(Un-)Covering Equivalent Mutants

David Schuler
Saarland University
Saarbrücken, Germany
schuler@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
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 test suite fails to detect a mutation, it may also fail to detect real defects—and hence should be improved. However, there also are mutations which keep the program semantics unchanged and thus cannot be detected by any test suite. Such equivalent mutants must be weeded out *manually,* which is a tedious task. In this paper, we examine whether *changes in coverage* can be used to detect non-equivalent mutants: If a mutant changes the coverage of a run, it is more likely to be non-equivalent. In a sample of 140 manually classified mutations of seven Java programs with 5,000 to 100,000 lines of code, we found that: (a) the problem is serious and widespread—about 45% of all undetected mutants turned out to be equivalent; (b) manual classification takes time—about 15 minutes per mutation; (c) coverage is a simple, efficient, and effective means to identify equivalent mutants—with a classification precision of 75% and a recall of 56%; and (d) coverage as an equivalence detector is superior to the state of the art, in particular violations of dynamic invariants. Our detectors have been released as part of the open source JAVALANCHE framework; the data set is publicly available for replication and extension of experiments.**

***Keywords*-mutation testing; code coverage; dynamic analysis;**

I. Introduction

To assess the quality of a software, one uses *testing:* executing the program with the purpose of detecting a defect. Obviously, the better the test suite, the higher the chance of finding errors. But how do we know how "good" a test suite actually is? One of the best ways to assess the quality of a test suite is *mutation testing*—that is, repeatedly seeding artificial defects ("mutations") into the software. If the test suite fails to find these artificial defects, it is likely to miss real defects, too—and hence should be improved. A typical usage of mutation testing is to seed thousands of mutations into the program—and then examine those which the test suite did not catch.

Mutation testing has been shown to be an effective assessment for test suite quality [1] and superior to common assessments such as coverage metrics [2], [3]. This effectiveness comes at a cost. The first problem is that the repeated execution of test suites requires significant computing resources. With appropriate optimizations, though, it is possible to mutation test even 100,000-line programs within a few CPU hours [4]. The second problem is more significant: It is possible that a mutation leaves the program's semantics unchanged. Such an *equivalent mutation* cannot be caught by any test. It needs to be weeded out manually; and it just wastes time as the developer focuses on the next uncaught mutation without improving the test suite. Although there are techniques to detect some equivalent mutations [5], [6], the general problem is *undecidable* [7].

How widespread is the problem of equivalent mutants? In this paper, we have manually assessed a random sample of 140 uncaught mutations in seven Java programs. Our results have serious consequences:

- **It takes 15 minutes to assess one single mutation.** It is surprisingly difficult to assess the effect of a single change to the code—in particular, if the change is randomly generated.
- **45% of all uncaught mutations are equivalent.** This high number may come as a surprise, but keep in mind that several *non-equivalent* mutants are already caught by the test suite.
- **The problem gets worse as the test suite improves.** Since the number of equivalent mutants stays fixed, their percentage increases further as the test suite finds more and more non-equivalent mutants.

We also evaluate *solutions*, though. In an earlier workshop paper [8], we had examined the impact of mutations on *coverage*—that is, whether lines are executed or not. In a proof of concept, it turned out that equivalent mutants tended to keep coverage unchanged, whereas non-equivalent mutants actually changed the coverage. In this paper, we have refined this technique and applied it to the 140 previously classified mutations. The results are promising: 75% of the mutants are correctly classified based on their impact on coverage. This means that the effort for mutation testing is significantly reduced; at the same time, the technique is easily deployed as coverage measurement tools are commonplace.

Our paper is organized as follows. We dig into the problem by showing some real-world equivalent and non-equivalent mutants (Section II). After introducing our JAVALANCHE mutation framework (Section III) and the subject programs (Section IV), our classification study gives details on the ubiquity of equivalent mutants (Section V).

```
...
for (final Iterator iter = methods.iterator();
      iter.hasNext();) {
  final Method method = (Method)iter.next();
  method.setAccessible(true);
  if (Factory.class.isAssignableFrom(
       method.getDeclaringClass())
     || ⇒ &&
     (method.getModifiers() & (Modifier.FINAL |
      Modifier.STATIC)) > 0) {
    iter.remove();
    continue;
...
```

Figure 1. A non-equivalent mutation from the XSTREAM project.

We then describe how to assess the impact of mutants on coverage (Section VI), followed by an evaluation of the approach (Section VII). After discussing the threats to validity (Section VIII), we explore the related work (Section IX) and close with conclusion and future work (Section X).

II. Equivalent Mutants

One use of mutation testing is to improve a test suite by providing tests for undetected mutants. To this end, mutations are applied to a program, and one checks whether the test suite detects them or not. This step is carried out automatically and results in a set of undetected mutants. A programmer then tries to add or modify existing tests such that previously undetected mutants are detected.

Unfortunately, there are several reasons why a test suite might fail to detect a mutation, which determine its usefulness to the programmer:

1) The mutation may not change the programs semantics and thus *cannot be detected*. These equivalent mutations cannot help in improving the test suite and place an additional burden on the programmer, because the equivalence of a mutation has to be assessed manually.
2) The mutated statement may *not be executed*. In order to find non-executed statements, standard coverage criteria can be used.
3) The mutation may not be detected because of an *inadequate test suite*. These are the *most valuable* mutations, since they provide indicators to improve the test suite that other coverage metrics might not provide. If a mutation is covered but not detected, this either means that the tests do not check the results well enough, or that the input data is not chosen carefully enough to trigger the erroneous behavior.

Let us characterize these different kinds of undetected mutations, using the XSTREAM project as example.

A. A regular mutation

Figure 1 shows a mutation in the *createCallbackIndexMap* method of class *CGLIBEnhancedConverter*, which changes an `||` operator to an `&&` operator. This causes the expression to evaluate to `true` when it should evaluate

```
void addValue(String value, Type type) {
  if (newLineProposed && ((format.mode()
        & ⇒ |
      Format.COMPACT_EMPTY_ELEMENT) != 0)) {
    writeNewLine();
  }
  if (type == Type.STRING) {
    writer.write('"');
  }
  writeText(value);
  if (type == Type.STRING) {
    writer.write('"');
  }
}
```

Figure 2. An equivalent mutation from the XSTREAM project.

```
public boolean aliasIsAttribute(String name) {
  return
     nameToType.containsKey(name) ⇒ null;
}
```

Figure 3. A mutation of XSTREAM project that is not executed by tests.

to `false`, and then to remove the method from an underlying map. In the end, this results in spurious entries in the XML representation of an object. An existing test case of the XSTREAM test suite triggers this behaviour (*testSupportProxiesUsingFactoryWithMultipleCallbacks* in class *com.thoughtworks.acceptance.CglibCompatibilityTest*). However, this test fails to check the results thoroughly. By modifying this test, the mutation can be detected.

B. An equivalent mutation

Another mutation of the XSTREAM project is shown in Figure 2, applied to line 198 of class `JsonWriter`. Here, the mutation changes an `&` operator to an `|` operator, which might cause the expression to evaluate to true when it should not. This expression is disjunct with the variable `newLineProposed`, and gets only executed when the variable evaluates to true. Further investigation shows that `newLineProposed` is only set to true in one place of the program, and only if the same condition as in the mutated statement `format.mode() & Format.COMPACT_EMPTY_ELEMENT) != 0` is true. Thus, in the mutated statement, this condition is always true when it is evaluated (when `newLineProposed` is true). The mutation is equivalent.

C. A mutation that is not executed

The method *aliasIsAttribute* of *ClassAliasingMapper* shown in Figure 3 returns true if the given name is an alias for another type. What happens if we mutate this method such that it always returns `null`? The existing test suite does not detect this mutation, because the statement is not executed. Thus a test should be added that checks this functionality. However, to detect uncovered code, we do not need to apply full-fledged mutation testing. Simple

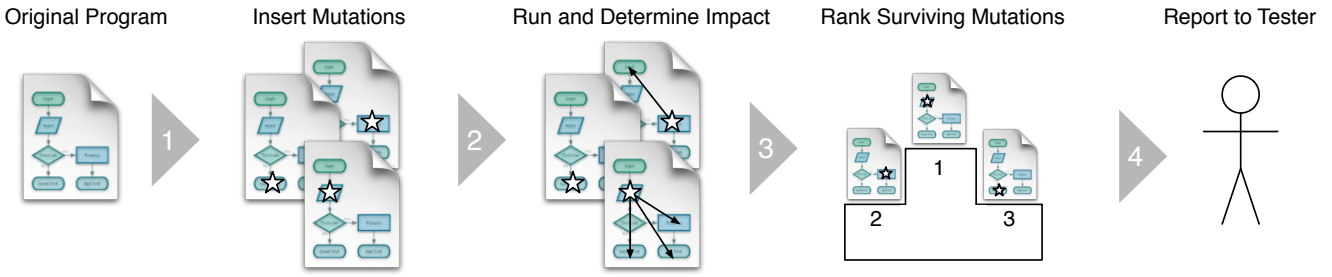

Figure 4. The JAVALANCHE process. After generating mutations (Step 1), JAVALANCHE runs the test suite on each and ranks mutations by their impact on data and coverage (Steps 2 and 3). Finally, the tester (Step 4) improves the test suite to detect the top-ranked mutations.

Table II
DESCRIPTION OF SUBJECT PROGRAMS.

Project Name	Description	Version	Program size (LOC)	Test code size (LOC)	Number of tests	Test suite runtime
ASPECTJ	AOP extension to Java	cvs: 2007-09-15	94,902	14,736	336	9s
BARBECUE	Bar code creator	svn: 2007-11-26	4,837	3,293	153	3s
COMMONS	Helper utilities	svn: 2009-08-24	19,583	34,125	1,608	22s
JAXEN	XPath engine	svn: 2008-12-03	12,438	8,399	689	10s
JODA-TIME	Date and time library	svn: 2009-08-17	25,909	48,178	3,497	48s
JTOPAS	Parser tools	1.0(SIR)	2,031	3,185	128	2s
XSTREAM	XML object serialization	svn: 2009-09-02	16,791	15,311	1122	20s

Lines of Code (LOC) are non-comment, non-blank lines as reported by `sloccount`.
For ASPECTJ, we only mutated the *org.aspectj.ajdt.core* package, which has 25,913 lines of source code and 6,828 lines of test code.

Table I
JAVALANCHE MUTATION OPERATORS

Replace numerical constant X by $X+1$, $X-1$, or 0.
Negate jump condition—which is equivalent to negating a conditional statement in the source code. (Since composite conditions compile into multiple jump instructions, this also negates individual subconditions.)
Replace arithmetic operator by another one, e.g. $+$ by $-$.
Omit method call—if the method has a return value, a default value is used instead, e.g. $x = Math.random()$ is replaced by $x = 0.0$.

statement coverage does this much more efficiently. For the remainder of the paper, we thus assume that mutations are only applied to statements that are executed by the test suite.

III. THE JAVALANCHE FRAMEWORK

As we wanted to assess the equivalence of mutations on projects of significant size, we developed the JAVALANCHE mutation testing framework [4] with a special focus on *automation and efficiency*. To this end, JAVALANCHE applies several optimizations, such as focusing on a subset of mutation operators (see Table I), using mutant schemata [9], using coverage data to reduce the number of tests that need to be executed, and allowing parallel execution of mutations.

Furthermore, JAVALANCHE allows to observe and trace the execution of mutations in order to determine their impact. Similar to an avalanche, where one small event can have a huge impact, JAVALANCHE aims at finding those mutations that have a big impact on the program run. The complete process for applying JAVALANCHE to a program is summarized in Figure 4.

IV. SUBJECT PROGRAMS

For our experiments we took seven open-source projects, from different application areas, listed in Table II. For each project, we took the most recent version from the version control system (column 3) — except for JTOPAS, which was taken from the software-artifact infrastructure repository (SIR) [10]. Each program comes with a JUnit test suite, from which we removed tests that fail, and tests whose outcome is dependent on the order or frequency of execution (which would be considered a flaw of the test suite).

Table III
RESULTS OF JAVALANCHE FOR THE 7 SUBJECT PROGRAMS.

Project Name	Number of Mutations	Covered Mutations	Covered & Detected
ASPECTJ	15,573	7,164	63%
BARBECUE	36,563	1,568	66%
COMMONS	19,404	14,609	85%
JAXEN	9,940	6,545	91%
JODA-TIME	23,920	17,678	87%
JTOPAS	1,921	1,512	85%
XSTREAM	9,230	6,725	90%

Table III shows the results for applying mutation testing without impact calculation to the subject programs. First JAVALANCHE determines all possible mutations for a program (column 2). From the total number of mutations, JAVALANCHE only considers those that are covered by at

Table IV
CLASSIFYING MUTATIONS MANUALLY.

Project Name	Non Equivalent mutations	Equivalent mutations	Average classification time
ASPECTJ	15 (75%)	5 (25%)	29m
BARBECUE	14 (70%)	6 (30%)	10m
COMMONS	6 (30%)	14 (70%)	8m
JAXEN	10 (50%)	10 (50%)	15m
JODA-TIME	14 (70%)	6 (30%)	20m
JTOPAS	10 (50%)	10 (50%)	7m
XSTREAM	8 (40%)	12 (60%)	11m
All	**77 (55%)**	**63 (45%)**	**14m28s**

least one test (column 3).[1] After executing all mutations, we get the *mutation score* for a project (column 4)—the number of mutations that are detected by the test suite (at least one test fails) divided by the total number of covered mutations.

V. MANUAL CLASSIFICATION

We saw that determining the equivalence of a mutation requires manual investigation. But how widespread is this problem in real programs? Offutt and Pan [11] reported 9.10% of equivalent mutants (relative to all mutants) for the 28-line `triangle` program. As we were interested in the extent of the problem on modern and larger programs, we applied mutation testing (Section III) to our seven subject programs, and investigated the results. For each of the seven projects, we randomly took 20 mutations from different classes that were not detected by the test suite for manual inspection. Then, we classified each mutation either

- as *non-equivalent*, as proven by writing a test case that detected the mutation; or
- as *equivalent* when manual inspection showed that the mutation does not affect the result of the computation.

A. Percentage of Equivalent Mutants

The results for classifying the 140 mutations for the seven projects are summarized in Table IV. Out of all classified mutations, 77 (55%) were non-equivalent and 63 (45%) were equivalent. The project with the highest ratio of non-equivalent mutants is ASPECTJ with 75%, while COMMONS had the lowest percentage with 30%. Such differences might also indicate differences in the quality of the test suites, as better test suites have a higher rate of equivalent mutations among their undetected mutations.

*On our sample of real-life programs, **45%** of the undetected mutations were equivalent.*

[1]Javalanche does not consider mutations that are only executed during class loading as covered. This explains the low coverage for BARBECUE.

B. Classification Time

The time required for classifying mutations as equivalent or non-equivalent varied heavily. While some mutations could be easily classified by just looking at the mutated statement, others involved examining large parts of the program for determining a potential effect of the mutated statement. This led to a maximum classification time of 130 minutes.

*On average, it took us 14 minutes 28 seconds to classify **one single** mutation for equivalence.*

C. Discussion

The number of 45% equivalent mutants is much higher than the 9% reported by Offutt and Pan, as their number is relative to all mutations, including the ones that are detected by the test suite. These mutations, however, are not of interest for improving the test suite, as they do not indicate a weakness of the test suite. If we also take the detected mutations into account, we found 7.39% of all mutations to be equivalent, which is roughly in line with the numbers reported by Offutt and Pan.

In practice, though, it is the percentage of equivalent mutations across the *undetected* mutations that matters—since these are the mutations that will be assessed by the developer. And here, 45% of equivalent mutants simply means 45% of wasted time. Even worse: While the percentage of equivalent mutations across *all* mutations stays fixed, the percentage of equivalent mutations across the *undetected* mutations increases as the test suite improves. This is due to the fact that an improved test suite detects more (non-equivalent) mutants. A perfect test suite would detect *all* non-equivalent mutants; hence, 100% of undetected mutants would be equivalent. In other words, as one improves the test suite, one has more and more trouble finding non-equivalent mutants among the undetected ones—with the growing effort as the test suite approaches perfection.

The percentage of equivalent mutants increases as the test suite improves.

VI. ASSESSING MUTATION IMPACT

Equivalent mutants are defined as having no observable impact on the programs output. This 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 being performed.* In particular, we can measure *changes in program behavior* between the mutant and the original version. The idea is that if a mutant impacts internal program behavior, it is also more likely to change external program behavior—and thus impacts the semantics of the program. If we focus on *mutations with impact,* we would thus expect to find fewer equivalent mutants.

How does one measure impact? *Weak mutation* [12] assesses whether a mutation changes the *local state* of a function or a component; if it does, it is considered detectable (and therefore non-equivalent). In this work, we are taking a more *global* stance and examine how the impact of a mutation propagates all across the system. To assess this impact degree, we consider two aspects:

- One aspect of impact is *control flow:* If a mutation alters the control flow of the execution, different statements will be executed in a different order—an impact that can be detected by using standard coverage measurement techniques.
- Another aspect of the behavior is the *data* that is passed between methods during the computation: If a mutation alters the data, different values would be passed to methods—an impact that can be detected by tracing the data that gets passed between methods.

In both cases, we measure the impact as *the number of changes detected all across the system;* as the number of impacted methods grows, so does the likelihood of the mutation to be generally detectable—and non-equivalent.

A. Impact on Coverage

In order to measure the impact of mutations on the control flow, we developed a tool 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. This gives us a set of lines that were covered together with frequency counts for every method of the program.

By comparing the coverage of the original execution with the coverage of the mutated execution, we can determine the *coverage difference.*

B. Impact on Return Values

Mutations with impact on the control flow manifest themselves in coverage differences, but it is also possible that a mutation has only impact on the data, which is not used in control flow affecting computations. In our manual investigation of 20 random undetected mutations without impact [8], we found two categories of non-equivalent mutations that had no impact on the code coverage:

- The first category are mutations that *changed return values* that were subsequently just passed around.
- The second category are mutations causing state changes that only manifest in a *change of the string representation* of an object.

Therefore, we decided to additionally trace the return values of public methods. We choose the public methods as they represent an object's communication to the environment.

Storing all return values of a program run would need a huge amount of disk space, which would be justifiable for one run of a test suite. As we plan to use this data for assessing each mutation, which involves several thousand executions of the test suite, we decided to abstract each return value into an integer value.

To obtain an integer value for returned objects, we compute its string-representation (by invoking `toString()`) and build the hash code for this string. For each primitive value, we store its natural integer representation; for 64-bit values, we compute the xor of the upper and lower 32 bits.

For each method, we store these integers and count how often they occurred. In this way, we end up with a set of integers for each method together with frequency counts.

Similar to coverage data, we can compare the sets of traced return values of the original execution with the mutated execution and obtain the *data difference*.

C. Impact on Invariants

In our previous work [4], we estimated the impact on the data using *dynamic invariants*. To this end, we learned dynamic invariants from the original program using DAIKON [13]. Then, we generated checkers that check those invariants at runtime and run the mutations, and finally obtained a set of violated invariants for each mutation.

The results showed that if a mutation violates dynamic invariants, it is very likely to be non-equivalent. However, mutations that violate dynamic invariants are rare. This finding motivated us to choose the impact on the return values as a more *fine-grained* view on impact.

D. Impact Metrics

The techniques defined above produce a set of *differences* between a run of the original and mutated program. Using these differences, we define *impact metrics* that quantify the difference between the original and mutated run:

- **Coverage impact**—the *number of methods* that have at least one statement that is executed at a different frequency in the mutated run than in the normal run—while leaving out the method that contains the mutation.
- **Data impact**—the *number of methods* that have at least one different return value or frequency in the mutated run than in the normal run—while leaving out the method that contains the mutation.
- **Combined coverage and data impact**—the *number of methods* that either have a coverage or data impact.

These metrics are motivated by the hypothesis that a mutation that has *non-local impact* on the program is more likely to change the observable behavior of the program. Furthermore, we would assume mutations that are undetected despite having impact across several methods to be particularly valuable for improving the test suite, as they indicate inadequate testing of multiple methods at once.

E. Distance Metrics

To further emphasize non-local impact, we use *distance metrics* that are based on the distance between the method

Table V
EFFECTIVENESS OF CLASSIFYING MUTATIONS BY IMPACT: PRECISION (LEFT) AND RECALL (RIGHT).

	Coverage Impact	Data Impact	Combined Impact	Coverage Distance	Data Distance	Combined Distance	Invariant Impact
ASPECTJ	72 / 87	72 / 87	72 / 87	77 / 67	67 / 67	67 / 67	100 / 7
BARBECUE	100 / 43	100 / 29	100 / 43	100 / 43	100 / 29	100 / 43	75 / 43
COMMONS	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	50 / 17
JAXEN	67 / 60	78 / 70	73 / 80	67 / 60	78 / 70	73 / 80	50 / 10
JODA-TIME	90 / 64	89 / 57	91 / 71	90 / 64	89 / 57	91 / 71	100 / 21
JTOPAS	100 / 70	43 / 30	64 / 70	100 / 60	50 / 30	67 / 60	100 / 10
XSTREAM	50 / 25	67 / 25	60 / 38	50 / 13	67 / 25	67 / 25	40 / 25
Total	**75 / 56**	**67 / 48**	**70 / 61**	**79 / 49**	**68 / 44**	**71 / 55**	**68 / 19**

First value in a cell gives the precision, the second the recall.

that contains the mutation and the method that has a coverage or data difference.

The distance between two methods M and N is computed as follows: First an undirected graph is built that contains a node for each method in the program. Two nodes V_N and V_M are connected if there exists a call from method M to N or vice versa. The distance between two methods is then the length of the shortest path between them.

Using this distance, we can define three distance metrics analogous to the impact metrics defined above:

- **Coverage distance**—For each method that has a coverage difference, we compute the shortest path to the method that contains the mutation. The coverage distance is then the length of the longest path.
- **Data distance**—For each method that has a data difference, we compute the shortest path to the method that contains the mutation. The data distance is then the length of the longest path.
- **Combined coverage and data distance**—the maximum of the data and coverage distance.

F. Equivalence Thresholds

Each of the metrics defined above (Sections VI-E and VI-D) produces a natural number that describes the impact of a mutation. As we want to automatically classify mutations that are less likely to be equivalent, we introduce a threshold t: A mutation is considered non-equivalent if and only if its impact is greater or equal to t.

VII. EVALUATION

We evaluated our approach in three experiments. First we applied our techniques to automatically classify mutations to the 140 manually classified mutations (Section VI-C). For our second experiment, we devised an evaluation scheme based on mature test suites. This automated evaluation scheme is presented in Section VII-B and compares the detection rate of *mutations with impact* (MI) and the *mutation with no impact* (MNI). Finally, we were interested if the mutations with the highest impact are less likely to be equivalent. We therefore ranked the mutations according to their impact and looked at the highest ranked mutations (Section VII-C), both for the manually classified mutations and the ones detected by the test suites.

A. Impact of the Manually Classified Mutations

In the first experiment, we wanted to evaluate our hypothesis that mutations with impact on coverage or return values are less likely to be equivalent. We therefore determined the coverage and data differences and computed the impact (Section VI-D) and distance metrics (Section VI-E) for the 140 manually classified mutations, and automatically classified the mutations using the all metrics with a threshold of 1. Then we compared these results to the actual results of the manual classification.

To quantify the effectiveness of the classification, we compute its *precision* and *recall*:

- The *precision* is the percentage of mutations that are correctly classified as non-equivalent. A high precision implies that the results of a classification scheme contain *few false positives*—that is, most mutations classified as non-equivalent are indeed non-equivalent.
- The *recall* is the percentage of non-equivalent mutations that are correctly classified as such. A high recall means that there are *few false negatives*—that is, a high ratio of the non-equivalent mutations was retrieved by the classification scheme.

While it is easy to achieve a 100% recall (just classify all mutations as non-equivalent), the challenge is to achieve both a high precision and a high recall.

The results for evaluating the different metrics on the classified mutants are summarized in Table V. Each entry gives first the precision of the metric, and then its recall. Besides the metrics defined above, the table also contains the results for the impact on dynamic invariants (Section VI-C).

From the average results (last row), we can see that all techniques have a high *precision*, ranging from 68% for the data distance and invariant metric up to 79% for coverage distance. This means that 68%–79% of all mutations classified as non-equivalent actually are non-equivalent. In comparison a simple classifier that classifies all mutations

Table VI
ASSESSING WHETHER MUTANTS WITH IMPACT ON COVERAGE ARE DETECTED BY TESTS.

Project Name	Number of MIs	Number of MNIs	MIs detected	MNIs detected	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	5,531	1,661	76%	20%	100%	100%	99%
BARBECUE	1,045	528	83%	32%	100%	97%	99%
COMMONS	10,061	4,559	97%	58%	98%	99%	99%
JAXEN	5,997	548	97%	26%	100%	100%	100%
JODA-TIME	15,883	2,037	95%	18%	100%	100%	99%
JTOPAS	1,362	150	93%	5%	100%	100%	100%
XSTREAM	5,940	788	97%	39%	100%	100%	100%

MI = Mutation with Impact, MNI = Mutation with No Impact.

Table VII
ASSESSING WHETHER MUTANTS WITH IMPACT ON DATA ARE DETECTED BY TESTS.

Project Name	Number of MIs	Number of MNIs	MIs detected	MNIs detected	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	5,186	2,006	80%	19%	100%	99%	99%
BARBECUE	956	617	92%	25%	100%	97%	99%
COMMONS	7,861	6,759	98%	70%	97%	98%	98%
JAXEN	6,005	540	95%	46%	100%	100%	100%
JODA-TIME	15,173	2,747	91%	55%	100%	100%	99%
JTOPAS	1,286	226	94%	31%	100%	100%	100%
XSTREAM	5,543	1,185	95%	64%	100%	100%	100%

as non-equivalent, would have a precision of 54%. Thus, the metrics improve over the simple approach by 14-25 percentage points.

Mutations with impact on coverage and data have a likelihood of 58–79% to be non-equivalent, compared to 54% across all mutations.

When we look at the results per project, COMMONS is a clear outlier, with a precision and recall of zero for almost all metrics. This is due to several mutations that alter the *caching behavior* of some methods. Although they are manually classified as equivalent, because the methods still return a correct object, they have a huge impact because new objects are created at every call instead of taking them from the cache. When we look at the result of the manual classification for COMMONS (Table IV), we also see that it is the project with the highest number of equivalent mutants—which might indicate that most mutations not detected by the test suite are equivalent.

The recall values for the coverage and data metrics range from 44% for data distance to 61% for the combined impact metric. Both the combined impact and combined distance metric have a higher recall than the two metrics they are based on. This, however, comes at a cost of a lower precision. Furthermore, all coverage and data metrics also have a far better recall than the earlier invariant-based techniques [4] (recall of 19%).

Only 19% of non-equivalent mutants are likely to impact invariants.

There is always a trade off between precision and recall. Increasing one of both values decreases the other. The simple classifier, for example, has a recall of 100% by definition while it only has a precision of 55 %. On the other hand, we can also increase the precision of our metrics by rising the threshold, e.g. when we use a threshold of two for the coverage impact, we get a precision of 81% percent and a recall of 44%.

All distance metrics have a lower recall than their corresponding impact metrics. This is because some mutations impact methods that are not connected via method calls—and thus, the impact propagates through state changes.

B. Impact and Tests

Besides our evaluation on the manually classified mutations, we also wanted a broader objective evaluation scheme that can be automated. However, in order to automatically determine the equivalence of a mutation we either need a test suite that detects all non-equivalent mutations, or an oracle that tells the equivalence of a mutation. Unfortunately, obtaining such a test suite or an oracle is infeasible. Thus, we decided to base our automated evaluation scheme on the existing mature test suites of the projects.

The rationale for our evaluation is as follows: A mutation classification scheme helps the programmer when it detects many non-equivalent and fewer equivalent mutants. For every mutant that is detected by the test suite, we know for sure that it is non-equivalent. If we can prove that classification scheme has a high precision on the mutations that are detected by the test suite, this might also hold for the mutations that are not detected by the test suite.

Thus, we applied the impact metrics to all mutations in each project and evaluated them on the mutations detected by the test suite. The results are given in Tables VI to VIII.

Table VIII
ASSESSING WHETHER MUTANTS WITH COMBINED COVERAGE AND DATA IMPACT ARE DETECTED BY TESTS.

Project Name	Number of MIs	Number of MNIs	MIs detected	MNIs detected	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	5,200	1,992	81%	17%	100%	100%	99%
BARBECUE	1,142	431	81%	25%	100%	97%	99%
COMMONS	10,467	4,153	95%	59%	98%	98%	99%
JAXEN	6,063	482	95%	41%	100%	100%	100%
JODA-TIME	15,841	2,079	91%	43%	100%	100%	99%
JTOPAS	1,388	124	92%	6%	100%	100%	100%
XSTREAM	6,059	669	94%	52%	100%	100%	100%

For each project and impact metric, we determined the number of mutations that had impact (MIs in column 2), and the number that had no impact (MNIs in column 3). For the MIs and MNIs, we then computed the ratio that was detected by the test suite (column 4 and 5).

In Section VI-F we saw that we need a *threshold* t when to consider a mutation to have an impact according to the underlying metric. As our manual classification showed 45% of the undetected mutations to be equivalent, we automatically set t such that at most 45% of the not detected mutations are classified as having no impact.

The ratio of mutations with impact ranges from 54% for COMMONS and data impact (Table VII) up to 93% for JAXEN and the combined impact metric (Table VIII). The number of mutations with impact that are detected is around 90% on average (i.e., at most 10% are equivalent), while the average ratio of mutations with no impact ranges from 28% for coverage impact to 45% for data and combined impact. These results indicate that the impact metrics classify the mutations with a high precision, while the coverage impact metric has the highest precision.

Of the mutations that have impact on coverage or data, at most 10% are equivalent.

C. Mutations with High Impact

In the previous experiments, we saw that mutations with impact are more likely to be non-equivalent. Besides that, we were interested whether mutations with a *high impact* are more likely to be non-equivalent.

To evaluate this hypothesis we did two experiments. First we ranked the mutations that were detected (as described in Section VII-B) by their impact, picked the top 5, 10, and 25 percent, and checked how many of them were non-equivalent. In a second experiment, we ranked the mutations from the manual classification according to their impact for the different impact metrics. Then, we picked the 15, 20, and 25% highest ranked mutations out of all mutations classified as non-equivalent by the metric, and checked if they were correctly classified.

The results for the first experiment (for mutations detected by the test suite) can be found in the last 3 columns of Tables VI - VIII. For many projects and impact metrics the 25% of mutations with the highest impact are *all* detected. If not all are detected, at least 98% of them are. For the impact on invariants [4], we observed a similar trend, but not as pronounced as for the data and coverage impact metrics.

Table IX
FOCUSING ON MUTATIONS WITH THE HIGHEST IMPACT: PRECISION OF THE CLASSIFICATION

Impact Metric	Top 15%	Top 20%	Top 25%
Coverage Impact	88%	91%	93%
Data Impact	88%	91%	86%
Combined Impact	90%	85%	76%
Coverage Distance	86%	80%	75%
Data Distance	88%	80%	85%
Combined Distance	89%	75%	80%

Table IX shows the results for the manual classification. For all impact metrics 75% or more out of the top 25% are non-equivalent. Compared to the precision results in Table V, picking the 25% mutations with the highest impact yields a higher ratio of non-equivalent mutations, than choosing mutations with impact in no specific order. In this setting again the *coverage impact* metric performs best. When we choose the top 25% ranked mutations, 93% of them are non-equivalent.

Of the mutations with the highest coverage impact, more than 90% are non-equivalent.

The results for the detected mutants indicate that a high impact strongly correlates with non-equivalence, and the results for the manually classified mutations confirm this finding for the undetected mutants.

In practice, this means that focusing on the mutations with the highest impact will yield the fewest amount of equivalent mutants. The question is whether mutations with a high impact are also the most *valuable* mutations—that is, whether they uncover the most errors, or the most important errors. Our intuition tells us that if I can make a change to a component that impacts several other components, yet the test suite does not detect it, such a change has a higher chance to be valuable than a change whose impact is hardly measurable. The relationship between impact and value of mutations remains to be assessed and quantified, though.

VIII. 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. In our studies, we have examined 20 sample mutations from seven non-trivial Java 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 [14], [3], [15], [1]. 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.
- **Threats to *internal validity*** concern our ability to draw conclusions about the connections between our independent and dependent variables. Regarding the manual classification (Section V), our own assessment may be subject to errors, incompetence, or bias. At the time we conducted the assessment, we did not know how the mutations would score in terms of impact; additionally, to counter these threats, all our assessments are publicly available (Section X).
  For assessing mutations based on coverage (Sections VI-C and VII-B), our implementation could contain errors that affect the outcome. To control for these threats, we ensured that earlier stages had no access to data used in later stages. We advise and support independent confirmation of our results and make the framework and necessary data publicly available (Section X).
- **Threats to *construct validity*** concern the appropriateness of our measures for capturing our dependent variables. Regarding the manual classification of mutations (Section V), being able to write a test is the ultimate measure whether a mutant is non-equivalent. When classifying mutations based on impact (Section VI-C), we directly provide the information as required by the programmer. Finally, in Section VII-B, our assumption that the test suite measures real defects is an instance of the "competent programmer hypothesis" also underlying mutation testing [16]. This hypothesis may be wrong; however, the maturity and widespread usage of the subject programs should suggest sufficient competence. Further studies will help completing our knowledge on what makes a test suite adequate.

IX. Related Work

The idea of using impact on executions to assess mutations was first presented in a short workshop paper [8], where we framed the problem and showed preliminary results for the JAXEN project. This paper adds impact on return values and method distance as an additional factor, and brings a full-fledged evaluation.

In [4], we experimented with an alternate approach, based on dynamic invariants as learned from the test suite. We found that mutations that violate dynamic invariants also have a higher likelihood to be non-equivalent. Our current approach, though, is more efficient to use, detects even minuscule alterations in behavior, and produces better results.

The problem of equivalent mutants was also diagnosed and tackled by other researchers. Baldwin and Seyward [5] proposed the usage of compiler optimization techniques to detect equivalent mutants. The idea of this approach is that some equivalent mutants are optimizations or de-optimizations themselves, or can be optimized away by a compiler. These techniques were later implemented by Offutt and Craft [6]. The results indicate that the techniques can detect about 10% of equivalent mutants.

Offutt and Pan [11] realized that detecting equivalent mutants is an instance of the *feasible path problem* and presented an algorithm based on mathematical constraints. To be non-equivalent, a mutation (1) must be reachable, (2) cause an incorrect state after it is executed, (3) and must have an effect on the final state. If a mutation cannot fulfill any of these conditions, then it must be equivalent.

These techniques are orthogonal to ours; if it can be statically proven that a mutation is equivalent, we do not need to compute its impact, and can focus on those mutations that cannot be handled with the static approaches. Another question is how well the static approaches scale. While we evaluated our impact metrics on programs of significant size, Offutt and Pan's [11], for example, evaluated their technique on 11 programs with 11–30 executable statements.

Other approaches to the problem of equivalent mutants include aiding the programmer in detecting equivalent mutants using program slicing [15], or generating less equivalent mutants by using genetic algorithms [17] or higher order mutants [18], [19].

Weak mutation as proposed by Howden [12] considers a mutation to be detected when its containing component computes a different result for at least one test case. Our data impact metric also is also based on these differences. However, instead of defining a mutation's result by these differences, we are trying to predict its equivalence. As we can see from the results in Section VII-A, there are cases where a mutation causes a component to produce different results, while the whole program produces a correct result.

X. Contributions and Conclusion

Our study shows that equivalent mutants are a serious problem that effectively inhibits widespread usage of mutation testing, as demonstrated on a sample of 140 mutations on seven Java programs. However, checking whether a mutation impacts coverage is an effective means to separate equivalent from non-equivalent mutations. In addition, the technique is easy to implement and to deploy. All in all, this paper makes the following contributions:

- **A case study on the abundance of equivalent mutants.** To our knowledge, the present study is the first assessing the percentage of equivalent mutants on a set of seven real-life programs. The percentage of equivalent mutants ranges from 25% to 70%.
- **Evidence into the effectiveness of checking coverage.** If a mutation changes coverage, it has a 75% chance to be non-equivalent. This paper substantiates this claim on all seven programs as shown above.
- **A benchmark data set for further studies.** We have made our framework and all experiment data publicly available (see below). Further researchers can thus use our classified mutations to evaluate their own techniques—and to improve upon our results.

Our own work does not stop at this point either. Our future work will concentrate on the following topics:

- **How effective is mutation testing in improving test suites?** By effectively weeding out equivalent mutants, we can run large case studies comparing mutation testing to classical coverage criteria—and assess how the value of mutations is related to their impact.
- **How can we find mutants with the highest impact?** If a mutation has a high impact on the program execution, but is undetected by the test suite, it may be particularly valuable. We are investigating genetic algorithms to systematically generate such mutants.
- **Are components with high impact mutations defect-prone?** If mutations in a component have a particularly high impact, this may indicate that changes to the component are particularly risky. We want to study how the impact of mutations propagates across components, and whether the impact can be used to predict defects.

The JAVALANCHE framework as well as data sets from the experiments in this paper are available at

`http://www.javalanche.org/`

ACKNOWLEDGMENTS

Gordon Fraser, Yana Mileva, Jeremias Rößler, Andrzej Wasylkowski as well as the anonymous reviewers provided helpful feedback on earlier revisions of this paper. Special thanks go to Bernhard Grün for contributing to the JAVALANCHE coverage checker.

REFERENCES

[1] 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*, 2005, pp. 402–411.

[2] P. J. Walsh, "A measure of test case completeness," Ph.D. dissertation, Binghamton University, 1985.

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

[4] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA 2009: Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[5] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations," Yale University, Department of Computer Science, Tech. Rep. 276, 1979.

[6] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification, and Reliability*, vol. 4, pp. 131–154, 1994.

[7] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, 1982.

[8] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Mutation '09: 4th International Workshop on Mutation Analysis*, 2009.

[9] 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*, 1993, pp. 139–148.

[10] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[11] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[12] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. on Software Engineering*, vol. 8, no. 4, pp. 371–379, 1982.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[14] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *COMPASS '96: Proceedings 11th Conference on Computer Assurance*, 1996, pp. 224–236.

[15] R. Hierons and M. Harman, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[16] R. A. DeMillo, R. J. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[17] 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*, vol. 3103, 2004, pp. 1338–1349.

[18] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[19] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *SCAM 08: Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 249–258.