The Impact of Equivalent Mutants

Bernhard J. M. Grün · David Schuler · Andreas Zeller
Saarland University, Saarbrücken, Germany
{gruen, schuler, zeller}@st.cs.uni-saarland.de

Abstract

If a mutation is not killed by a 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. We found such equivalent mutants *to be surprisingly common: In an experiment on the JAXEN XPATH query engine, 8/20 = 40% of all mutations turned out to be equivalent. Worse, checking the equivalency took us 15 minutes for a single mutation. Equivalent mutants thus make it impossible to automatically assess test suites by means of mutation testing.*

To identify equivalent mutants, we are currently investigating the impact *of a mutation on the execution: the more a mutation alters the execution, the higher the chance of it being non-equivalent. First experiments assessing the impact on code coverage are promising.*

1. Introduction

One of the research areas at the chair for software engineering at Saarland University is *mining software repositories*—that is, learning patterns and rules from software archives such as version and bug databases to summarize past development and to predict future development. An important issue is *defect prediction*—that is, predicting the defect density of individual components. As we expect the defect density to decrease with increasing test quality, we studied mutation testing as an assessment for test quality.

When starting experimenting with mutation testing, we expected a large usage of computing resources, which indeed happened to be the case. What we were not prepared for, though, was the problem of *equivalent mutants*—mutations that leave the program's overall semantics unchanged, and therefore cannot be caught by any test suite.

To illustrate the problem, consider the code in Figure 1. This code, excerpted from the JAXEN XPATH engine, checks whether a string `myPrefix` is already contained in a map `nsMap`. If not, a new node is created and added to `nsMap` together with a new `NamespaceNode` object.

```
// org.jaxen.dom.DocumentNavigator, Line 360
String myPrefix = n.getPrefix();
if (!nsMap.containsKey(myPrefix)) ⇒ if (true)
{
  NamespaceNode ns =
    new NamespaceNode((Node)contextNode,
      myPrefix, myNamespace);
  nsMap.put(myPrefix, ns);
}
```

Figure 1. Mutating the condition to true has no effect on JAXEN semantics.

A possible mutation to this code is to change the condition to a constant such as `true`. This updates the map entries even if the prefix string already is in the map, and thus should result in very different map contents. However, none of the JAXEN tests detects ("kills") this mutation. Does this mean the test is inadequate?

Further investigation of the JAXEN code shows that the above mutation does not affect the remaining code—it is an equivalent mutant. This is because the values of the `contextNode` and `myNamespace` variables have the same values for every `myPrefix` instance—and thus, the `NamespaceNode` objects are the same, too. The map update as forced by the mutation does not alter the map contents—it just replaces an existing node with a new, equivalent node. Establishing that this property holds is time-consuming, as one has to trace back the origins and dependencies of all involved variables.

Equivalent mutants thus act as *false positives:* They appear to indicate a weakness in the test suite, but in fact do not, as no test can detect them. 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.

False positives also occur using other test criteria. Using statement coverage to improve a test suite, one may also find statements that can not be reached by any test. This, however, implies a defect in the code. Using branch coverage, basic condition coverage, or def/use coverage, a false positive reveals a fault as well. Using mutation testing, a false positive just wastes valuable time.

We could easily live with equivalent mutants if there were few, or if assessing them was easy. Unfortunately, neither holds: We found many equivalent mutants, and we found that assessing them was very time-consuming. In fact, equivalent mutants effectively prohibited any automatic assessment of test quality by means of mutation testing. We therefore postponed our initial plan to use mutation testing for defect prediction, focusing on the problem of equivalent mutants first.

In this paper, we present the current state of our investigations. We make the following contributions:

Equivalent mutants may be widespread. We give a first assessment on how significant the problem of equivalent mutants actually is. In a sample of 20 mutations, 8 were equivalent—that is, no less than 40%.

Assessing mutation equivalence is time-consuming. On average, it took us 15 minutes to assess the equivalence of a single mutation. This effort makes mutation testing prohibitive even for small programs.

Equivalent mutations have less impact on execution. We have discovered that mutations that alter the dynamic control flow are less likely to be equivalent; the higher the impact, the lower the chance of equivalence.

Our investigations are still at an early stage. We think, though, that these early results are both disturbing (when it comes to the extent of the problem) as well as promising (when it comes to possible solutions).

2. The Javalanche Framework

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 [9], none met our requirements in terms of *automation* and *scalability*. We therefore decided to implement our own mutation engine with special focus on these requirements.

The key features of our implementation, called JAVALANCHE, are:

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 [11]. JAVALANCHE therefore uses the same small set of operators as proposed by Offutt [11] 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 large-scale program, this easily result in thousands of different mutated versions, which are too many to be handled effectively. To reduce the number of generated versions, we use *mutant schema generation* as proposed by Untch et al. [15].

Table 1. JAVALANCHE **mutation operators**

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 in the source code.
Replace arithmetic operator. Replace an arithmetic operator by another arithmetic operator, e.g. $+$ by $-$.
Omit method calls. Omit a method call. If the method has a return value, a default value is used instead, e.g. $x = \mathit{Math.random}()$ is replaced by $x = 0.0$

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 additionally works directly on Java byte code, it also avoids costly recompilation. As discussed in the next section, JAVALANCHE easily scales up to medium-sized programs of 10,000 lines and more.

3. The Problem: Equivalent Mutants

The first program we experimented with is the JAXEN XPATH query engine, as it has a reasonable size, comes with an extensive test suite, and is being used in dozens of projects. In version 1.1.1, JAXEN has 12,449 lines of Java code, and comes with 8,371 lines of testing code, organized in 680 individual tests. These tests cover 66.79% of all statements.

We ran JAVALANCHE on JAXEN, creating 9,819 individual mutations. Of these, 5,127 were not detected by any test. When we started examining them, we noticed that a number of them were equivalent, and thus did not help in improving the test suite. Furthermore, we were surprised at how long it took us to identify equivalence.

To quantify the extent of the problem, we set up an experiment: We took a sample of JAXEN mutations, as produced by JAVALANCHE, and manually assessed whether the resulting mutants would be equivalent or not; furthermore, we measured how much time the assessment would take.

3.1. Experiment Setting

For the experiment, we divided the mutation testing results in three sets:

Not covered mutations. These mutations were not exercised by the test suite, and thus cannot be detected. Of the 9,819 mutations, 3,194 (32%) ended up in this set.

Whether such a mutation is equivalent or non-equivalent does not matter much: The obvious first step for improving the test suite is to *increase coverage* such that the mutation is exercised.

Killed mutations. The mutations in this set were exercised and detected by the test suite. This set contained 4,692 (48%) mutations.

All these mutations are non-equivalent, since the different behavior was detected by the test suite.

Covered and not killed mutations. The mutations were exercised, but not detected by the test suite. This set contained 1,933 (20%) mutations.

Such mutations can either be equivalent or non-equivalent, because it is unknown if they change the observable behavior of the program. Therefore, we took our sample from this set.

Of the set of *covered and not killed* mutations, we took a sample of 20 mutations that were chosen randomly in such a way that each mutated a different Java class.

This set of 20 mutations was then assessed by two authors of this paper independently. We had worked with the program previously, but had no detailed knowledge about its implementation. For every mutation, each of us two classified it to be *non-equivalent,* proven by writing a test case that kills the mutation, *equivalent,* or *undecided.* We also recorded how much time it took to asses the mutation.

After finishing the study individually, we merged the results. If at least one author wrote a test case that kills the mutation, it was considered to be *non-equivalent.* If there was no test case, and both authors classified it as equivalent, the mutation was considered to be *equivalent.* Otherwise, the mutation remained classified as *undecided.*

3.2. Results

Out of 20 mutations taken from 20 different classes,

- 10 (50%) were considered to be non-equivalent,
- 2 (10%) were undecided, and
- 8 (40%) were considered equivalent.

```
JaxenHandler handler = new JaxenHandler();
handler.setXPathFactory(new DefaultXPathFactory());
⇒ call to setXPathFactory(...) gets omitted
```

Figure 2. An equivalent mutation in unneeded code.

Assessing a single mutation took between 1 and 50 minutes. Investigating all mutations took both of us a total of 10 hours, which results in a mean average of about 15 minutes per mutation for one person.

For all mutations that were considered to be non-equivalent, we wrote tests. A few mutations could be easily checked for, because a function or constructor returns a different value. Most of the tests were derived from existing tests, e.g. by using modified XPATH queries or writing stronger asserts. A rather complex test involved using a different locale (Turkish) to trigger some special upper-case behavior that was altered by a mutation.

Out of the eight mutations that where considered equivalent, we found four reasons why they were equivalent:

Mutations in unneeded code. Some parts of the code just duplicate some default behavior, or set a value that is later reset without using it in-between. When mutating these parts, the program is not affected.

As an example for such a mutation, consider Line 104 of `org.jaxen.pattern.PatternParser`, which suppresses the call `setXpathFactory()` (Figure 2). This call is equivalent, as the standard constructor of `JaxenHandler` sets the appropriate field to `DefaultXPathFactory()` anyway. Thus, this mutation has no effect on the program execution.

Mutations that suppress speed improvements. These mutations suppress some performance improvement, but the results of the computation remain the same.

An example for this type is the mutation that adds the same key-value pair to a `Map` several times, which was shown in Figure 1 and discussed earlier.

Equivalent mutations that alter state. These mutations alter the private state of a class or the return value of private methods. However, given the way these altered values are used in the program, the behavior remains unchanged.

As an example, take the mutation in Line 147 in class `org.jaxen.expr.NodeComparator` (Figure 3): It initializes a value to start the depth computation with 1 instead of 0, which causes different depth values to be returned by the method. Since this method is private, and the depth for all nodes that are compared are increased by 1, the comparison of nodes remains correct.

```
private int getDepth(Object o)
    throws UnsupportedAxisException {
  int depth = 0; ⇒ depth = 1;
  Object parent = o;
  while ((parent = navigator.getParentNode(parent))
      != null) {
    depth++;
  }
  return depth;
}
```

Figure 3. An equivalent mutation that alters state.

```
NamespaceNode (Node parent, Node attribute)
  String attributeName = attribute.getNodeName();
  if (attributeName.equals("xmlns")) {
  ⇒ if(false) {
    this.name = "";
  }
  else if (attributeName.startsWith("xmlns:")) {
    this.name = attributeName.substring(6);
    // the part after "xmlns:"
  }
  else { // workaround for Crimson bug;
      // Crimson incorrectly reports the
      // prefix as the node name
    this.name = attributeName;
  }
  this.parent = parent;
  this.value = attribute.getNodeValue();
}
```

Figure 4. An equivalent mutation that could not be triggered.

Mutations that cannot be triggered. These are mutations that would cause the program to fail under specific conditions. However, meeting these conditions causes other failures upstream.

For instance, consider the mutation in Line 147 of `org.jaxen.dom.NamespaceNode` (Figure 4).[1] It replaces the contents of the first if condition with `false`, such that the body is never executed. However, supplying an XML Document that would evaluate the non-mutated condition to true causes an exception by the XML parser.

We do not know whether an equivalent mutation rate of 40% is common for other programs; nor do we know whether examining another sample would have yielded other results. For us, the results effectively meant that mutation testing can not be used for an automatic assessment of test quality; a false positive rate this high is far too much to ignore.

We concede that more work should be done to put this proportion of 40% into perspective. The phrase "more work should be done" is not to be taken lightly, though. If we were to assess all 1,933 undetected exercised mutations, spending a total of 2×15 minutes on each sums up to ~1,000 working hours, or one person-year—indeed, a lot of "more work", and just for one single program.

[1] Yes, this is again Line 147. Some coincidences just happen.

Table 2. Impact on coverage versus equivalence.

	non-equivalent	equivalent	undecided
impact	6	2	2
no impact	4	6	0

4. A Solution: Assessing Mutation Impact

Equivalent mutants are defined as having no impact on the program execution. The impact of a mutation can be assessed by checking the program state at the end of a computation, as tests do. However, we can also assess the impact of a mutation while the computation is not complete. In particular, we can measure *changes in program behavior* between the mutant and the original version. One aspect that is particularly easy to measure is *control flow:* If a mutation alters the control flow of the execution, different statements would be executed in a different order—an impact that is easy to detect using standard coverage measurement techniques. But would such an impact imply non-equivalence?

In order to test the relationship between coverage and non-equivalence, we developed a program that computes the code coverage of a program, and integrated it into the JAVALANCHE framework. The program records the *statement coverage* for each test case and every mutation—that is, the number of times a statement is executed.

By comparing the coverage of the original execution with the coverage of the mutated execution, we can determine the *coverage difference.* As an impact measure, we have chosen the *number of classes* that have different code coverage. This measure is motivated by the hypothesis that a mutation that has *non-local impact* on the coverage is more likely to change the observable behavior of the program. Furthermore, we would assume mutations that are undetected despite having impact across several classes to be particularly valuable for improving the test suite, as they indicate inadequate testing of multiple classes at once.

4.1. Mutations and Impact

In a first experiment, we checked the hypothesis that *mutations with high impact would be more likely to be non-equivalent.* We therefore took the 20 JAXEN random mutations, as already assessed in Section 3, and checked whether the mutation would also impact the code coverage.

Our results are summarized in Table 2: We can see that if a mutation had impact on code coverage, it was more likely to be non-equivalent; if it did not have impact on code coverage, it was more likely to be equivalent.

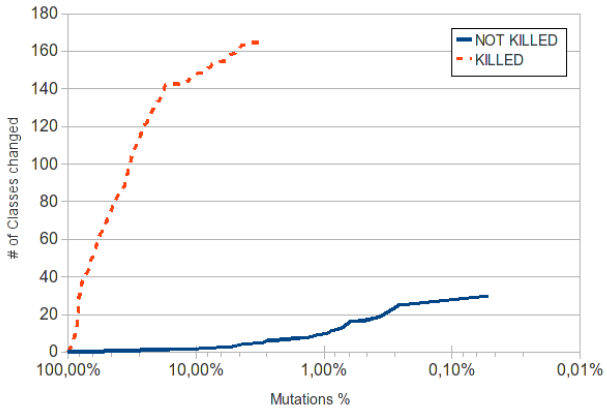

Figure 5. Impact of killed and non-killed mutations.

4.2. Kill Rate and Impact

In a second experiment, we examined the relation between kill rate and impact. The idea is that if the test suite detects a mutant, it is by definition non-equivalent. The hypothesis followed that *mutations with high impact would be more likely to be detected,* establishing a relationship between impact on code coverage and impact on testable behavior. We therefore measured the impact of all killed and non-killed mutations on code coverage.

Our results are summarized in Figure 5, showing the coverage impact on different classes for killed and non-killed mutations. The x-axis gives the percentage of mutations (note that the x-axis has a reverse logarithmic scale), and the y-axis the number of classes with coverage differences. A data point (x, y) in this diagram means that x percent of mutations have at least an impact on y classes.

The diagram shows that more killed mutations have an impact on the coverage than not killed and that they also have an impact on more classes. 98% of the killed mutations had an impact on the coverage, while only 27% of the not killed mutations had an impact.

These results suggest that there is a strong correlation between a mutation being killed by the test suite and its impact on the coverage.

4.3. Ranking along Impact

In our third experiment, we examined whether there would be a relationship between the *amount of the impact* and the likelihood of non-equivalence. The hypothesis was that *those mutations with the highest impact would be those that are least likely to be non-equivalent,* and vice versa. The hypothesis implies that when faced with thousands of surviving mutations, one should focus on those mutations with the highest impact—as one would run little risk to spend time on equivalent mutations, and as one may also obtain particularly valuable mutations, as discussed earlier.

For this experiment, we ranked the mutations that were not detected by the test suite according to their impact. We then took

- the 20 mutations that had *the most impact* (i.e., that induced coverage differences in a maximum of classes)
- the 20 mutations that had the *least impact,* effectively showing no difference in coverage.

As in Section 4, we made sure to choose only one mutation per class. Otherwise, both sets would have been populated with several mutations in the same class which happen to have the same big or little effect. We then manually assessed these mutations as discussed in Section 4, again taking manual effort.

These are our results:

- For the 20 not killed mutants with the *highest impact,* we found 18 non-equivalent mutants and 2 equivalent. The rate of 90% non-equivalent mutants is higher than the rate of 50% that we found in our study of randomly chosen mutations.

  The mutation with the highest impact was applied to the JAXEN verifier; altered the code coverage in 25 different classes. It causes JAXEN to ignore all characters with a ASCII code of 192 or higher (say, a UTF-8 character such as a € sign), which caused several changes downstream, as branches were no longer taken. None of the JAXEN tests checks for this condition; however, it is easy to improve the test suite by writing a test which detects the mutation.

- For the 20 not killed mutants with the *lowest impact,* we found 9 non-equivalent mutants and 11 equivalent mutants. The rate of 55% of equivalent mutants is higher than the 10% for the mutations with the highest impact and also improves upon the 40% from our random sample.

The implications of these results are clear: *One can effectively reduce the number of non-equivalent mutants by focusing on those mutants with the highest impact.* At the same time, one obtains mutations that effectively change many aspects of program execution, yet are undiscovered by the test suite.

The drawback of our approach is that it creates *bias* in the set of mutations. Rather than assessing a set of mutations that is applied evenly across the program, one would now deal with a very specific subset, which may or may not be helpful to assess test quality suite. At this point, our assumption is that such mutations are particularly valuable (such as the *verifier* mutation); this assumption is also backed by the fact that mutations with impact also tend to be detected by the test suite (Section 4.2). The benefits of such a choice for general test assessment are to be demonstrated in future work, though.

4.4. Killed without Impact

Finally, we examined a sample of 20 mutations which had *no impact* on coverage, but were killed by the test suite. These mutations indicate that it is possible to impact program execution, but leave coverage unchanged, which implies possible improvements of our technique. We identified two classes of mutations that fell into this category:

1. *Mutations that alter the string representation of a class.* These faulty string representations have no impact on the program execution, but are checked for by the test suite.

2. *Mutations that alter return values of methods.* These mutations alter a return value of a method that is just passed through from this point in the program and is not used in conditional expressions anymore. However, the test suite checks for this return value.

These results suggest that besides impact on control flow, as indicated by coverage, one should additionally assess the impact of mutations on data values and data flow.

4.5. 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. Our results are taken from just one program, JAXEN, and while we think that the results form a good starting point for discussion and further inquiry, we cannot claim that the results would be generalizable to other programs. 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 the assessment of equivalence or non-equivalence, our own assessment may be subject to errors, incompetence, or bias. We advise and support independent confirmation of our results and make the necessary data available; see Section 6 for details.

Threats to *construct validity* concern the appropriateness of our measures for capturing our dependent variables. We already have commented on the risk of bias by focusing on the mutations with the highest impact. Regarding the assessment of equivalence nor non-equivalence, being able to write a test is the ultimate measure whether a mutant is non-equivalent.

5. Related Work

We were not the first to be frustrated by the problem of equivalent mutants. Frankl et al. [6] made a revealing statement on the enormous amount of work needed to eliminate out equivalent mutants:

> *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.*

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 [10] to detect approximately 10% of equivalent mutants.

In 1996, Offutt and Pan [12] 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 [8]; this approach additionally provides guidance in detecting the locations potentially affected by a mutant. Modern change impact analysis [14] can do this in presence of subtyping and dynamic dispatch. The recent concept of *differential symbolic execution* [13] 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 on control flow, we can effectively omit the run-time tests. The question is how well these static approaches scale up when it comes to detecting mutant equivalence in real programs. Offutt and Pan [12]'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 JAXEN program we have been looking at is larger by several orders of magnitude.

6. Conclusion and Future Work

The effort it takes to identify equivalent mutants is an important problem. A rate of 40% equivalent mutants, as observed in our JAXEN sample, may be tolerable if programmers are willing to spend much time on improving test suites anyway. However, it threatens to make the results of

mutation testing useless for any kind of automatic or uninterpreted assessment.

We found that if a mutant alters control flow, it is more likely to be detectable by an actual test. When improving test suites, test managers therefore may focus on those surviving mutations that have the greatest impact on code coverage. However, all these results are premature; further studies involving more subjects and more assessment will be needed to fully validate the approach.

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

Focusing on mutations with impact. One of our assumptions is that mutations with a high impact not only create fewer equivalent mutants, but also are more valuable for test suite improvement than other, low-impact mutants. The main obstacle for this kind of work is the high amount of work it takes to classify mutant equivalence; we are currently investigating evaluation schemes that allow for a higher degree of automation.

Alternative impact measures. While we consider changes of coverage to be useful as predictors of non-equivalence, there are many ways to determine the impact of a change. One can measure impact in anything that characterizes a run; including *dynamic invariants* on variable values [5], *numerical ranges* of data and increments [7], or *sequences* of executed methods [4]. We want to examine how these characteristics are suited for assessing the impact of a change and the usefulness of the resulting mutation.

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.

To support this research, we have made the description of all JAXEN mutants and tests publicly available, allowing for easy replication (and extension) of our experiments. For more information, visit

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

Acknowledgments. Valentin Dallmeier and Yana Mileva gave helpful comments on an earlier revision of this paper.

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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.