SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2013; **23**:353–374
Published online 4 April 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1473

Covering and Uncovering Equivalent Mutants

David Schuler*,† and Andreas Zeller

Saarland University, Saarbrücken, Germany

SUMMARY

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 are also mutations that 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 5000 to 100 000 lines of code, we found that (i) the problem is serious and widespread—about 45% of all undetected mutants turned out to be equivalent; (ii) manual classification takes time—about 15 min per mutation; (iii) coverage is a simple, efficient and effective means to identify equivalent mutants—with a classification precision of 75% and a recall of 56%; and (iv) 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. Copyright © 2012 John Wiley & Sons, Ltd.

Received 24 September 2010; Revised 25 December 2011; Accepted 23 February 2012

KEY WORDS: mutation testing; code coverage; dynamic analysis

1. 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 do mutation testing on 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].

*Correspondence to: David Schuler, Saarland University, Saarbrücken, Germany.
†E-mail: ds@cs.uni-saarland.de

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 min 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 most *non-equivalent* mutants are already caught by the test suite.

The problem gets worse as the test suite improves. As 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 on the basis of 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 2). After introducing our JAVALANCHE mutation framework (Section 3) and the subject programs (Section 4), our classification study gives details on the ubiquity of equivalent mutants (Section 5). We then describe how to assess the impact of mutants on coverage (Section 6), followed by an evaluation of the approach (Section 7). After discussing the threats to validity (Section 9), we explore the related work (Section 10) and close with conclusion and future work (Section 11).

2. EQUIVALENT MUTANTS

One usage scenario 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 it is checked 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 so that previously undetected mutants are detected. There are several reasons why a test suite might fail to detect a mutation, which determine the usefulness to the programmer:

1. The mutation may not change the program's semantics and *cannot be detected*. These equivalent mutants cannot help improve 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 because they provide indicators to improve the test suite, which other coverage metrics might not provide. If a mutation is covered but not detected, this means either that the tests do not check the results well enough or that the input data are not chosen carefully enough to trigger the erroneous behaviour.

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

```
...
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.

```
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.

2.1. 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 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, one can detect the mutation.

2.2. 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.

2.3. 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 so that it always

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

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

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 $x = 0.0$.

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 a full-fledged mutation testing. A simple 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.

3. 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:

Focusing on a subset of 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 [9]. JAVALANCHE therefore uses the same small set of operators as proposed by Offutt [10] and later adapted by Andrews *et al.* [1], listed in Table I.

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 efficiently. To reduce the number of generated versions, we use mutant schema generation [9]. 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.

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.

4. 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

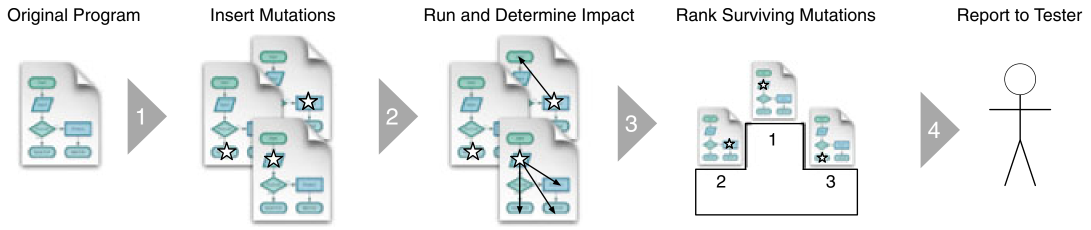

Figure 4. The JAVALANCHE process. After generating mutations (Step 1), JAVALANCHE runs the test suite on each and ranks mutations by their impact on data and coverage (Steps 2 and 3). Finally, the tester (Step 4) improves the test suite to detect the top-ranked mutations.

Table II. Description of subject programs.

Project name	Description	Version	Program size (LOC)	Test code size (LOC)	Number of tests	Test suite runtime (s)
ASPECTJ	AOP extension to Java	cvs: 2007-09-15	94 902	14 736	336	9
BARBECUE	Bar code creator	svn: 2007-11-26	4 837	3 293	153	3
COMMONS	Helper utilities	svn: 2009-08-24	19 583	34 125	1608	22
JAXEN	XPath engine	svn: 2008-12-03	12 438	8 399	689	10
JODA-TIME	Date and time library	svn: 2009-08-17	25 909	48 178	3497	48
JTOPAS	Parser tools	1.0(SIR)	2 031	3 185	128	2
XSTREAM	XML object serialization	svn: 2009-09-02	16 791	15 311	1122	20

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 6828 lines of test code.

Table III. Results of JAVALANCHE for the seven subject programs.

Project name	Covered mutants	Number of mutants	Mutation score (%)	Mutation score for covered (%)
ASPECTJ	17 328	9 168	35.41	66.93
BARBECUE	17 631	1 687	5.68	59.40
COMMONS	14 716	13 748	79.16	84.73
JAXEN	9 285	6 333	48.39	70.95
JODA-TIME	21 052	13 293	51.42	82.83
JTOPAS	1 678	1 400	67.64	81.07
XSTREAM	8 240	6 488	68.54	87.05
Total	89 930	52 117	45.47	78.45

3)—except for JTOPAS, which was taken from the software-artifact infrastructure repository (SIR) [11]. 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 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 least one test (column 3). [‡] After executing all mutations, we get the *mutation score* for a project (column 4) and the mutation score relative to the covered mutations (column 5)—the number of mutations that are detected by the test suite (at least one test fails) divided by the total number of covered mutations.

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

Table IV. Classifying mutations manually.

Project name	Non-equivalent mutants	Equivalent mutants	Average classification time
ASPECTJ	15 (75%)	5 (25%)	29 min
BARBECUE	14 (70%)	6 (30%)	10 min
COMMONS	6 (30%)	14 (70%)	8 min
JAXEN	10 (50%)	10 (50%)	15 min
JODA-TIME	14 (70%)	6 (30%)	20 min
JTOPAS	10 (50%)	10 (50%)	7 min
XSTREAM	8 (40%)	12 (60%)	11 min
All	77 (55%)	63 (45%)	14 min 28 s

5. MANUAL CLASSIFICATION

We saw that determining the equivalence of a mutant requires manual investigation. But how widespread is this problem in real programs? Offutt and Pan [12] 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 3) 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 detects the mutation, or
- as *equivalent* when manual inspection showed that the mutation does not affect the result of the computation.

5.1. 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%, whereas 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. Notice that the ratio of 45% of equivalent mutants relates to the undetected ones. Relative to all mutants, we obtain a ratio of 7.39% of equivalent mutants.

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

5.2. Classification Time

The time required for classifying mutations as equivalent or non-equivalent varied heavily. Whereas 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 min.

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

5.3. Mutation Operators

JAVALANCHE generates mutations by using different mutation operators as introduced in Table I. In order to check the relation between the equivalence of a mutants and its underlying mutation operators, we grouped the 140 manually classified mutations according to their operator. The results are summarized in Table V. Some operators produce far more mutants than others (column 2). For example, the operator *replace numerical constant* produces over half of the mutations in our

Table V. Classification results per mutation operator.

Mutation operator	Number of mutants	Non-equivalent mutants	Equivalent mutants
Replace numerical constant	78	34 (44%)	44 (56%)
Negate jump condition	12	10 (83%)	2 (17%)
Replace arithmetic operator	7	3 (43%)	4 (57%)
Omit method call	43	30 (70%)	13 (30%)

sample. However, we can also check the ratios of non-equivalent (column 3) and equivalent (column 4) mutants for the operators. Here, we see that for the operators *replace numerical constant* and *replace arithmetic operator*, which manipulate data, around 57% of all produced mutants are equivalent, whereas the operators *negate jump condition* and *omit method call*, which manipulate the control flow, only produce around 30%.

Mutation operators that change the control flow produce fewer equivalent mutants than those that change the data.

5.4. 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—because 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, whereas 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.

6. ASSESSING MUTATION IMPACT

Equivalent mutants are defined to have no observable impact on the program's output. This impact of a mutation can be assessed by checking the program state at the end of a computation, just like 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 behaviour* between the mutant and the original version. The idea is that if a mutant impacts internal program behaviour, it is more likely to change external program behaviour. Thus, it is also more likely to impact 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* [13] 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 are executed. This is an impact that can be detected by using standard coverage measurement techniques.
- Another aspect of the behaviour is the *data* that are passed between methods during the computation: If a mutation alters the data, different values are passed to methods—an impact that can be detected by tracing the data that gets passed between methods.
- Another aspect of the behaviour concerns the *data* that is passed between methods during the computation: If a mutation alters the data, different values are passed between methods. This is an impact that can be detected by tracing the data that are 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.

6.1. 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 execution frequency for each statement that is executed for each test case and each mutation. Note that the data collected by our tool are very similar to *statement coverage*, which computes whether a statement is executed or not. In addition to statement coverage, our tool also stores the execution frequency of a statement.

Running the complete test suite of a program and tracing its coverage provide us with a set of lines that were covered together with frequency counts for every test case of the test suite. By comparing the coverage of a run of the original version with the coverage of the mutated version, we can determine the *coverage difference*.

6.2. 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 a 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 is mutations that *changed return values* that were subsequently never used in computations, but were passed to the output of the program.
- The second category is mutations causing state changes that only manifest in a *change of the string representation* of an object (as described below).

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

Storing all return values of a program run would require a huge amount of disk space. For example, objects can cover huge parts of the program state through references. The storage of all this data for each return value might be justifiable for one run of a test suite. As we plan to use these 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.

For each public method that has a return value, we store these integers and count how often they occur. 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*.

6.2.1. Abstracting Return Values. To obtain an integer value for returned Java objects, we compute its string representation by invoking `toString()`. Then, we remove substrings that represent memory locations, as returned by the standard implementation of the `toString()` method in `java.lang.Object`, because these locations change between different runs of the program even though the computed data stay the same. From the resulting string, we then compute the hash code. Thereby, we obtain an integer value that characterizes the object.

For each primitive value (`int`, `char`, `float`, `short`, `boolean`, `byte`), we store its natural integer representation; for 64-bit values (`long`, `double`), we compute the exclusive or the upper and lower 32 bits.

6.3. *Impact on Invariants*

In our previous work [4], we estimated the impact on the data by using *dynamic invariants*. To this end, we learned dynamic invariants from the original program by using DAIKON [14]. 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. Our abstraction over the return values is more fine grained because it takes into account every observed return value. The dynamic invariants, however, hold for all observed runs, and they are discarded as soon as one violation is observed.

6.4. *Impact Metrics*

The techniques defined above produce a set of *differences* between a run of the test suite on 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 have either 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 behaviour of the program. Furthermore, we would assume that mutations that are undetected despite having an impact across several methods can be considered as particularly valuable for the improvement of a test suite, as they indicate inadequate testing of multiple methods at once.

6.5. *Distance Metrics*

To further emphasize non-local impact, we use *distance metrics* that are based on the distance between the method that contains the mutation and the method that has a coverage or data difference.

The distance between two methods is the length of the shortest path between them in the *undirected call graph*. The undirected call graph is a variant of the traditional call graph that contains a node V_M for each method M in the program. There is an edge between two nodes V_M and V_N if there exists a call from method M to N or vice versa.

By 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 distance via the shortest path to the method that contains the mutation. The coverage distance is the longest of these distances because this represents the furthest coverage impact this mutation has.
Data distance—For each method that has a data difference, we compute the distance via the shortest path to the method that contains the mutation. The data distance is then the longest of these distances, as this represents the furthest data impact this mutation has.
Combined coverage and data distance—The maximum of the data and coverage distance.

6.6. Equivalence Thresholds

Each of the metrics defined above (Sections 6.5 and 6.4) 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 mutant is considered to have an impact if and only if its impact metric is greater than or equal to t.

7. EVALUATION

We evaluated the coverage and data-based impact metrics in three experiments. First, we applied our techniques to automatically classify mutants to the 140 manually classified mutants (Section 7.1). For our second experiment, we devised an evaluation scheme on the basis of mature test suites. This automated evaluation scheme is presented in Section 7.2 and compares the detection rate of *mutants with impact* (MI) and the *mutants with no impact* (MNI). Finally, we were interested if the mutants with the highest impact are less likely to be equivalent. We, therefore, ranked the mutants according to their impact and looked at the highest-ranked mutants (Section 7.3), both for the manually classified mutants and the ones detected by the test suites.

7.1. Impact of the Manually Classified Mutations

In the first experiment, we wanted to evaluate our hypothesis that MI on coverage or return values are less likely to be equivalent. We, therefore, determined the coverage and data differences and computed the impact (Section 6.4) and distance metrics (Section 6.5) for the 140 manually classified mutants and automatically classified them by using a threshold of 1 for all metrics. Then, we compared these results with the actual results of the manual classification. To quantify the effectiveness of the classification, we computed its *precision* and *recall*:

- The *precision* is the percentage of mutants that are correctly classified as non-equivalent, that is, the mutant has an impact and is non-equivalent. A high precision implies that the results of a classification scheme contain *few false positives*—that is, most mutants 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.

Although it is easy to achieve a 100% recall (just classify all mutants 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 VI. 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 6.3).

When considering 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 to 79% for coverage distance. This

Table VI. Effectiveness of classifying mutations by impact: precision (left) and recall (right).

	Coverage impact	Data impact	Comb. impact	Coverage distance	Data distance	Comb. 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

The first value in a cell gives the precision and the second the recall.

means that 68% to 79% of all mutations classified as non-equivalent actually are non-equivalent. In comparison, a simple classifier that classifies all mutations as non-equivalent would have a precision of 54%. Thus, the metrics improve over the simple approach by 14 to 25 percentage points.

Mutations with impact on coverage and data have a likelihood of 68 to 79% to be non-equivalent, compared to 54% across all mutations.

When we look at the results of each 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 behaviour* 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 that are 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 technique [4], which has a recall of only 19%.

Coverage and data impact have better recall values than invariant impact.

There is always a trade-off between precision and recall. Increasing either value decreases the other one. 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 increasing the threshold; for example, when we use a threshold of 2 for the coverage impact, we get a precision of 81% and a recall of 44%.

All distance metrics have a lower recall than their corresponding impact metrics. A reason for this is that some mutations impact methods that are not connected via method calls. In these cases, the impact propagates through state changes.

7.1.1. Sensitivity Analysis. The previous results were all computed with a threshold of 1. Thus, it is not clear how the metrics perform when a higher threshold is used. In order to analyse its influence, we repeated the previous experiment with varying thresholds. Figure 5 shows a graph for each distance metric. The x-axis displays the threshold, and the y-axis shows the percentage for precision and recall. A continuous line stands for the precision whereas a dashed line stands for the recall. For a higher threshold, we would expect higher precision as only the mutations with a higher impact are considered. This comes at the cost of a lower recall as fewer mutations are considered in total. By looking at the graphs, we can see that all metrics follow this trend. For the distance-based metrics, however, this trend only holds up to a threshold of 15. For thresholds greater than 15, both recall and precision are 0 because our sample contains no non-equivalent mutation with a distance impact greater than 15. In contrast to the distance-based metrics, the trend holds for the coverage, data and combined impact up to a threshold of 100.

Coverage and data metrics are more stable for higher thresholds than distance based metrics.

7.2. 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 need either 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

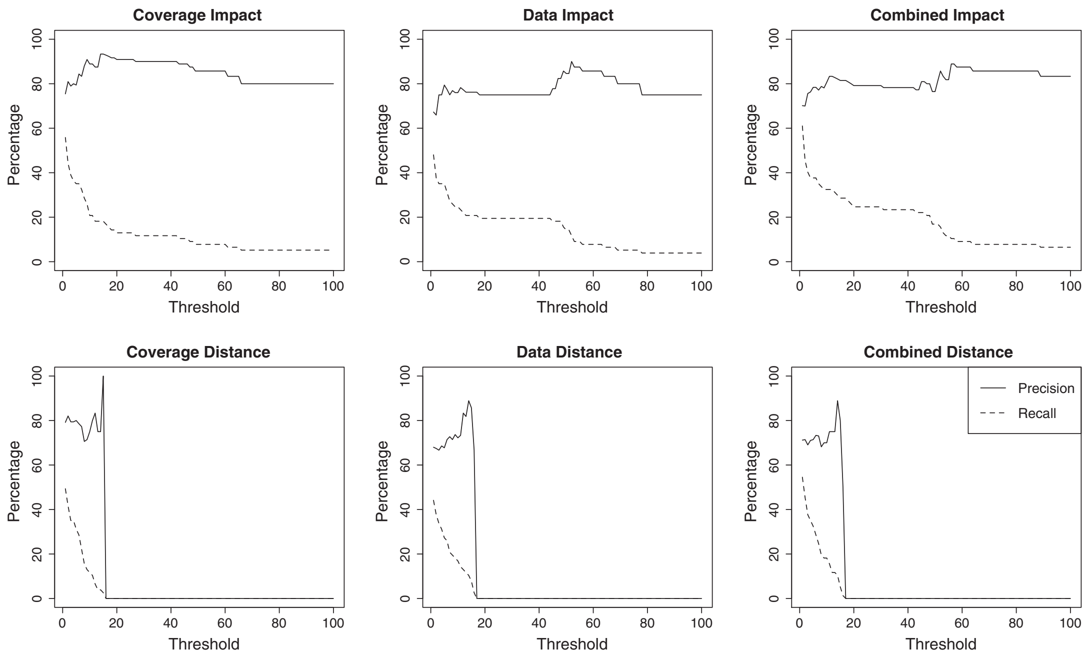

Figure 5. Precision and recall of the impact metrics for different thresholds.

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 a 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 VII–IX.

For each project and impact metric, we determined the number of mutations that had an 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 6.6, we saw that we need a *threshold t* when we 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* so that at most 45% of the undetected mutations are classified as having no impact.

Table VII. 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 VIII. 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

MI, mutation with impact; MNI, mutation with no impact.

Table IX. 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

MI, mutation with impact; MNI, mutation with no impact.

The ratio of mutations with impact ranges from 54% (7861 out of 14 620) for COMMONS and data impact (Table VIII) up to 93% (6063 out of 6545) for JAXEN and the combined impact metric (Table IX). The number of mutations with impact that are detected is around 90% on average (i.e., at most 10% are equivalent). 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, whereas the coverage impact metric has the highest precision.

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

7.2.1. Sensitivity Analysis. We investigated the sensitivity of our approach to the threshold by repeating the previous experiment for the coverage impact measure and by varying values for the threshold. The results are shown in Figure 6. For each of the seven projects, there is one graph, where the x-axis gives the different threshold values and the y-axis the percentage for three different measurements: (i) a solid line for the mutations with impact, (ii) a dashed line for the ratio between detected and undetected mutations with impact and (iii) a dotted line for the ratio between detected and undetected mutations with no impact.

The percentage of mutations with impact declines for all projects, and for BARBECUE, COMMONS and JTOPAS, there are no mutations with an impact greater than 200, whereas for JAXEN, JODA-TIME and XSTREAM, there are some mutations with an impact greater than 800. The ratio of detected mutations with impact rises up to 100% for all mutations when higher thresholds are used. This is because only mutations with a big impact (mostly mutations that cause exceptions) are considered. The ratio of detected mutations with no impact also rises for higher thresholds, because more mutations are considered to have no impact.

At higher thresholds 100% of the mutations with impact are detected.

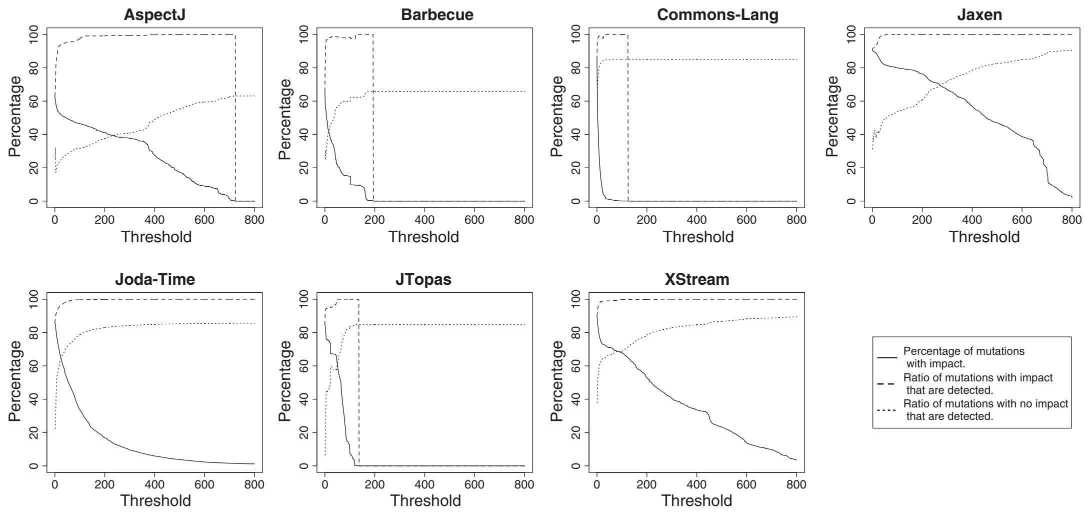

Figure 6. Percentage of mutations with impact and detection ratios of mutations with and without impact for varying threshold.

7.2.2. Mutation Operators. As the results from the manual classification indicated that there is a difference in ratio between equivalent and non-equivalent mutations for different mutation operators (Section 5.3), we were interested in the detection ratios for mutation operators. To this end, we grouped the results for combined coverage and data impact by mutation operator and combined the results for all projects. Table X summarizes the results. The number of mutations (column 2) that are produced for an operator ranges from 2359 for *replace arithmetic operator* to 22 457 for *replace numerical constant*. Similar to the results concerning equivalence (Section 5.3), there is a difference between operators that manipulate the control flow (*negate jump condition* and *omit method call*) and operators that manipulate data (*replace numerical constant* and *replace arithmetic operator*).

Compared with the data manipulating operators, the control flow manipulating operators have higher detection rate (87% to 90% vs 77% to 81% in column 3), more mutations with impact (92% to 94% vs 85% to 87% in column 4) and higher detection rates for mutations with impact (88% to 91% vs 84% to 88% in column 5).

Mutations that manipulate the control flow have higher impact and higher detection rates than mutations that manipulate data.

7.3. 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 in whether mutations with a *high impact* are more likely to be non-equivalent.

Table X. Detection ratios for different operators.

Mutation operator	Number of mutants	Detected mutants (%)	MI (%)	MI detected (%)
Replace numerical constant	22 457	76.96	85.08	83.59
Negate jump condition	9 790	90.41	91.65	90.69
Replace arithmetic operator	2 359	80.63	86.90	87.95
Omit method call	21 484	86.55	94.42	88.11
Total	56 090	83.14	89.88	86.85

MI, mutation with impact.

Table XI. 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%

To evaluate this hypothesis, we did two experiments. First, we ranked the mutations that were detected (as described in Section 7.2) by their impact, picked the top 5%, 10%, and 25% 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% of the 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 three columns of Tables VII–IX. 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 distinctive as for the data and coverage impact metrics.

Table XI shows the results for the manual classification. For all impact metrics, 75% or more out of the top 25% are non-equivalent. Compared with the precision results in Table VI, picking the 25% of mutations with the highest impact attains a higher ratio of non-equivalent mutations than choosing mutations with impact in no specific order. In this setting, the *coverage impact* metric performs best again. 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 most errors or the most important errors. Our intuition tells us that if we can make a change to a component that impacts several other components, while the test suite still 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.

8. MUTATION OPERATORS

JAVALANCHE uses a reduced set of mutation operators (as described in Table I) in its standard configuration. This standard set of operators was also used for the experiments presented in this paper. The set was adapted from the mutation operators used by Andrews *et al*. [1], and their set again was adapted from the sufficient mutation operators as determined by Offutt *et al*. [10]. In their study, Offutt *et al*. determined a reduced set of mutation operators that can be used to obtain results similar to applying all possible mutation operators. In an empirical study on 10 Fortran programs, they showed that five out of 22 mutation operators are sufficient; that is, test suites that detect all mutants produced by these operators detect 99.5% of all mutants.

Compared with other research, this choice of different operators might influence our results because the operators might produce mutations that are easier or more difficult to detect, and they might produce a different ratio of equivalent mutants. Notice that also other factors might influence our results; for example, a different programming language is used (Fortran vs Java), the investigated programs have a different structure and a different size (small procedural programs vs large object-oriented systems consisting of multiple classes), and the types of test suites differ (simple test suites that test a small program exhaustively vs complex test suites that test huge parts of a system).

In order to quantify the differences between using the JAVALANCHE standard configuration and using the sufficient operators as proposed by Offutt *et al*., we implemented the sufficient operators in JAVALANCHE, applied them to our subject programs and compared the results.

The operators were implemented according to their description in the book *Introduction to software testing* by Ammann and Offutt [15]. As JAVALANCHE manipulates the bytecode rather than the source code, some adaption had to be made, and we implemented the operators as follows:

Absolute value insertion (ABS) Every load of a numerical variable (which corresponds to the opcodes ILOAD, LLOAD, FLOAD, DLOAD, GETSTATIC, GETFIELD) is replaced with its absolute value, the negative of the absolute value, and a call to a `failOnZero()` method with the variable as argument. The `failOnZero()` method considers a mutation to be detected when zero is passed as an argument.

Arithmetic operator replacement (AOR) In an arithmetic expression with two operants, the arithmetic operator is replaced with all possible other operators (addition, subtraction, multiplication, division, modulo); in addition, the whole expression is replaced with each of the two operators (e.g. the operator and an operand are removed). The power operator is left out because Java does not support it as a native operator. In the bytecode, this corresponds to manipulating the opcodes IADD, ISUB, IMUL, IDIV, IREM and corresponding opcodes for the other primitive types LADD, FADD, DADD and so forth.

Relational operator replacement (ROR) In a relational expression with two operands, the operator is replaced with all possible other operators (greater/less (or equal), equals, not equals), and the whole expression is replaced with `true` and `false`. This operator corresponds to manipulating the conditional jump opcodes in the bytecode, such as IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPGT, IF_ICMPLE, IF_ICMPGE and corresponding opcodes for other primitive types.

Logical operator replacement (LOR) In a logical expression with two operands, the operator is replaced by all possible other operators (bitwise and/or/exclusive or); in addition, the expression is replaced with each of the two operators. Technically, this corresponds to manipulating the opcodes IAND, IOR, IXOR, LOR, LAND and LXOR.

Unary operator insertion (UOI) Variables and constants are prepended with a unary operator (arithmetic minus, and bitwise negate). The unary plus is not inserted because it is the default and, thus, would only produce equivalent mutants. Moreover, the logical complement is left out because the boolean type does not exist in the bytecode (it is expressed via integers). In the bytecode, this operator corresponds to manipulating variable loads like ILOAD, LLOAD, FLOAD, DLOAD, GETSTATIC, GETFIELD and instructions that load constants, for example, LDC, BIPUSH, SIPUSH, ICONST_0, ICONST_1 and so forth.

The AOR, ROR, LOR and parts of the UOI operator were realized by extending existing operators, and the ABS and parts of the UOI operator were newly implemented.

8.1. Results

The results for applying the JAVALANCHE standard operators and the sufficient operators are summarized in Table XII. The table has one line for each project, and the last line summarizes the results for all projects. Each entry first shows the results for the JAVALANCHE standard operators and then for the sufficient operators. Each result consists of the total number of mutations, the mutation score and the mutation score relative to the covered mutations for the mutations.

Table XII. Comparison of JAVALANCHE standard operators and Offutt-96 sufficient operators.

	JAVALANCHE standard operators			Sufficient operators (Offutt *et al.*)		
Project name	Number of mutants	Mutation score (%)	Mutation score for covered (%)	Number of mutants (%)	Mutation score (%)	Mutation score for covered (%)
ASPECTJ	17 328	35.41	66.93	30 643	27.08	53.43
BARBECUE	17 631	5.68	59.40	19 387	15.90	61.38
COMMONS	14 716	79.16	84.73	47 683	72.58	74.86
JAXEN	9 285	48.39	70.95	18 559	43.23	58.94
JODA-TIME	21 052	51.42	82.83	54 032	51.62	76.31
JTOPAS	1 678	67.64	81.07	5 089	75.44	86.78
XSTREAM	8 240	68.54	87.05	13 236	64.83	74.77
Total	89 930	45.47	78.45	188 629	50.00	71.00

When we compare the number of mutations produced by the different operators (column 2 and 5), we see that the JAVALANCHE standard operators produce fewer mutations than the sufficient ones for all projects. This is most pronounced for the COMMONS project, where the sufficient operators produce more than three times as many mutants than the standard operators. In total, the sufficient operators produced more than twice as many mutants than the standard operators (188 629 vs 89 930).

The mutation score, which is the ratio of detected mutants among all mutants, is higher for the standard operators for four out of seven projects. In total, however, the mutation score for the sufficient operators (50%) is higher than the total score for the standard operators (45%). We also computed the mutation score relative to all covered mutants because the test suites of the projects used for this study fail to cover many mutations, and therefore, these mutations cannot be detected. The mutation score for the covered mutations is higher for the standard operators in five out of seven cases. This trend also holds when we consider the total score over all projects, where 78% of the covered mutations produced by the standard operators are detected versus 71% of the covered mutations produced by the sufficient operators.

From these results, we can conclude that the mutations produced by the standard operators are less likely to be covered by a project's test suite, but if they are covered, they are easier to detect. As the sufficient mutations are slightly harder to detect, we assume that they also produce slightly more equivalent mutants. As the differences are moderate, we believe that they have no significant impact on our results; for example, similar results would be obtained by using other operators. Furthermore, our techniques to detect non-equivalent mutants do not depend on the operators and might as well be applied on mutations produced by the sufficient operators. Regarding their effectiveness, there is no reason to believe that they would perform differently for these operators than for the standard operators. Further research will shed more light on the 'best' choice on mutation operators.

9. 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 [1, 3, 16, 17]. 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 5), 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 11).

For assessing mutations on the basis of coverage (Sections 7.1 and 7.2), 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 11).

Threats to *construct validity* concern the appropriateness of our measures for capturing our dependent variables. A threat to validity for the manual classification of mutations (Section 5) is that it was carried out by one programmer. We addressed this by writing a test case for every mutation classified as non-equivalent. This is the ultimate measure whether a mutant is non-equivalent, as it proves the non-equivalence. The time needed to classify a mutant depends on the expertise of a programmer for a project. The classification times reported in our paper refer to a programmer with a medium expertise of the projects and programmers with a deep knowledge of a project might classify mutations faster. When classifying mutations on the basis of impact (Section 7.1), we directly provide the information as required by the programmer. Finally, in Section 7.2, our assumption that the test suite measures real defects is an instance of the 'competent programmer hypothesis' also underlying mutation testing [18]. This hypothesis may be wrong; however, the maturity and widespread usage of the subject programs should suggest sufficient competence. Further studies will help in completing our knowledge on what makes a test suite adequate.

10. RELATED WORK

10.1. Mutation Testing

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 an earlier paper [4], we experimented with an alternate approach, on the basis of 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 behaviour and produces better results.

The problem of equivalent mutants was also diagnosed and tackled by other researchers. Baldwin and Sayward [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 [12] 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 (i) must be reachable, (ii) cause an incorrect state after it is executed (iii) 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 we can focus on those mutations that cannot be handled with the static approaches. Another question is how well the static approaches scale. Whereas we evaluated our impact metrics on programs of significant size, Offutt and Pan [12], for example, evaluated their technique on 11 programs with 11 to 30 executable statements.

Other approaches to the problem of equivalent mutants include aiding the programmer in detecting equivalent mutants using program slicing [17], or generating fewer equivalent mutants by using genetic algorithms [19] or higher-order mutants [20, 21].

Weak mutation, as proposed by Howden [13], considers a mutation to be detected when its containing component computes a different result for at least one test case. In the paper, however, it is not exactly specified what part of a program should be considered as a component. Thus, Woodward and Halewood [22] introduced *firm mutation* as a middle ground between weak and regular strong mutation testing. Firm mutation allows to define a starting point, where a mutation is carried out, and an end point, where the results are compared with executions of an unmutated program to decide whether the mutation is detected by the tests.

Similar to our data impact, both techniques also measure some differences of internal data during the program run. In contrast to our approach, they only monitor a part of the program while we collect the differences across the whole program. Furthermore, we are trying to predict a mutant's equivalence from the differences in internally used data instead of using it to define whether a mutant is detected or not. As we can see from the results in Section 7.1, there are cases where a mutation causes a component to produce different results, while the whole program produces a correct result. Another perspective on our approach is that it can be seen as a mix between weak and strong mutation. An impact indicates that a mutation is weakly detected (killed) by an input. If the mutation is not detected by regular strong mutation testing, the impact can provide clues on how to strongly detect it.

10.2. Statement Coverage

The traditional use of statement coverage is to measure how well tests exercise the code under test and to detect areas of the code that are not covered by the tests. In contrast to our approach, which also takes into account the execution frequency of a statement, traditional statement coverage just measures whether a statement is executed or not. Besides its traditional use, statement coverage is also used in different scenarios.

Jones and Harrold [23] presented the TARANTULA tool that uses coverage information for *bug localization*. By comparing the statement coverage of passing and failing test cases, the suspiciousness of a statement is computed. The intuition behind this approach is that statements that are *primarily executed by failing test cases* are more suspicious than statements primarily executed by passing tests. The statements can then be ranked according to their suspiciousness. The results of their evaluation show that the defective statement is ranked in the top 10% for 56% of the cases. Both our approach and TARANTULA follow the idea that changed coverage expresses anomalous behaviour. TARANTULA uses coverage differences between test cases to find a defective statement whereas our approach uses coverage difference between runs of the test suite to estimate the impact of a mutation.

Elbaum *et al.* [24] investigated how statement coverage data change when the source code is changed. The results suggest that even small changes can have a huge impact on the code coverage. However, the authors also state that the changes in code coverage are hard to predict. These findings support our decision to use statement coverage as an impact metric because mutations are also small changes. Furthermore, not all changes result in coverage changes, which indicates that the coverage changes are sensitive to semantic changes.

Gordia [25] proposed the concept of *dynamic impact analysis*. To this end, the *dynamic impact graph* is built, which is directed and acyclic. The nodes of the graph represent different executions of the program elements. Edges are between nodes that can potentially impact each other. The edges carry a probability that tells how likely an element impacts its direct successor. By traversing the graph, it can be computed how likely it is that a program element impacts an output element. The proposed applications of this approach are to estimate the risk of changes and to aid in test case selection for mutation testing. However, this approach might also be used in a similar way as our approach. For a not detected mutant, its probability of manipulating the output can be computed, which corresponds to its chance of being non-equivalent.

Test case prioritization [26] is concerned with meeting a specific performance goal faster by reordering the execution of test cases, for example, to detect faults earlier in the testing process or to reach specific coverage goals faster. To this end, coverage data are used to prioritize the test cases according to different strategies.

Software change impact analysis aims to predict the results of code changes, that is, which parts of the code are affected by a change. Orso *et al*. [27, 28] proposed an approach for software impact analysis, which is also called COVERAGEIMPACT. They use their *Gamma* approach to collect actual field data from deployed programs to compute the potential impact of a program change. To this end, traces are taken from a sample of deployed programs. A trace, in this context, consists of all methods that were called during an execution of the program. Every trace that traverses a changed method is identified, and all methods that are covered by this trace are added to a set of *covered methods*. Then, a static forward slice for every method that was changed is computed, and the methods that are covered by the slice are collected in a set of *slice methods*. The intersection of both sets is the set of methods that are potentially impacted by this change. Besides using the same name, our approach and the approach by Orso *et al*. are different. The approach of Orso *et al*. samples many executions on different machines to approximate the impact of potentially larger changes on deployed programs and typical usage scenarios, whereas our approach just aims to assess the impact of one small change (mutation) for a specified usage scenario (test cases) for one deployed version. However, using static forward slices can also help in assessing the equivalence of a mutant. A similar idea was proposed by Hierons *et al*. [17].

11. 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 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 to assess 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. Thus, if a developer focuses on the mutations with most impact, she has to deal with fewer equivalent mutants. This paper substantiates this claim in an evaluation on 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 with 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 is available at http://www.javalanche.org/.

ACKNOWLEDGEMENTS

This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1. Gordon Fraser, Yana Mileva, Jeremias Rößler, Andrzej Wasylkowski and 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. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005; 402–411, DOI: http://doi.acm.org/10.1145/1062455.1062530.
2. Walsh PJ. A measure of test case completeness. *Ph.D. Thesis*, Binghamton University, 1985.
3. Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: an experimental comparison of effectiveness. *Systems and Software* 1997; **38**:235–253.
4. Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. *ISSTA 2009: Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, USA, 2009.
5. Baldwin D, Sayward F. Heuristics for determining equivalence of program mutations. *Technical Report 276*, Yale University, Department of Computer Science, 1979.
6. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification, and Reliability* 1994; **4**:131–154.
7. Budd AT, Angluin D. Two notions of correctness and their relation to testing. *Acta Informatica* 1982; **18**:31–45.
8. Grün BJM, Schuler D, Zeller A. The impact of equivalent mutants. *Mutation '09: 4th International Workshop on Mutation Analysis*, Denver, CO, USA, 2009.
9. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. *ISSTA '93: Proceedings of the 1993 International Symposium on Software Testing and Analysis*, Cambridge, MA, USA, 1993; 139–148, DOI: http://doi.acm.org/10.1145/154183.154265.
10. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1996; **5**(2):99–118. DOI: http://doi.acm.org/10.1145/227607.227610.
11. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435. DOI: http://dx.doi.org/10.1007/s10664-005-3861-2.
12. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability* 1997; **7**(3):165–192.
13. Howden WE. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering (TSE)* 1982; **8**(4):371–379. DOI: http://dx.doi.org/10.1109/TSE.1982.235571.
14. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)* 2001; **27**(2):99–123.
15. Ammann P, Offutt J. *Introduction to Software Testing*, 2nd ed. Cambridge University Press: New York, NY, USA, 2008.
16. Offutt AJ, Pan J. Detecting equivalent mutants and the feasible path problem. *COMPASS '96: Proceedings 11th Conference on Computer Assurance*, Gaithersburg, MD, USA, 1996; 224–236.
17. Hierons R, Harman M. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 1999; **9**(4):233–262.
18. DeMillo RA, Lipton RJ, Sayward F. Hints on test data selection: help for the practicing programmer. *Computer* 1978; **11**(4):34–41. DOI: http://dx.doi.org/10.1109/C-M.1978.218136.
19. Adamopoulos K, Harman M, Hierons RM. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. *Genetic and evolutionary computation* 2004; **3103**:1338–1349.
20. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1992; **1**(1):5–20.
21. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. *SCAM 08: Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation*, Beijing, China, 2008; 249–258.
22. Woodward M, Halewood K. From weak to strong, dead or alive? An analysis of some mutation testing issues. *ISSTA '88: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, 1988; 152–158.

23. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. *ASE '05: Proceedings of the 20th international Conference on Automated Software Engineering*, Long Beach, CA, USA, 2005; 273–282.
24. Elbaum S, Gable D, Rothermel G. The impact of software evolution on code coverage information. *ICSM '01: Proceedings of the 17th International Conference on Software Maintenance*, Florence, Italy, 2001; 170–179.
25. Goradia T. Dynamic impact analysis: a cost-effective technique to enforce error-propagation. *ISSTA '93: Proceedings of the 8th International Symposium on Software Testing and Analysis*, Cambridge, MA, USA, 1993; 171–181.
26. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *Transactions on Software Engineering* 2001; **27**(10):929–948.
27. Orso A, Apiwattanapong T, Harrold MJ. Leveraging field data for impact analysis and regression testing. *ESEC/FSE '03: Proceedings of the 9th European Software Engineering Conference held jointly with 11th International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, 2003; 128–137.
28. Orso A, Apiwattanapong T, Law J, Rothermel G, Harrold MJ. An empirical comparison of dynamic impact analysis algorithms. *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, UK, 2004; 491–500.