Efficient Mutation Testing using Whole Test Suite Generation

Gordon Fraser
University of Sheffield
Sheffield, UK
Gordon.Fraser@sheffield.ac.uk

Andrea Arcuri
Certus Software V&V Center at Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

Abstract—By seeding artificial faults (mutants), mutation testing can tell us how good existing tests are, and it can help to direct test generation efforts. However, mutation based test generation is hampered because there usually are simply too many mutants, and too many of these mutants are either trivially killed or equivalent. Any effort spent on test generation for equivalent mutants is per definition wasted, and misdirects resources from where they could be put to better use – for example to produce tests revealing more non-equivalent mutants. To overcome this problem, our search-based EvoSuite test generation tool integrates two optimizations: First, we avoid redundant test executions on mutants by monitoring state infection conditions, and second we use whole test suite generation to optimize test suites towards killing the highest number of mutants, rather than selecting individual mutants. These optimizations allowed us to perform one of the largest empirical studies on mutation testing to date, where we applied EvoSuite to a random sample of 100 open source projects consisting of a total of 8,963 classes, leading to a total of 1,380,302 mutants. The experiment not only demonstrates that EvoSuite scales well, but it also quantifies the relations between weak and strong mutation testing as well as branch coverage, and points out current limitations.

Keywords—mutation testing; test case generation; search-based testing; testing classes; unit testing

I. INTRODUCTION

In mutation testing, the problem of choosing which test cases to generate is directed by mutants – small syntactic changes in program code intended to imitate real defects. A set of test cases that can distinguish between a program and its mutants is sensitive with respect to errors, and is thus thought to be a good sample of the usually infinitely large domain of test cases. Indeed, experiments have shown that mutation testing can lead to test suites that are superior to those produced using traditional structural code coverage [31] or dataflow criteria [9], [26], [32].

However, mutation testing suffers from well-known problems: An average program already results in so many mutants that test generation faces serious scalability issues. Yet not all mutants can add value to a test suite: Many mutants are trivially killed by any test case that executes them, essentially making them useless for improving test suites. Even worse, equivalent mutants are semantically equivalent to the original program, such that a test case that would distinguish between program and mutant does not exist. Unfortunately, detecting such equivalent mutants is an undecidable problem. Any resources spent on trying to generate a test case for an equivalent mutant are per definition wasted – yet it is difficult to decide when to stop trying to generate a test, as the mutant might simply be difficult and would add important value to the resulting test suite. Consequently, the question of which mutant to target next during test generation is a difficult one to answer.

Our EvoSuite test generation tool addresses the general problem of trivial and infeasible testing targets (of which equivalent mutants are an instance) by generating test suites for all testing targets at the same time, rather than considering individual targets. This way, there is no danger of misdirecting resources during test generation. To counter the scalability problem, EvoSuite uses several optimizations: Runtime checks determine if a mutant would lead to a state infection for a given test execution; this way we can effectively perform weak mutation testing without ever executing a test on a mutant, and it reduces the number of tests executed during strong mutation testing. This and further optimizations lead to a scalable approach to mutation test generation. In detail, in this paper:

- We introduce mutation testing as a search-based whole test suite generation problem (Section III).
- We describe the mutation operators implemented in EvoSuite, including the infection distance metrics previously unspecified in the literature (Section IV).
- We empirically investigate the effectiveness of the approach, and compare it to branch coverage as well as the traditional approach of targeting one mutant at a time on a large, statistically valid sample of open source software (Section V).

Our experiments show with strong statistical significance that EvoSuite is superior to the traditional approach for weak and strong mutation testing. On the open source classes the mutation scores are lower than those usually reported in the literature, indicating open problems related to environmental dependencies and testability.
II. BACKGROUND

Mutation testing was introduced in the 1970s [7], [13] as a method to evaluate how good a test suite is at detecting faults, and to give insight into how and where a test suite needs improvement. The key idea of mutation testing is to seed artificial faults based on real errors commonly made by programmers. The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. A mutant that is detected as such is considered killed, and of no further use. A live mutant, however, shows a case where the test suite fails to detect an error and therefore needs improvement. This distinguishes mutation testing from traditional coverage criteria, which merely check whether some artifact has been executed. A test suite with no oracles (e.g., assert statements on the outputs of a class under test) at all may easily achieve a coverage of 100%—but the lack of oracles would invariably be detected by mutation testing.

Scalability: An important issue in mutation testing is scalability. The number of mutants for a given system can be huge, and executing a test suite against each of the mutants is expensive. This has been addressed by a number of optimizations; see [18].

Of relevance for this paper is the distinction between strong and weak mutation testing [16]: In weak mutation testing, a mutant is considered to be killed if the program state has changed after the execution of the mutant, whereas a mutant is strongly killed only if the program output changes. There are different options of the point at which this difference needs to be observed in weak mutation [25] (e.g., after the mutated expression, statement, or basic block), and experiments [22] have shown that weak mutation testing results in test suites that are almost as good as those produced using strong mutation testing, yet the computational costs are reduced.

Just et al. [19] recently presented further optimizations: First, there is intrinsic redundancy in some mutation operators such as relational or logical operator replacement, which can be avoided by constructing not all, but only the necessary operator replacements. Furthermore, they showed that prioritizing test cases by execution time can significantly improve the performance of mutation analysis.

Equivalent Mutants: The second main issue with mutation testing is equivalent mutants—mutants that only change the program’s syntax, but not its semantics, and thus are undetectable by any test. Establishing equivalence is undecidable in general, and therefore imposes manual work. In a recent study [29], it took 15 minutes on average to assess one single mutation for equivalence. The problem of equivalent mutants has been known since the early days of mutation and has ever since been the bugbear of mutation testing – to date there is no satisfactory solution to the problem. For example, some equivalent mutants can be detected by compiler optimization techniques [5], which was reported to detect about 10% of the equivalent mutants [24]. Some equivalent mutants can be detected through constraint solving [27], and other approaches to the problem of equivalent mutants include aiding the programmer in detecting equivalent mutants using program slicing [15], generating less equivalent mutants by using genetic algorithms [1] or higher order mutants [17], [23].

Mutation Test Generation: Research on automated test case generation has resulted in a great number of different approaches, deriving test cases from models or source code, using different test objectives such as coverage criteria, and using many different underlying techniques and algorithms. In this paper we consider generating mutant-killing test cases directly from the source code.

In the context of mutation testing, DeMillo and Offutt [8] have adapted constraint based testing to derive test data that kills mutants, defining conditions that lead to an infection of the system state. Bottaci [6] proposed to use similar conditions as fitness function for search based approaches, and this has been used for experiments ant colony optimization to derive test data that kills mutants [4]. Fraser and Zeller [12] used a genetic algorithm to derive unit tests with assertions that reveal mutants, which addresses not only the problem of reaching mutants, but also propagating the state changes to observable outputs. Harman et al. [14] presented a combination of dynamic symbolic execution (DSE) to reach mutants and infect the state, and search to propagate the state changes. Papadakis and Malevris [28] presented an application of DSE and program transformations to produce test inputs for mutation testing. Zhang et al. [33] instrumented programs with additional branches that lead to state infection, and then applied DSE to derive test data.

All of these approaches to generate tests target one mutant at a time. As some mutants are trivial to kill, others are more difficult, and some are equivalent, the order in which mutants are considered can have a severe effect on the performance of a testing tool. This is not specific to mutants, but applies to any type of testing target. We have therefore recently introduced the concept of whole test suite generation [11], where rather than generating individual tests, we generate entire test suites targeting all testing targets at the same time. In our initial work we addressed branch coverage; in this paper we extend this work to mutation testing.

III. WHOLE TEST SUITE GENERATION

In this section, we describe a search-based approach to generate test suites that maximize mutation scores.

A. Search-based Testing

In search-based testing, the problem of test data generation is cast as a search problem, and search algorithms are used to derive test data. Genetic algorithms are perhaps
the most popular family of meta-heuristic search algorithms.
They mimic the evolutionary processes in nature: A population of initially randomly generated candidate solutions is evolved using genetically inspired search operators; individuals reproduce using crossover, are mutated, and selection is guided by a fitness function that heuristically measures how good a candidate solution is at solving the problem at hand. The fitness of the individuals would gradually improve from generation to generation, and the search is stopped when an optimal solution has been found, or when some other predefined stopping condition (e.g., maximum number of generations or fitness evaluations) has been met.

B. Object-Oriented Unit Test Generation

In this paper we consider unit test generation for object oriented code. In this scenario, a test case is a sequence of statements \( t = \langle s_1, s_2, \ldots, s_l \rangle \) of length \( l \). In the traditional approach of generating test cases for individual goals, the individuals of a search-based approach would be such test cases. We consider the case of whole test suite generation, where we target all testing goals at the same time. An individual of the search is thus a test suite, which is a set \( T \) of test cases \( t_i \). Given \( |T| = n \), we have \( T = \{ t_1, t_2, \ldots, t_n \} \).

To apply a genetic algorithm to a population of test suites one needs mutation and crossover operators, and a method to produce the initial population. Crossover between two test suites creates two offspring test suites, each containing subsets from both parents. Mutation of test suites leads to insertion of new test cases, or change of existing test cases. When changing a test case, we can remove, change, or insert new statements into the sequence of statements. To create a new test case, we simply apply this statement insertion on an initially empty sequence until the test case has a desired length. For details on these search operators we refer to [11].

C. Branch Coverage

A popular criterion used to guide test generation is branch coverage, which requires that every branching instruction (e.g., if, while) evaluates to true and to false. For a given test suite \( T \), the fitness value is measured by executing all tests \( t \in T \) and keeping track of the set of executed methods \( F_T \). Once all mutants are reached, then every mutant needs to be reached, therefore the fitness function for weak mutation operators have been presented to date. We will define impact distances for our mutation operators in Section IV. For each mutant \( M_n \in M \), where \( M \) is the entire set of mutants, there is a distance function \( d_i(M_n) \geq 0 \) that estimates the distance towards a state infection; if \( d_i(M_n) = 0 \) then the state is infected. For a test suite \( T \) and a mutant \( M_n \), we thus have:

\[
d_i(M_n, T) = \begin{cases} 
1 & \text{if } M_n \text{ was not reached,} \\
\nu(d_{min}(M_n, T)) & \text{if } M_n \text{ was reached.}
\end{cases}
\]

Using whole test suite generation, the aim of weak mutation testing is to achieve state infection on all mutants. A prerequisite for this is that all mutants are reached. By definition, if we achieve branch coverage then necessarily all mutants are reached, therefore the fitness function for weak mutation testing is based on the branch coverage fitness function.

D. Weak Mutation Testing

A mutant is weakly killed if it leads to a state change, i.e., if it infects the state. For each mutation operator it is possible to define a constraint that precisely describes the conditions that lead to a state infection; this has been used for test generation [8], [14], [28], [33]. Rather than such constraints, in search-based testing we require a distance estimation that guides the search towards infection. The idea of an infection distance has been initially proposed by Bottacci [6] and even though it has been used in the literature (e.g., [12]) to the best of our knowledge no actual infection distance functions for individual mutation operators have been defined so far. For each mutant \( M_n \in M \), there is a distance function \( d_i(M_n) \geq 0 \) that estimates the distance towards a state infection; if \( d_i(M_n) = 0 \) then the state is infected. For a test suite \( T \) and a mutant \( M_n \), we thus have:

\[
d_i(M_n, T) = \nu(d_{min}(M_n, T))
\]

The fitness function for branch coverage estimates how close a test suite is to covering all branches of a program, therefore it is important to consider that each predicate has to be executed at least twice so that each branch can be taken. Consequently, we define the branch distance \( d(b, T) \) for branch \( b \) on test suite \( T \) as follows:

\[
d(b, T) = \begin{cases} 
0 & \text{if the branch has been covered,} \\
\nu(d_{min}(b, T)) & \text{if the predicate has been executed at least twice,} \\
1 & \text{otherwise.}
\end{cases}
\]

Here, \( \nu(x) \) is a normalizing function in \([0,1]\) (e.g., \( \nu(x) = x/(x+1) \)). The resulting (minimizing) fitness function is:

\[
f_B(T) = |F| - |F_T| + \sum_{b \in B} d(b, T)
\]
### E. Strong Mutation Testing

Weak mutation testing results in test cases where the state is infected; however, a state infection does not necessarily propagate to an observable output. In particular, in the case of unit tests propagation might require additional method calls, and furthermore propagation needs to be observed using test assertions on the public API of the CUT.

With μTest [12] we proposed to use an impact measurement as an estimator of propagation, which was later also used by Harman and Jia [14]: The more control flow and data differences can be observed between the test run on the original program and on the mutant, the closer the state infection is assumed to be to propagation. Let impact(M, t) be the impact measurement of mutant M on test case t. We thus define the propagation distance as follows:

\[
d_p(M, T) = \begin{cases} 
0 & \text{if } M \text{ can be asserted,} \\
1 & \text{if } d_i(M, T) = 0, \\
\frac{1}{1 + \text{impact}_{\text{inf}}(M, T)} & \text{else}
\end{cases}
\]

In our scenario, a mutant is strongly killed if there exists an assertion for the test case such that it evaluates to false if the test is executed on the mutant, and to true if it is executed on the original class. The details of this assertion generation are described in Section IV-C. This results in the following fitness function for strong mutation testing:

\[
f_{SM}(T) = f_B(T) + \sum_{M_k \in M} (d_i(M_k, T) + d_p(M_k, T))
\]

In contrast to weak mutation, the propagation distance requires execution of test cases also on mutants. Test cases only need to be executed if their infection distance is 0, but still a significant increase in the costs of the fitness function is expected.

### IV. EvoSuite: Efficient Mutation Test Generation

#### A. Mutation Operators

EvoSuite works on Java byte-code level and collects all necessary information by instrumenting byte-code and analysing at runtime using Java Reflection. This means that it does not require the source code of the CUT, and in principle is also applicable to other languages that compile to Java byte-code (such as Scala or Groovy, for example). The implemented set of operators is based on the sufficient set used previously [2], [29], but adds variable replacement, unary operator insertion, and implements the “delete statement” operator in two versions as “Delete Call” and “Delete Field”, whereas the first replaces a method call with a default value of the return type, and the latter does this for a field access. The “negate condition” operator is omitted, as all conditions in byte-code are atomic, such that this operator is subsumed by relational operator replacement.

For each mutated location, the instrumentation first adds instructions to calculate the infection distance, and then calls a monitor that records mutant execution and infection. This means that executing a test case once produces all the data that is later used by the fitness function for branch coverage and weak mutation testing. For strong mutation testing, there is additional instrumentation that tracks return values. EvoSuite creates one meta-mutant containing all mutants (see e.g. [30]), thus for each mutant there is also instrumentation checking whether it should be activated.

Although the idea of an infection distance to guide the search towards infection has been proposed in the literature (e.g., [6]), we are not aware of any work in the literature where this distance is actually defined. We therefore define the distance functions for our mutation operators. Some of the operators always immediately infect the state, but in other cases the distance is a binary choice (state is infected / state is not infected) offering little guidance to the search; here, the distance functions are basically the same as the previously used constraints, and could in theory be used in a hybrid test generation approach using search and constraint solving [14]. As EvoSuite works on byte-code level, we define infection such that the state of the local frame (i.e., the values on the stack) after execution of the mutated byte-code instruction is different. This may not match the granularity of expressions or statements on the source code level. However, in principle other definitions of infection could be used, for example by adding instrumentation that checks the state in terms of the local variables. In the following, we describe the implemented mutation operators together with how the infection distance is measured:

1) Delete Call: Removes a method invocation. If the removed method has a return value, then a default-value is put on the stack. For numerical types, this is the equivalent to 0, for references this is the special value null. If the method call changes the state (i.e., it is an impure method), then by definition the infection distance is 0. For pure methods we need to compare the return value to the default value: If they are equal, the infection distance is 1, else 0.

2) Delete Field: This operator removes a field access and replaces it with a default value (0 / null). The infection distance is 1 if this equals the field value, else 0.

3) Insert Unary Operator: This operator adds 1 to, subtracts 1 from, or negates a numerical value after it was loaded on the stack. By definition, resulting mutants will always infect the state, thus the infection distance is 0.

4) Replace Arithmetic Operator: This mutation operator replaces an arithmetic operator σ in an expression a σ b with all other applicable operators. The infection distance between a σ b and a σ′ b is 1 if the results of the two are equal, else it is 0. If the original operation resulted in a division by zero error, then the distance is 0 if the mutated operator does not result in a division by zero error.
5) **Replace Bitwise Operator**: Just like for arithmetic operators, the distance is 1 if the old and the new operator result in the same value, and it is 0 if they differ. There is a special case when replacing a signed shift-right operator $x > > y$ with an unsigned shift-right operator $x > > > y$. If $x \geq 0$ and $y \neq 0$ this change has no effect, as $x$ has to be negative to make a difference. In this case, the infection distance is thus $x + 1$.

6) **Replace Comparison Operator**: This mutation operator replaces one relational operator with others using the rules defined by Just et al. [19]. This operator offers the most opportunities to guide the search, but the guidance depends on the actual operator combinations. Table I describes the infection distance calculations for this mutation operator.

7) **Replace Constant**: This operator replaces constants with the special values $-1$, $0$, $+1$, and with $c + 1$ and $c - 1$. As it only replaces a constant $c$ with a constant of different value, the infection distance is always 0.

8) **Replace Variable**: This operator replaces variables with all other variables of the same type in scope (local variables and field variables). Here, the infection distance is 0 if the values of the old and new variable differ at this point, else it is 1.

**B. Scalability**

To calculate the fitness, in weak mutation testing (see Section III-D) we only need to execute each test once. For strong mutation testing, however, tests also need to be executed on mutants to measure the impact. When code results in many mutants, calculating the fitness thus becomes expensive, as a naive approach would need to run every test case on every mutant. A first optimization is to execute a test on a mutant only if that mutant has been reached and has infected the state. In our setting this means that when evaluating the fitness of a test suite, for each mutant we first determine which test cases have reached the mutant and then calculate the minimal infection distance. Only test cases that achieve infection distance 0 need to be executed for a mutant. Based on the results of Just et al. [19], we prioritize the test cases with infection distance 0 based on their execution time, and then execute each test successively to determine the minimal propagation distance. If any test achieves propagation distance 0 that means the mutant is strongly killed, and we can stop executing tests on it.

As a further optimization for strong mutation testing, whenever a mutant has been strongly killed, we keep a copy of the killing test in a pool, and from this moment on exclude this mutant from the fitness calculation; i.e., $d_i(M_k, T)$ and $d_p(M_k, T)$ are 0 for such a mutant. This way, the time wasted on trivial and already covered mutants is reduced, and fitness calculations become cheaper the more mutants are killed. In future work, we will consider more sophisticated techniques to sample subsets of mutants with the goal of reducing the costs of fitness calculations even further.

**C. Assertion Generation**

The test case representation discussed so far only encompasses sequences of method calls, but does not entail test assertions, which serve as test oracles in unit testing and which are required to strongly kill a mutant (Section III-E). Each value produced in a sequence of method calls can be checked with different assertions: For example, primitive values are checked using `assertEquals(expectedValue, observedValue)`, and complex objects are checked in terms of their primitive fields or observer methods; values can be compared with each other, etc. For a given test execution, `EvoSUITE` collects all possible assertions. By comparing the assertions resulting from execution on the mutant and on the original class we can determine if the mutant is killed, i.e., if there exists an assertion that is true on the original class but false on the mutant.

As the number of assertions can be quite large, mutation testing also serves to filter down these assertions to those that are required to distinguish between the original program and its mutants. When `EvoSUITE` creates JUnit test cases with assertions, it only includes assertions that are needed in order to kill mutants. Note that this process of assertion selection is a post-processing step, and it is independent of the coverage criterion that is used to drive test generation. For example, `EvoSUITE` also uses mutation analysis to select assertions for test cases generated for branch coverage. This process is described in detail in [12].

**V. Evaluation**

To evaluate the effects of the described approach and optimizations, we ran a large scale experiment using `EvoSUITE`. In detail, we aim to answer the following research questions:

**RQ1**: Does the whole test suite approach improve performance in weak mutation testing?

**RQ2**: Does the whole test suite approach improve performance in strong mutation testing?

**RQ3**: How does the achieved branch coverage affect the mutation score?

**RQ4**: How do the mutation scores of weak and strong mutating testing compare to those of test suites generated targeting only branch coverage?

**RQ5**: What is the correlation between branch coverage and mutation score?

**A. Case Study**

To answer the research questions addressed in this paper, we use the SF100 corpus as case study [10]. The SF100 corpus is composed of 100 open source projects randomly selected from SourceForge, which is perhaps the largest web repository for open source software. The SF100 corpus contains a total of 8,963 classes, having more than 290 thousand byte-code level branches. Because the SF100 corpus is a large and unbiased selection of open source projects, this gives us confidence that our answers to the
addressed research questions can generalize to other open source software as well. More details on the SF100 corpus can be found in [10] and at our website www.evosuite.org.

B. Experiment Setup

We applied EvoSuite on each of the 8,963 classes in the SF100 corpus. EvoSuite can be used both in the whole test suite generation approach described in this paper as well as a traditional approach targeting one goal at a time. In the latter case, the same representation and mutation operators are used, but crossover on test cases is different (see [11]). The fitness function used for strong mutation testing is the one from [12]; for weak mutation testing it is a variant without the propagation distance. Mutants to target are randomly selected; the best individuals for previous mutants are used to seed the initial population for successive mutants.

We considered five different configurations: default in which only branch coverage is sought, weak mutation testing with and without whole test suite approach, and the same for strong mutation testing. For each class/configuration, experiments were repeated 10 times with different random seeds to take into account the stochastic nature of EvoSuite [3]. In total, we had $8,963 \times 5 \times 10 = 448,150$ runs of EvoSuite.

For each run, the search was stopped after a two minute timeout. The more a search is left running, the better results one would expect. In general, due to the presence of infeasible targets (e.g., equivalent mutants), one cannot know when the search has covered all the feasible ones.

After the test suites are generated, there are two post-processing steps. First, we use a minimization algorithm to remove all statements in the test suite that do not contribute to branch coverage and mutation score. Second, we add mutant-killing assertions (see Section IV-C). For each of these steps, we used a 10 minute timeout. Note: in contrast to the search, both post-processing finish after a finite amount of time, which can be lower than the chosen timeout. We chose a high timeout just for sake of experimentation, just to have high confidence that those post-processing steps were always (or at least most of the time) completed. In other words, the choice of how to best distribute the testing budget among the different steps (i.e., test case generation, minimization and generation of assertion) is an open research question that is not in the scope of the paper.

Given the above settings, the computational effort of the empirical study lies between 622 and 6,846 days, requiring the use of a cluster of computer to run all experiments. On average, each class resulted in 154 mutants (median 58), with some classes having up to 27,828 mutants, leading to a total of 1,380,302 mutants. To the best of our knowledge, this is the largest study on mutation testing to date.

All the data resulting from this empirical study were analyzed using statistical methods following the guidelines in [3]. In particular, we used the Vargha-Delaney $A_{12}$ effect size and Wilcoxon-Mann-Whitney U-test.

Given a performance measure $K$ (e.g., branch coverage or mutation score), $A_{xy}$ measures the probability that running algorithm $x$ yields higher $K$ values than running algorithm $y$. If the two algorithms are equivalent, then $A_{xy} = 0.5$. This effect size is independent of the raw values of $K$, and it becomes a necessity when analyzing the data of large case studies involving artifacts with different difficulty and different orders of magnitude for $K$. E.g., $A_{xy} = 0.7$ entails one would obtain better results 70% of the time with $x$.

The Wilcoxon-Mann-Whitney U-test is used when algorithms (e.g., result data sets $X$ and $Y$) are compared on single classes (in $R$ this is done with wilcox.test(X,Y)). We also used this test to check on the entire case study if effect sizes are symmetric around 0.5. On some classes, an algorithm can be better than another one (i.e., $A_{12} > 0.5$), but on other classes it can be worse (i.e., $A_{12} < 0.5$). A test for symmetry (in $R$ this is done with wilcox.test(Z,mu = 0.5), where for example $Z$ contains 8,963 effect sizes, one per class in SF100) determines if there are as many classes in which we get better results as there are classes in which we get worse results. Note that this test makes sense if and only if the case study is a valid statistical sample (as it is the case for the SF100 corpus). On hand-picked case studies, the bias in their selection (e.g., proportion of different application types) would make this type of analysis pointless.

Note: although the case study is composed of 8,963 classes, data were obtained only for 8,771. For 192 classes, EvoSuite either quits or crashes without generating any output. These problems are currently under investigation (for example, in Java there is a hard limit of 64KB of bytecode per method which may be exceeded by the mutation instrumentation in some cases, and some classes cannot be initialized because of the security manager restrictions), but it does not bear any major threat to the empirical study, as only 2% of the case study is involved.

<table>
<thead>
<tr>
<th>Operator $\sigma$</th>
<th>$\neq$</th>
<th>$&lt;$</th>
<th>$\leq$</th>
<th>$&gt;$</th>
<th>$\geq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=$</td>
<td>-</td>
<td>$\delta &gt; 0?\delta : 0$</td>
<td>$\delta \geq 0?\delta + 1 : 0$</td>
<td>$\delta &lt; 0?\delta : 0$</td>
<td>$\delta \leq 0?\delta + 1 : 0$</td>
</tr>
<tr>
<td>$\neq$</td>
<td>-</td>
<td>-</td>
<td>$\delta \leq 0?\delta + 1 : 0$</td>
<td>$\delta &lt; 0?\delta : 0$</td>
<td>$\delta \geq 0?\delta + 1 : 0$</td>
</tr>
<tr>
<td>$\leq$</td>
<td>$\delta &gt; 0?\delta : 0$</td>
<td>$\delta \leq 0?\delta + 1 : 0$</td>
<td>-</td>
<td>$</td>
<td>\delta</td>
</tr>
<tr>
<td>$&gt;$</td>
<td>$\delta &lt; 0?\delta : 0$</td>
<td>$\delta &lt; 0?\delta + 1 : 0$</td>
<td>$</td>
<td>\delta</td>
<td>$</td>
</tr>
<tr>
<td>$\geq$</td>
<td>$\delta \leq 0?\delta + 1 : 0$</td>
<td>$\delta &gt; 0?\delta : 0$</td>
<td>$0$</td>
<td>$</td>
<td>\delta</td>
</tr>
</tbody>
</table>

Table I

### Infection Distances for Replacements of Comparison Operators $\sigma$. Given the Comparison $x \sigma y$, then we define $\delta = x - y$.
analyzes and compares RQ1 and RQ2, Table II analyzes and compares the mutation scores of the whole test suite approach with the traditional approach of targeting one mutant at a time. The results in Table II show with high statistical confidence that, for both weak and strong mutation testing, the whole test suite approach significantly improves performance.

**RQ1, RQ2**: Whole test suite generation improves mutation testing significantly, with average effect size $\hat{A}_{12}$ up to 71%.

The variety of classes in SF100 ranges from simple classes with just getters and setters on their internal fields to very complex software. Furthermore, from previous experiments [10] we know that there are several classes for which EVO SUITE cannot achieve high coverage, regardless of how long we leave it running. For example, some classes require the generation of files and GUI events, which EVO SUITE does not support yet. To better understand how the difficulty of a class affects the approach, RQ3 studies and compares the mutation score based on the achieved branch coverage of the whole test suite approach.

Figure 1 shows a boxplot comparing the whole test suite approach for weak mutation score and the traditional approach of targeting each mutant separately, and Figure 2 shows the same boxplot for strong mutation. In other words, the 8,963 classes were divided in 10 groups, based on the achieved branch coverage. As branch coverage is different between weak and strong mutation testing, the partitions in Figure 1 and Figure 2 are not necessarily the same.

As we can see in Figure 1 and Figure 2, there is not much difference in mutation score on the classes for which EVO SUITE achieves coverage either below 10% or above 90%. This is as expected: If the coverage is this low for a class, then it represents some problem that EVO SUITE cannot overcome; if only few parts of a class are executed, there is no chance to increase the mutation score. The majority of classes for which we achieve very high coverage can be simply trivial, and so it would not be possible to generate challenging enough mutants that would be able to distinguish among different EVO SUITE configurations.

For both weak and strong mutation testing, the biggest difference in mutation score can be seen in the range between 20% and 40% of branch coverage, where we expect classes that are difficult to test, but not untestable like those in the 0-10% range. The differences in mutation score achieved by the whole test suite approach for strong mutation testing are more marked than in the case of weak mutation score. For example, the median values for the $\hat{A}_{12}$ effect sizes for weak mutation testing are around 0.5 between the 40% and 80% coverage range. In the same range, for strong mutation testing the median effect sizes are above 0.8.

In both Figure 1 and Figure 2 we can see an interesting trend: Starting from low coverage, the difference in mutation score increases, but then it decreases when it reaches around 50% coverage. Then it increases again, followed by a decrease for coverage above 90%. Currently, we are not able to explain the reasons behind this peculiar behavior.

**RQ3**: There is little increase in mutation score for classes with very high or low branch coverage, and surprisingly also around 50% branch coverage.

D. Branch Coverage vs. Mutation Score

Code coverage metrics are widely used in industry, whereas mutation testing has received, so far, less attention. It is hence important to provide more insight on their relations. To see how mutation testing compares to branch coverage testing (RQ4), Table III summarizes the results of the five configurations on the entire SF100 corpus. In that table, we report the obtained branch coverage, mutation score and size. Size is defined as the total number of statements in the generated test suite after the minimization post-processing, but excluding assert statements. On more fine grained details, statistical analyses on mutation score and size are presented in Table IV.

The results in Table III and Table IV might look surprising at a first look. Although as expected the mutation score is better for both weak and strong mutation testing (average
effect sizes $\hat{A}_{12}$ are equal to 0.60 and 0.72, respectively), the obtained branch coverage is lower! Recall that, when EvoSuite generates test suites using weak and strong mutation testing, still it uses branch coverage for the fitness function. In other words, mutation testing is considered as a secondary objective to further optimize.

The lower achieved branch coverage (but with higher mutation score) can be explained as follows. To kill all mutants in a code block, that code block might need to be executed several times with different inputs. Even if a code block is easy to reach, still a fitness function based only on branch coverage would get that code executed only once. On the other hand, mutation testing might generate (and retain in the test suites) several test data for that block, which ultimately might lead to higher mutation score. However, more test data means that the test suites become bigger (as can be seen in Table III, test suites for strong mutation testing are on average 60% bigger) and so taking more time to execute. Furthermore, mutation testing introduces a computational overhead through the additional instrumentation (e.g., to calculate the infection distances). These two effects would reduce the number of fitness evaluations within the time limit for the search, and so reducing the time the search spends in trying to generate test data for complex branches (and so achieving lower code coverage).

Given enough search budget, there should not be any major difference in code coverage between mutation testing and the default settings of EvoSuite. Theoretically, given infinite time, the two coverage values should converge, with mutation testing generating larger test suites, as a further objective beside coverage (i.e., mutation score) is sought. In this paper, we used a two minute timeout for the search. Based on the results of the experiments, this two minute timeout was low enough such that the overhead of mutation testing affected the achieved branch coverage. The choice of a two minute timeout was rather arbitrary. We chose a value that, on one hand, could represent a realistic usage of EvoSuite in practice and, on the other hand, letting the experiments finish in reasonable time.

For low search budgets (e.g., in the order of minutes) and non-trivial classes, it might make sense to run EvoSuite only using branch coverage as objective to optimize. For higher test budgets, it could make sense to use more demanding criteria such as weak and strong mutation testing. Note that what is done during the search is not visible to the final user. Once the search is finished, the final user will only see a generated test suite written as a JUnit file. Therefore, the choice of targeting mutation testing during the search could be done automatically based on the defined test budget. How to best make such choice automatically will be a matter of future investigations.

**RQ4: Mutation testing leads to higher mutation scores but may lead to lower code coverage, if the search budget is limited.**

One interesting thing to note about Table III is that the mutation scores are in the range of 21%–34%, whereas previous reports (e.g., summarized in [14]) list average mutation scores in the range of 60%–90%. The main reason for this difference lies in the use of SF100 for experimentation, which is an unbiased sample of open source software which contains a range of problems that are not yet solved for test generation, e.g., environmental dependencies [10]; this
is reflected by the also low branch coverage values.

In Table III it is also noteworthy that strong mutation testing leads to a significantly higher mutation score than weak mutation testing, which is in contrast to previous empirical studies (e.g., [22]), where weak mutation testing led to almost as high mutation scores as strong mutation testing. We suspect that the main reason for this is that propagation in the case of object-oriented classes may require additional statements in the test case for which only the strong mutation testing fitness function offers guidance. We will investigate this in more detail in future work.

Finally, to investigate the correlation between branch coverage and mutation score (RQ5) we calculated the average branch coverage and mutation score achieved by EVO SUITE for each class when using strong mutation testing and the whole test suite approach. Figure 3 displays this data as a scatter plot. There are three notable clusters of classes around the corners of this plot: There are classes with very low branch coverage and mutation scores; these are classes which represent problems EVO SUITE cannot handle. Given the low search budget we used during the experiments, classes with branch coverage and mutation score around 100% are likely trivial classes that pose no problems. There is also a cluster of classes with close to 0% branch coverage but 100% mutation score; these are classes that EVO SUITE has problems to handle although the code itself is so trivial that it results in hardly any mutants (a class with no mutant would have a 100% mutation score). The problems in these classes are largely related to environmental dependencies [10]. For classes outside these three extreme cases, we see that usually branch coverage is higher than the mutation score, which shows that achieving state infection and to propagating the infection to an assertion are difficult tasks. In particular, there are cases in which there is high branch coverage but low mutation score (classes close to the x-axis of Figure 3). In these cases it is likely that the API of the CUT simply is not rich enough (i.e., not enough observer methods) to reveal sufficient internal details that would allow to produce assertions, or EVO SUITE’s choice of assertions is too limited (e.g., only observers that return primitive values are considered currently).

To quantify what is visible in Figure 3, we studied the correlation between the mutation score and branch coverage. There is a 0.60 positive linear correlation, with strong statistical validity (p-value very close to 0, where the null hypothesis is correlation value equal to 0), and with a 95% confidence interval equal to [0.59,0.62].

**RQ5: Higher branch coverage usually means higher mutation score, but this depends on infection, propagation, and the richness of the API and assertions.**

### VI. Threats to Validity

Threats to internal validity might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 10 times, and we followed rigorous statistical procedures to evaluate their results.

To cope with possible threats to external validity, the SF100 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge [10]. In contrast to hand-picked case studies, the use of the SF100 corpus provides high confidence in the possibility to generalize our results to other open source software as well.

In this paper, we only used EVO SUITE, and we did not compare with other tools. The reason is that we are aware of no other tool that can be automatically and safely applied to SF100. The SF100 corpus is freely available, and it will allow tool comparisons in the future when other Java tools are mature enough to handle SF100.

### VII. Conclusions

Mutation testing can be used to evaluate how good a test suite is at revealing possible faults. It can also be used during automated test case generation to create test suites that are better than those optimized only for code coverage. Unfortunately, mutation testing is hampered by scalability issues and the equivalent mutant problem.

To improve mutation testing, this paper extends and evaluates the whole test suite generation approach [11] for mutation testing. In previous work, the whole test suite approach led to large improvements in performance for branch coverage. One simple reason to explain such large improvements is that, with the whole test suite approach, the presence of infeasible testing targets does not harm the search. This paper confirms that being the case also for mutation testing, by performing an empirical study based on 100 Java projects randomly selected from SourceForge,
i.e. the SF100 corpus. Besides the whole test suite approach, EVO$\text{SUITE}$ also includes several optimizations for mutation testing, such as the use of infection conditions, optimised mutation operators, and prioritized test execution.

The use of a large and unbiased case study such as the SF100 corpus entailed us to also investigate general relations between coverage criteria such as byte-code branch coverage and mutation testing. For example, our experiments show that, when automatically generating test suites targeting different coverage criteria, the available testing budget (e.g., how long can we run the search?) has a major impact on performance. It is possible that, for a low testing budget such as two minutes, mutation testing can lead to generate bigger test suites that achieve higher mutation score, but, at the same time, have lower code coverage.

To learn more about EVO$\text{SUITE}$ and the SF100 corpus, visit our Web site:

http://www.evosuite.org

Acknowledgments. This project has been funded a Google Focused Research Award on “Test Amplification”. Andrea Arcuri is funded by the Norwegian Research Council.

REFERENCES


