation testing, we are not interested in detected mutants, though; what we want is non-detected mutants, as these help us to improve the test suite. But if a scheme *generally* produces fewer equivalent mutants, this property should hold whether the test suite detects them or not. Hence, the ratio of equivalent mutants can be expected to be lower in the set of undetected mutants as well.

In our case, the mutant generation scheme favors those mutations with impact on invariants. If we can show that impact on invariants correlates with detection by the test suite, this means that impact on invariants also correlates with non-equivalence, as non-equivalence is implied by test suite detection. In formal terms, we have an implication

$$\textit{detected by test} \implies \textit{non-equivalent}$$

and if we can show (statistically) that

$$\textit{violates invariants} \stackrel{?}{\implies} \textit{detected by test}$$

then we would conclude that

$$\textit{violates invariants} \implies \textit{non-equivalent}$$

This indirect evaluation approach relies on the assumption that test suites, as they stand, are already good detectors

of changed behavior. This assumption also was the base of previous studies [2, 17]; there is no reason to believe it would not hold for our experiment subjects. (For actually *applying* JAVALANCHE, rather than evaluating it, such a mature test suite is not required; instead, it is our aim to achieve this level of maturity.)

7.3.2 Hypothesis

Given our limited set of mutations, the constrained range of invariants checked, and the complexity and richness of the test suites involved, it is not obvious at all that invariant violations would correlate with test outcome. The hypothesis to be checked in our experiment was thus:

H2 *Mutants that violate invariants are more likely to be detected by actual tests.*

7.3.3 Experimental Setup

Our setup for **H2** is straight-forward: We execute the test suite to learn dynamic invariants, identify surviving mutants and classify them as violating or non-violating.

7.3.4 Results

Our results are summarized in Table 6. Let us compare the detection rate of invariant-violating mutants (VMs) versus non-violating mutants (NVM) (columns 4 and 5). With the exception of ASPECTJ, all projects show a higher detection rate for VMs. The difference is statistically significant according to the χ^2 test.

The difference can be dramatic: In JAXEN, for instance, 98% of invariant-violating mutants are detected versus 44% of the non-violating mutants. This also means that in JAXEN, the rate of equivalent mutants across all generated invariant violators is not higher than 2%.

Mutants that violate invariants are more likely to be detected by actual tests.

7.4 Are mutations with the highest impact most likely to be detected?

For our third experiment, we wanted to explore what was so special about ASPECTJ that the invariant-violating mutations were less likely to be detected than the non-violating mutations. In Table 6, we see that ASPECTJ has far more violating mutants than all other projects combined. We wanted to *rank* these invariants, focusing on those with the *highest impact:* If a mutant violates many invariants, it should have a strong impact on the behavior of the program and is therefore less likely to be equivalent than mutants that violate fewer invariants.

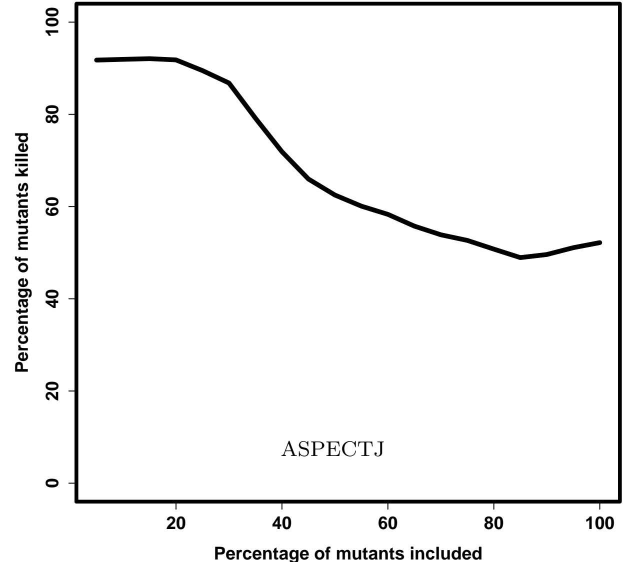
7.4.1 Hypothesis

In this experiment, we not only classify mutations by whether they violate invariants or not, but we actually rank them by impact—the number of *invariants violated.* This is our hypothesis:

H3 *The more invariants a mutant violates, the more likely it is to be detected by actual tests.*

7.4.2 Experimental Setup

Our experimental setup for **H3** is the same as for **H2** discussed in Section 7.3.3, except that we now compare the detection rate of the top n% of the invariant-violating mutants, where n ranges from 5–100.



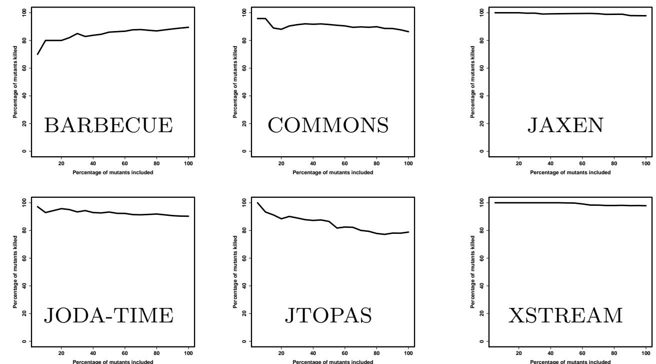


Figure 7: Detection rates (y) for the top x% mutations with the highest impact.

7.4.3 Results

Our results can be easily summarized. For ASPECTJ, there is a clear trend: The higher the ranking of a mutation (= the more invariants it violates), the higher its likelihood to be detected by the full test suite. This is shown in Figure 7: The x axis shows the subset considered, ranging from 5% (the top 5% mutations which violated the most invariants) to 100% (all mutations that violated at least one invariant. The y axis shows the respective detection rate.

This trend holds for all projects (see Figure 7), except for BARBECUE, the project with the lowest number of violating mutants: Just as ASPECTJ, the project with the highest number of violating mutants, benefits from ranking, it is reasonable to assume that the low number of violating invariants in BARBECUE prevents a meaningful ranking. Still, even focusing on the top 5% still yields a higher detection rate than average.

In all seven projects, higher-ranked mutations are more likely to be detected than all mutations (violating or non-violating). For all projects except BARBECUE, higher-ranked mutations are always more likely to be detected than the average across all violating mutations.

Table 6: Results for H2 and H3. Invariant-violating mutants (VM) have higher detection rates than non-violating mutants (NVM); best results are obtained by ranking VMs by the number of invariants violated.

Project Name	Number of NVMs	Number of VMs	NVMs detected (%)	VMs detected	p-value χ^2 test	Top 5% VMs detected	Top 10% VMs detected	Top 25% VMs detected
ASPECTJ	1159	13133	61.69	52.18	< 0.0001	91.77	91.93	89.49
BARBECUE	613	200	60.03	89.50	< 0.0001	70.00	80.00	82.00
COMMONS	10215	967	82.48	86.35	0.0021	95.83	95.83	90.46
JAXEN	2860	1250	44.48	97.84	< 0.0001	100.00	100.00	99.68
JODA-TIME	7523	2122	75.50	90.29	< 0.0001	97.17	92.92	95.09
JTOPAS	566	609	64.13	78.82	< 0.0001	100.00	93.33	90.13
XSTREAM	1835	1765	86.32	97.85	< 0.0001	100.00	100.00	100.00

VM = Invariant-Violating Mutant, NVM = Non-Violating Mutant.

In the rightmost columns of Table 6, we have shown the detection rate of the 5%, 10%, and 25% violating mutations with the highest impact. All detection rates are higher than those of 100% non-violating mutations (column five), thus confirming **H3**.

The more invariants a mutant violates, the more likely it is to be detected by actual tests.

Again, this implies that the undetected high-impact mutants will also have a low rate of equivalent mutants.

7.5 Discussion

The results of our case study suggest that developers should not only focus on those mutations that violate invariants, but that they actually should focus on those mutations that violate the most new invariants—which is exactly the ranking produced by JAVALANCHE.

Whenever mutation testing results in a large number of undetected mutants, it thus seems a good idea to *prioritize* the mutants by their impact on invariants:

1. By focusing on those mutants with the highest impact on invariants, one creates a *bias* towards non-equivalence. This is good, as this minimizes the number of equivalent mutants to deal with.

2. As these invariants are originally learned from test suite executions, this implies a bias towards mutations whose induced behavior *differs most* from the "normal" behavior as characterized by the test suite. By repeatedly re-learning invariants, this favors *diversity* in the test suite—a diversity that is based on program semantics (i.e., invariants) rather than program structure (i.e., standard coverage criteria). We consider such semantic diversity to be a desirable feature of test suites.

3. Focusing on high-impact mutants also implies focusing on those areas where a defect can create *the most damage* across the program execution. Again, we consider such a focus a very valuable property of a test suite.

Our results also indicate that it is generally useful to focus on those mutations that violate the most invariants. Even in a program with few violating mutants like BARBECUE, which did not benefit much from ranking (see the discussion in Section 7.4.3), the top-ranked mutations consistently yielded better detection results than the average mutation.

Finally, our results also place an upper bound on the number of equivalent mutants. Omitting BARBECUE due to its low number of violating mutants, the detection rate for the top 5% of violating mutations (Figure 7) is 92–100%, with an average of 97%. Thus, only 3% of these high-impact mutations were undetected, placing an upper bound on the number of equivalent (undetectable) mutants. (Note that in Section 7.2, only 17% of this small high-impact set were actually found to be equivalent, suggesting an even lower overall rate.) This very low rate is what makes our approach to mutation testing efficient.

On average, focusing on the top 5% of invariant-violating mutants yields ***less than 3% of equivalent mutants.***

7.6 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results.

Threats to *external validity* concern our ability to generalize the results of our study. The results of **H1** should be considered promising, but not generalizable, as the sample is small; the manual effort generally stands in the way of larger studies. Regarding **H2** and **H3**, we evaluated our approach on seven programs with different application domains and sizes; some of them were larger by several orders of magnitude than programs previously used for evaluation of mutation testing [20, 8, 11, 2]. Generally, our results were consistent across a wide range of programs. Still, there is a wide range of factors of both programs and test suites that may impact the results, and we therefore cannot claim that the results would be generalizable to other projects. Prospective users are advised to conduct a retrospective study like ours.

Threats to *internal validity* concern our ability to draw conclusions about the connections between our independent and dependent variables. Regarding **H1**, our own assessment may be subject to errors, incompetence, or bias; to counter these threats, all our assessments are publicly available on the project Web site. For **H2** and **H3**, our implementation could contain errors that affect the outcome. To control for these threats, we ensured that earlier stages (Figure 1) had no access to data used in later stages. Our statistical evaluation was conducted using textbooks techniques implemented in widely used frameworks. We advise and support independent confirmation of our results and make the necessary data available; see Section 9 for details.

Threats to *construct validity* concern the appropriateness of our measures for capturing our dependent variables. Regarding **H1**, being able to write a test is the ultimate measure whether a mutant is non-equivalent. In **H2** and **H3**, our assumption that the test suite measures real defects is an instance of the "competent programmer hypothesis" also underlying mutation testing [6]. This hypothesis may be wrong; however, the maturity and widespread usage of the subject programs suggest anything but incompetence. Further studies will help completing our knowledge on what makes a test suite adequate.

8. RELATED WORK

8.1 Mutation testing

Originally proposed in 1971 by Richard Lipton in a term paper [21], it took until 1978 until the first major work on mutation testing was published [6]. Mutation testing frameworks that are in use today include MOTHRA [5] for Fortran programs and μJava [14] for Java.

A number of studies have shown the effectiveness of mutation testing for assessing the adequacy of a test suite. Andrews et al. [2] showed that carefully selected operators yield trustworthy results—that is, generated mutants are similar to real faults. Walsh [27] found mutation testing to be more powerful than statement or branch coverage. Frankl [8] found mutation testing to be superior to all-use data flow coverage criteria. All of these studies were conducted on small programs, the largest one being the *Space* program (5,905 LOC) from the Siemens suite [2]. Still, the common conclusion that mutation testing is an effective technique for improving test suites forms the basis for our research.

Most of the more recent evaluations used programs with *mature test suites,* assuming the test suite is so exhaustive that "all mutants that were not killed by any test case were then deemed to be equivalent." [2]. The assumption of having a mature test suite is also the base for our automated empirical evaluation; the assumption of the test suite detecting all equivalent mutants is not.[4]

Assessing the state to check the impact of mutations is also related to the concept of *weak mutation testing,* as proposed by Howden [12]. Weak mutation testing assesses the effect of a mutation by assessing the state after its execution: If the state is different, then the mutation is detectable. Weak mutation testing thus checks whether the test suite could *possibly* detect a mutation; it does not matter whether the tests actually pass or not. Strong mutation testing, which is what we assume, assesses the test suite by determining whether it *actually* detects a mutation.

[4]The assumption that every mutant not detected by a "perfect" test suite must be equivalent can be motivated as follows. If the ratio of equivalent mutants being created is fixed, and the test suite is constantly being improved to detect more and more mutations, than the ratio of equivalent mutants among the non-detected mutations must raise constantly—up to 100%. Unfortunately, this has an interesting side effect. If I have an "almost perfect" test suite, wouldn't this imply that *almost all* of the non-detected mutants would be equivalent? And how would we then ever be able to come even close to perfection?

8.2 Equivalent Mutants

The issue of equivalent mutants has frustrated generations of mutation testers. In Section 1, we have quoted Frankl et al. [8] on the enormous amount of work needed to eliminate equivalent mutants. A number of researchers have tackled the problem of detecting equivalent mutants. Baldwin and Sayward [3] were the first ones to suggest *heuristics* for detecting equivalent mutants. Their approach, based on detecting idioms from semantics-preserving compiler optimizations, was shown by Offutt and Craft [18] to detect approximately 10% of equivalent mutants.

In 1996, Offutt and Pan [20] realized that detecting equivalent mutants is an instance of the *infeasible path* problem which also occurs in other testing techniques. They presented an approach based on solving *path conditions* that originate from a mutant. If the constraint solver can show that all subsequent states are equivalent, the mutant is deemed equivalent. The technique was reported to detect 48% of equivalent mutants. A similar approach, based on *program slicing,* was presented by Hierons and Harman [11]; this approach additionally provides guidance in detecting the locations potentially affected by a mutant. Modern change impact analysis [24] can do this in presence of subtyping and dynamic dispatch. The recent concept of *differential symbolic execution* [23] brings the promise of easily detecting potential impact of changes.

All of these techniques are orthogonal to ours; indeed, if we can prove statically that a mutation will have no impact, we can effectively omit the run-time tests. The question is on how well these static approaches scale up when it comes to detecting mutant equivalence in real programs. Offutt and Pan [20]'s technique, for instance, was evaluated on eleven Fortran 77 programs which "range in size from about 11 to 30 executable statements". In contrast, the programs we have been looking at are larger by several orders of magnitude.

8.3 Invariants and Contracts

The idea of *checking* program state at run-time is as old as programming itself. *Design by contract* [16] mandates specifying invariants for every public method in a program; the resulting runtime checkers effectively catch errors at the moment they originate.

If the programmer does not provide invariants, one can *infer* them from program runs. This is the idea of dynamic invariants, as realized in the DAIKON tool by Ernst et al. [7]. Most related to our work is the ECLAT tool by Pacheco and Ernst [22] which selects, from a set of test inputs, a subset that is most likely to reveal defects, by assessing the impact of the inputs on dynamic invariants. McCamant and Ernst [15] use dynamic invariants to check whether invariants had changed after a code change—indicating a potential problem in the future. (Our approach, of course, explicitly looks for such invariant changes.)

Sosič and Abramson [25] suggest another technique to detect the impact of changes. The idea of *relative debugging* is to compare the execution of two programs (in our setting, the original vs. the mutant) and automatically report any differences in variable values. While the differences do not translate into invariants, they could nonetheless serve as impact indicators.

Our concept of efficient dynamic invariant checking was inspired by the DIDUCE tool by Hangal and Lam [10], flagging invariant violations as they occur during the run. Nei-

ther approach discussed in this section was applied to mutation testing so far.

9. CONCLUSION AND CONSEQUENCES

If a mutant violates even very simple invariants, it is more likely to be detectable by an actual test. When improving test suites, test managers therefore should focus on those surviving mutations that have the greatest impact on invariants. With its low rate of equivalent mutants, our approach provides a precise, reasonably efficient, and fully automatic measure of the adequacy of a test suite—making mutation testing finally scalable to large-scale programs.

Besides generally evolving JAVALANCHE, our future work will concentrate on the following topics:

Alternative impact measures. While we consider violations of invariants to be particularly useful predictors of failures, there are many ways to determine the impact of a change. One can measure impact in anything that characterizes a run; including *coverage criteria* such as statement or branch coverage, *sequences* of executed methods [4], or *numerical ranges* of data and increments [10]. First experiments with statement coverage [9] indicate a better suitability for ranking mutations for programs with small impact (Section 7.4); taken on its own, statement coverage misses impact that takes place on data alone. We want to examine how these characteristics are suited and can be combined for assessing the impact of a change and the equivalence of the resulting mutation.

Impact as similarity measure. A common criticism against mutation testing is that mutations may be different from actual defects. From defect history, we can assess how earlier defects impacted program execution—and then generate mutations that have similar impact. This will allow *calibration* of mutants to match a given defect history.

Adaptive mutation testing. If we establish large impact or similarity to defect history as desirable properties, one can also *evolve* appropriate mutants—by assessing the properties of the "fittest" mutants and propagating them to another generation of mutants [1]. This will allow for automated natural selection of mutants, optimizing them towards a specific goal—a maximum impact or a maximum similarity with history.

Eventually, large-scale automation in mutation testing also enables large-scale *assessment* of mutation testing, opening the doors for lots of future research. To support this research, our subject programs, instrumented with dynamic invariant checkers, together with a description of the mutants and tests from Section 7.2, are publicly available, allowing for easy replication (and extension) of our experiments. For more information, visit

`http://www.st.cs.uni-saarland.de/mutation/`

Acknowledgments. The idea of measuring the impact of changes was born out of fruitful discussions with Andreas Leitner. Irina Brudaru, Yana Mileva, Frank Padberg, Rahul Premraj, Andrzej Wasylkowski, and Tom Zimmermann provided helpful feedback on earlier revisions of this paper; special thanks go to Michael Ernst for his detailed comments as well as DAIKON support. Valentin Dallmeier is a member of and supported by the *Saarland Graduate School of Computer Science*.

10. REFERENCES

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation—GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1338–1349, Seattle, Washington, 2004. Springer.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, New York, NY, USA, 2005. ACM.

[3] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical Report 276, Yale University, Department of Computer Science, 1979.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP '05: Proceedings of 19th European Conference on Object-Oriented Programming*, number 3586 in Lecture Notes in Computer Science, pages 528–550. Springer, 2005.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, 1988. IEEE Computer Society Press.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[8] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38:235–253, 1997.

[9] B. J. M. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. Technical report, Saarland University, 2009. Short paper submitted to Mutation 2009: International Workshop on Mutation Analysis.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In M. Young and J. Magee, editors, *ICSE '02: Proceedings of the 24st International Conference on Software Engineering*, pages 291–302, Orlando, Florida, 2002.

[11] R. Hierons and M. Harman. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[12] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[13] http://asm.objectweb.org/. ASM 2.2.1 GPL, 2008.

[14] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: a mutation system for Java. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 827–830, New York, NY, USA, 2006. ACM.

[15] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE '03: Proceedings of the 9th European Software Engineering Conference and the 11th International Symposium on Foundations of Software Engineering*, pages 287–296, New York, NY, USA, 2003. ACM.

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[17] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 351–360, New York, NY, USA, 2008. ACM.

[18] A. J. Offutt and W. M. Craft. Using compiler optimiziation techniques to detect equivalent mutants. *Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[20] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *COMPASS '96: Proceedings 11th Conference on Computer Assurance*, pages 224–236, Gathersburg, MD, 1996.

[21] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the orthogonal*, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[22] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP '05: Procceedings of the 9th European Conference on Object-Oriented Programming*, pages 504–527, Glasgow, Scotland, 2005.

[23] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE 08: Proceedings of the 16th International Symposium on the Foundations of Software Engineering*, Atlanta, Georgia, 2008.

[24] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 3rd Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, New York, NY, USA, 2001. ACM.

[25] R. Sosič and D. A. Abramson. Guard: A relative debugger. *Software Practice and Experience*, 27(2):185–206, Feb. 1997.

[26] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ISSTA '93: Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 139–148, New York, NY, USA, 1993. ACM.

[27] P. J. Walsh. *A measure of test case completeness (software, engineering)*. PhD thesis, State University of New York at Binghamton, Binghamton, NY, USA, 1985.