Noname manuscript No.
(will be inserted by the editor)

Achieving Scalable Mutation-based Generation of Whole Test Suites

Gordon Fraser and Andrea Arcuri

Received: date / Accepted: date

Abstract Without complete formal specification, automatically generated software tests need to be manually checked in order to detect faults. This makes it desirable to produce the strongest possible test set while keeping the number of tests as small as possible. As commonly applied coverage criteria like branch coverage are potentially weak, mutation testing has been proposed as a stronger criterion. However, mutation based test generation is hampered because usually there are simply too many mutants, and too many of these are either trivially killed or equivalent. On such mutants, any effort spent on test generation would per definition be wasted.

To overcome this problem, our search-based EVOSUITE test generation tool integrates two novel 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 apply EVOSUITE to a random sample of 100 open source projects, consisting of a total of 8,963 classes and more than two million lines of code, leading to a total of 1,380,302 mutants. The experiment demonstrates that our approach scales well, making mutation testing a viable test criterion for automated test case generation tools, and allowing us to analyze the relationship of branch coverage and mutation testing in detail.

Keyword: mutation testing; test case generation; search-based testing; testing classes; unit testing

G. Fraser
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
E-mail: Gordon.Fraser@sheffield.ac.uk

A. Arcuri
Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
E-mail: arcuri@simula.no

1 Introduction

Automated test data generation has proven most useful so far when automated oracles are available (e.g., segmentation faults in C programs, runtime exceptions like null pointers in Java, violations of formal specifications). However, not all types of faults may be covered by automated oracles. The absence of an *automated* oracle implies the existence of a *manual* test oracle. This makes it important that the generated test suites are not too large, as checking tests manually consumes time, and it is not reasonable to assume that developers will evaluate thousands of automatically generated tests. Even in a regression testing scenario, where the current program version serves as automated test oracle, the size is important to keep test execution costs small. Therefore, a common approach is to choose a test criterion (e.g., branch coverage), and to produce a *minimized* test set that satisfies this criterion. There is a trade-off between size and the ability to detect faults. Even though branch coverage is an established default criterion in the literature, it may produce weak test sets, and software engineering research has considered many other, stronger criteria. Among these, *mutation testing* has received a significant amount of attention in the literature (Jia and Harman, 2011).

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 (Walsh, 1985) or dataflow criteria (Frankl et al, 1997; Offutt et al, 2004; Wong et al, 1995).

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.

In this paper, we evaluate the use of mutation testing in an automated test case generation tool to obtain test suites of better quality than just aiming at branch coverage. Motivated by previous encouraging results in mutation testing (Fraser and Zeller, 2012), we implemented a naive extension of EvoSuite for mutation testing, and quickly learned that it simply does not scale anywhere close to the extent that EvoSuite's default criterion, branch coverage, scales — thus rendering it unusable in practice. We thus designed and investigated several novel techniques to make mutation testing more scalable.

Our EVOSUITE test generation tool (Fraser and Arcuri, 2011a, 2013b) 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 further counter the scalability problem in mutation testing, 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, together with suitable fitness functions.
- We use novel bytecode-level instrumentation to monitor state infection for mutants during execution on the original program, thus greatly reducing the number of tests that need to be executed during mutation testing.
- We describe the mutation operators implemented in EVOSUITE, including the infection distance metrics previously unspecified in the literature.
- 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.

To obtain results of practical value, our experiments are based on $8{,}963$ classes, coming from 100 projects randomly selected from SourceForge, and consisting of more than two million lines of code. Such large, sound experiments allow us to answer our research questions with strong statistical confidence and high chance of generalization to other systems. On one hand, our experiments clearly point out that indeed traditional mutation testing does not scale to real-world software. On the other hand, the novel techniques we implemented in EVOSUITE not only led to better results than the traditional approach for weak and strong mutation testing, but they are also useful as they make mutation testing a viable option for practitioners using tools like EVOSUITE.

A somewhat surprising result is that the mutation scores observed on these open source classes are lower than those usually reported in the literature on mutation testing. We investigate this phenomenon in detail, and reveal that an important cause are open problems in testing object-oriented software related to environmental dependencies and testability. To further support those conclusions, we also carried out an empirical study on 40 Java classes used in a recent mutation testing investigation. The high mutation scores we obtained on these classes confirm our findings on the difference in testability of real-world software and examples popular in the literature.

This paper is organized as follows: Section 2 discusses test generation and mutation testing. Section 3 introduces the whole test suite approach to mutation test generation, whereas Section 4 introduces the EVOSUITE tool that implements this approach for the Java programming language. Section 5 describes our experiments and findings. Threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Background

Mutation testing was introduced in the 1970s (DeMillo et al, 1978; Hamlet, 1977) 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.

Several empirical studies have shown the usefulness of mutation testing: Walsh (1985) determined empirically that mutation testing is superior to statement and branch coverage, and Frankl et al (1997) and Offutt et al (2004) showed that mutation testing is also better at finding faults than data flow testing.

2.1 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, which are usually categorized into the three strategies (Offutt and Untch, 2001) "do fewer" (e.g., mutant sampling (Acree, 1980; Budd, 1980), selective mutation (Offutt et al, 1993)), "do smarter" (e.g., parallelization (Fleyshgakker and Weiss, 1994) or weak mutation (Howden, 1982)), and "do faster" (e.g., mutant schema (Untch, 1992), mutation of bytecode instead of source code (Offutt et al, 2004)); see (Jia and Harman, 2011) for an overview.

Of relevance for this paper is the distinction between *strong* and *weak* mutation testing (Howden, 1982): 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 (Offutt and Lee, 1991) (e.g., after the mutated expression, statement, or basic block), and experiments (Offutt and Lee, 1994) 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 (2012) recently presented further optimizations: 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 prioritising test cases by execution time can improve the performance of mutation analysis. Finally, it was suggested that state in-

fection conditions can lead to a reduction of the costs of mutation analysis (Just et al, 2013).

2.2 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 (Schuler and Zeller, 2010), 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 (Baldwin and Sayward, 1979), which was reported to detect about 10% of the equivalent mutants (Offutt and Craft, 1994). Some equivalent mutants can be detected through constraint solving (Offutt and Pan, 1997), and other approaches to the problem of equivalent mutants include aiding the programmer in detecting equivalent mutants using program slicing (Hierons et al, 1999), generating less equivalent mutants by using genetic algorithms (Adamopoulos et al, 2004) or higher order mutants (Jia and Harman, 2009; Offutt, 1992).

2.3 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 this context, the recently most successful approaches can be categorized as random techniques (e.g., Randoop (Pacheco and Ernst, 2007)), symbolic techniques using constraint solvers (e.g., DART (Godefroid et al, 2005)), or search-based techniques (e.g., (McMinn, 2004)).

Whether systematic, random, or both, all these test case generation techniques focus on reaching specific points in the code in order to satisfy some coverage criterion, but in fact a test case is more than just the execution: we also need an *oracle* that assesses the result of the execution. Besides the cases of system crashes (e.g., segmentation faults, thrown exceptions) and when formal specifications (e.g., class invariants, post-conditions) are available, there is the need of generating and selecting appropriate assertions that capture the current behavior of the software (Staats et al, 2011). But, in general, research on test generation focuses on generating test inputs, not test oracles.

In the context of mutation testing, DeMillo and Offutt (1991) have adapted constraint based testing to derive test data that kills mutants, defining conditions that lead to an infection of the system state. Bottaci (2001) proposed to use similar conditions as fitness function for search based approaches, and this has been used for experiments with ant colony optimization to derive test data that kills mutants (Ayari et al,

2007). Fraser and Zeller (2012) 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 (2011) 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 (2010) used program transformations to produce test inputs for mutation testing, and Zhang et al (2010) similary 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* (Fraser and Arcuri, 2013c), 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.

Besides automated test generation, mutation analysis has also been used to optimize existing test generation approaches. Patrick et al (2013) use mutation analysis to optimize subdomains from which a random test generator samples its values. Baudry et al (2005) use a search-based approach to optimize the values contained in unit and system tests towards increasing a test suite's overall mutation score.

3 Whole Test Suite Generation

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

3.1 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.

3.2 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 has a desired length. For details on these search operators we refer to (Fraser and Arcuri, 2013c).

3.3 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 out of the set of all methods F as well as the minimal *branch distance* $d_{min}(b,T)$ for each branch $b \in B$, where B is the set of branches for a given class under test (CUT). The branch distance is a common heuristic to guide the search for input data to solve the constraints in the logical predicates of the branches (Korel, 1990; McMinn, 2004). The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see (Korel, 1990; McMinn, 2004) for details). For example, for predicate $x \geq 10$ and x having the value 5, the branch distance to the true branch is $10 - 5 + k$, with $k > 0$. In practice, to determine the branch distance each predicate of the CUT is instrumented to keep track of the distances for each execution.

The fitness function for branch coverage estimates how close a test suite is to covering *all* branches of a program. It is important to consider that each predicate has to be executed at least twice to avoid that the search oscillates between the two values of the branch (Fraser and Arcuri, 2013c). 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., $\nu(x) = x/(x+1)$ (Arcuri, 2013)). The resulting (minimizing) fitness function is:

$$f_B(T) = |F| - |F_T| + \sum_{b_k \in B} d(b_k,T)$$

3.4 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 (DeMillo and Offutt, 1991; Harman et al, 2011; Papadakis and Malevris, 2010; Zhang et al, 2010). 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 Bottaci (2001) and even though it has been used in the literature (e.g., (Fraser and Zeller, 2012)) to the best of our knowledge no actual infection distance functions for individual mutation operators have been presented to date. We will define infection distances for our mutation operators in Section 4. For each mutant $M \in \mathcal{M}$, where $\mathcal{M}$ is the entire set of mutants, there is a distance function $d_i(M) \geq 0$ that estimates the distance towards a state infection; if $d_i(M)$ is 0 then the state is infected. For a test suite T and a mutant M we thus have:

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

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. Once all mutants are reached, then every mutant needs to infect the state, i.e., the infection distance needs to be 0. The infection distance can be measured without executing the tests on the mutants, therefore a single execution of each test is sufficient to calculate reachability and infection.

This results in the following fitness function for weak mutation testing:

$$f_{WM}(T) = f_B(T) + \sum_{M_k \in \mathcal{M}} d_i(M_k,T)$$

3.5 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 (Fraser and Zeller, 2012) we proposed to use an *impact* measurement as an estimator of propagation, which was later also used by Harman et al (2011): 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:

$$d_p(M,T) = \begin{cases} 0 & \text{if } M \text{ can be asserted,} \\ 1 & \text{if } d_i(M,T) > 0, \\ \frac{1}{1+\text{impact}_{min}(M,T)} & \text{if } d_i(M,T) = 0 \end{cases}$$

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 4.3. This results in the following fitness function for strong mutation testing:

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

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.

4 EvoSuite: Scalable Mutation Test Generation

EVOSUITE (Fraser and Arcuri, 2011a, 2013b) is an advanced research prototype that automatically generates JUnit test cases for Java classes. EVOSUITE works on Java bytecode level and collects all necessary information by instrumenting bytecode and analyzing 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 bytecode (such as Scala or Groovy, for example).

EVOSUITE has been successfully applied to a range of different systems (Fraser and Arcuri, 2012b). It is well suited for the experiments in this paper as it has been studied in detail with regard to its parameters (e.g., (Arcuri and Fraser, 2013; Fraser and Arcuri, 2011b, 2012a, 2013c; Fraser et al, 2013)). For more details on the tool and its abilities we refer to (Fraser and Arcuri, 2011a), and for more implementation details we refer to (Fraser and Arcuri, 2013b).

4.1 Mutation Operators

The implemented set of operators is based on the sufficient set used previously (Andrews et al, 2005; Schuler and Zeller, 2010), 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 bytecode 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., (Untch, 1992)), 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., (Bottaci, 2001)), 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 (Harman et al, 2011). As EvoSuite works on bytecode level, we define infection such that the state of the local frame (i.e., the values on the stack) after execution of the mutated bytecode 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:

4.1.1 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.

4.1.2 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.

4.1.3 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.

Table 1 Infection distances for replacements of comparison operators σ. Given the comparison $x \sigma y$, then we define $\delta = x - y$

Operator σ	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
$=$	-	0	$\delta > 0 ? \delta : 0$	$\delta \geq 0 ? \delta + 1 : 0$	$\delta < 0 ? \lvert\delta\rvert : 0$	$\delta \leq 0 ? \lvert\delta\rvert + 1 : 0$
$\neq$	0	-	$\delta \leq 0 ? \lvert\delta\rvert + 1 : 0$	$\delta < 0 ? \lvert\delta\rvert : 0$	$\delta \geq 0 ? \delta + 1 : 0$	$\delta > 0 ? \delta : 0$
$<$	$\delta > 0 ? \delta : 0$	$\delta \leq 0 ? \lvert\delta\rvert + 1 : 0$	-	$\lvert\delta\rvert$	$\delta = 0 ? 1 : 0$	0
$\leq$	$\delta \geq 0 ? \delta + 1 : 0$	$\delta < 0 ? \lvert\delta\rvert : 0$	$\lvert\delta\rvert$	-	0	$\delta = 0 ? 1 : 0$
$>$	$\delta < 0 ? \lvert\delta\rvert : 0$	$\delta \geq 0 ? \delta + 1 : 0$	$\delta = 0 ? 1 : 0$	0	-	$\lvert\delta\rvert$
$\geq$	$\delta \leq 0 ? \lvert\delta\rvert + 1 : 0$	$\delta > 0 ? \delta : 0$	0	$\delta = 0 ? 1 : 0$	$\lvert\delta\rvert$	-

4.1.4 Replace Arithmetic Operator

This mutation operator replaces an arithmetic operator σ in an expression $a\ \sigma\ b$ with all other applicable operators. The infection distance between $a\ \sigma\ b$ and $a\ \sigma'\ b$ 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.

4.1.5 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 $x + 1$.

4.1.6 Replace Comparison Operator

This mutation operator replaces one relational operator with others using the rules defined by Just et al (2012). This operator offers the most opportunities to guide the search, but the guidance depends on the actual operator combinations. Table 1 describes the infection distance calculations for this mutation operator.

4.1.7 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.

4.1.8 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.

4.2 Scalability

To calculate the fitness, in weak mutation testing (see Section 3.4) 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 (2012), 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.

Finally, we add an optional parameter K that limits the fitness function to K mutants at a time. Initially, these K mutants are randomly sampled from the entire set of mutants. Whenever a mutant has been killed, then in this set of K mutants it is replaced with another randomly sampled mutant that is not yet killed. We keep track of the number of iterations of the genetic algorithm in which no new mutants were killed, and if there are X consecutive iterations in which no mutant was killed, then the set of K mutants considered in the fitness function is replaced with a new randomly sampled set of live mutants. For classes with many mutants, this optimization may allow the search to spend more time for evolution rather than on evaluation of all mutants.

4.3 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 3.5). 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 (Fraser and Zeller, 2012).

5 Evaluation

To evaluate the effects of the described approach and optimizations, we ran a large scale empirical study using EVOSUITE. Furthermore, this study investigates the relations between branch coverage, weak and strong mutation testing, in order to provide further insight to get a better understanding of their characteristics and produce, over time, a reliable body of knowledge leading to widely accepted and well-formed theories. In detail, in this paper we aim to answer the following research questions:

RQ1: Does traditional mutation testing help in producing better test suites than optimizing for branch coverage, given fixed computational resources?

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

RQ3: Does the whole test suite approach help mutation testing in producing better test suites than optimizing for branch coverage, given fixed computational resources?

RQ4: How does mutation testing perform with increasing search budget?

RQ5: How does the achieved branch coverage affect the mutation score in traditional and whole test suite generation?

RQ6: How does the mutation score correlate to branch coverage and the number of generated mutants in whole test suite generation?

RQ7: How do mutation score results on 100 randomly selected projects compare to previous results in the literature?

5.1 Case Study

To answer the research questions addressed in this paper, we use the SF100 corpus as case study (Fraser and Arcuri, 2012b). 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[1], consisting of more than two million lines of code having more than 290 thousand bytecode 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

[1] We used the 1.02 version of SF100. The original version in (Fraser and Arcuri, 2012b) had 8,784 classes, but more classes became available once we fixed some classpath issues (e.g., missing jars) in some of the projects.

addressed research questions can generalize to other open source software as well. More details on the SF100 corpus can be found in (Fraser and Arcuri, 2012b) and on our website *www.evosuite.org*.

To answer **RQ7**, we also chose a further case study employed in a recent mutation testing investigation (Deng et al, 2013). That case study is composed of 40 Java classes coming from different libraries and applications, but no information was provided by Deng et al (2013) regarding how these classes were selected.

5.2 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 (Fraser and Arcuri, 2013c)). The fitness function used for strong mutation testing is the one from (Fraser and Zeller, 2012); 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.

In this paper, we carried out three different sets of experiments. The first one is using the entire SF100 corpus, whereas in the second set of experiments we only used 100 classes selected at random. The second set of experiments is smaller as larger search budgets (i.e., for how long EvoSuite is left running) are employed. The third set of experiments is based on the 40 Java classes used in (Deng et al, 2013).

In the first set of experiments, we considered eight different configurations: a default configuration 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 strong mutation testing using the whole test suite we considered four configurations: when all mutants are used (parameter K is ignored), and when a random sample K of 20, 50 and 100 mutants are employed. When this sampling is used, then the number of consecutive iterations without change necessary to reset the random sample was set to 10.

For each class/configuration, experiments were repeated 10 times with different random seeds to take into account the stochastic nature of EvoSuite (Arcuri and Briand, 2012). In total, we had $8{,}963 \times 8 \times 10 = 717{,}040$ 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., non-executable branches and equivalent mutants), one cannot know when the search has covered all the feasible ones.

In the second set of experiments, we used only two configurations: default and strong mutation testing using whole test suite generation and the "all" configuration. For each of the 100 randomly selected classes, we ran EvoSuite with five different search budgets, i.e., $\{4,\ 8,\ 16,\ 32,\ 64\}$ minutes. Each experiment was repeated 10

times with different random seeds. In total, it was a further $100 \times 2 \times 5 \times 10 = 10{,}000$ runs.

In the third set of experiments, we only used the whole test suite approach targeting strong mutation testing. For each of the 40 Java classes in (Deng et al, 2013), we ran EVOSUITE 10 times with different random seeds, each run with a time limit for the search of 2 minutes.

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 4.3). For each of these steps, we used a 10 minute timeout. In contrast to the search, both post-processing steps 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 assertions) 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 on SF100 was at least $((717{,}040 \times 2) + \sum_{i=2}^{6} 2^i \times 2000)/(60 \times 24) = 1{,}168$ days. Such large empirical study required the use of a cluster of computers to run all these experiments. The employed cluster was running a Linux operating system, with 2Ghz frequency and 1GB of memory per running core.

On average, each class in SF100 resulted in 154 mutants (median 58), with some classes having up to $27{,}828$ mutants, leading to a total of $1{,}380{,}302$ mutants. Considering that the SF100 corpus is composed of 100 projects, to the best of our knowledge this is one of the largest studies on mutation testing to date.

All the data resulting from this empirical study were analyzed using statistical methods following the guidelines of Arcuri and Briand (2012). In particular, we used the Vargha-Delaney $\hat{A}_{12}$ effect size and Wilcoxon-Mann-Whitney U-test. Given a performance measure W (e.g., branch coverage or mutation score), $\hat{A}_{xy}$ measures the probability that running algorithm x yields higher W values than running algorithm y. If the two algorithms are equivalent, then $\hat{A}_{xy} = 0.5$. This effect size is independent of the raw values of W, and it becomes a necessity when analyzing the data of large case studies involving artifacts with different difficulty and different orders of magnitude for W. E.g., $\hat{A}_{xy} = 0.7$ 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., $\hat{A}_{12} > 0.5$), but on other classes it can be worse (i.e., $\hat{A}_{12} < 0.5$). 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

Table 2 Average branch coverage, mutation score and size for each configuration, averaged over all classes in SF100. In other words, for each of the 8771 classes we calculated the average of the 10 runs with different seeds. Then, we averaged those 8771 average values.

Name	Coverage	Mutation Score	Size
Base: only coverage	0.57	0.12	23.47
Weak mutation: standard	0.38	0.14	15.41
Weak mutation: "whole"	0.55	0.16	27.71
Strong mutation: standard	0.23	0.16	17.84
Strong mutation "all": "whole"	0.53	0.29	37.99
Strong mutation "20": "whole"	0.52	0.28	39.56
Strong mutation "50": "whole"	0.53	0.29	40.28
Strong mutation "100": "whole"	0.53	0.29	40.74

their selection (e.g., proportion of different application types) would make this type of analysis pointless.

Note that 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.

5.3 Mutation Test Generation with Fixed Search Budget

Table 2 summarizes the results of the eight 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 suites after the minimization post-processing, but excluding assert statements. For strong mutation testing using the whole test suite approach, Table 2 does not show any particular difference among the four different configurations. Therefore, for the rest of paper we will only consider one of them (in particular the one using all the mutants).

RQ1 is concerned with the question of whether traditional mutation testing actually leads to better test suites. If there are no limits to the computational resources available for test generation, then the answer to **RQ1** is clearly yes – past empirical studies have shown that mutation testing is superior to statement and branch coverage (Walsh, 1985) as well as data flow testing (Frankl et al, 1997; Offutt et al, 2004). However, in practice the assumption of unbounded resources may not hold. Given a fixed amount of resources for test generation, does mutation testing lead to better test suites than branch coverage?

Thus, to answer **RQ1**, from Table 2 we can compare the results of standard (i.e., targeting one goal at a time) weak/strong mutation testing with the base (only coverage). Mutation testing leads to much worse branch coverage[2] (-19% for weak mutation, and -34% for strong mutation), but the benefits in terms of mutation scores are

[2] Note that the base case of branch coverage is produced using whole test suite generation; targeting individual branches would lead to lower branch coverage (Fraser and Arcuri, 2013c).

Table 3 Average mutation score results for whole test suite approach for both weak and strong mutation testing. Comparisons are with the traditional approach of targeting each mutant individually. For each class, we counted how often the whole test suite approach led to worse ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and better ($\hat{A}_{12} > 0.5$) mutation score. Values in brackets are for the comparisons that are statistically significant at $\alpha = 0.05$ level; p-values are of the test for $\hat{A}_{12}$ symmetry around 0.5.

Type	Mutation Score		Worse	Equivalent	Better	$\hat{A}_{12}$	p-value
	standard	whole					
Weak	0.14	0.16	604 (125)	5269	2828 (2094)	0.60	< 0.001
Strong	0.16	0.29	266 (42)	2759	5676 (5037)	0.78	< 0.001

minimal (+2% and +4%). The test suites are also significantly smaller due to the low code coverage. The question of course is whether a small improvement on mutation score could offset the negative effect of losing so much branch coverage. The answer to this question likely varies from project to project — considering that code coverage is a common metric in industrial practice it seems unlikely that slightly higher mutation scores make the coverage sacrifice acceptable.

The results in Table 2 might look surprising at a first look. How is it possible to obtain slightly higher mutation scores when branch coverage plummets? 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 the same block before running out of resources, which ultimately might lead to higher mutation score without leading to an increase in the coverage value.

In addition, the costs of mutation analysis are higher than those of optimizing for branch coverage — mutation testing introduces a computational overhead through the additional instrumentation (e.g., to calculate the infection distances, to activate mutants, etc.), and the instrumentation is added at many more places in the program as there are more mutants than branches in code. Overall, this reduces the number of possible fitness evaluations within the time limit for the search, and thus reduces the time the search spends in trying to generate test data for complex branches. Note that these factors would also affect any other type of test generation tool, so this result is not specific to search-based testing.

RQ1*: With 2 minutes search budget, traditional mutation test generation only slightly increases mutation scores, but significantly reduces coverage.*

5.4 Effects of Whole Test Suite Generation

Considering this somewhat negative result, we now turn to the question of whether whole test suite generation can overcome these problems. The second research question investigates whether whole test suite generation in principle leads to better results than the traditional approach of targeting individual coverage goals to mutation test

Table 4 Statistical analysis of mutation score for whole test suite generation when applied for weak and strong mutation testing compared to just targeting branch coverage. For each class, we counted how often mutation testing led to worse ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and better ($\hat{A}_{12} > 0.5$) mutation score. Values in brackets are for the comparisons that are statistically significant at $\alpha = 0.05$ level; p-values are of the test for $\hat{A}_{12}$ symmetry around 0.5.

Name	Mutation Score	Worse	Equivalent	Better	$\hat{A}_{12}$	p-value
Only coverage	0.12	-	-	-	-	-
Weak mutation	0.16	498 (106)	5261	2942 (2196)	0.61	< 0.001
Strong mutation	0.29	338 (122)	2636	5727 (5105)	0.78	< 0.001

generation. To answer **RQ2**, Table 3 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 3 show with high statistical confidence that, for both weak and strong mutation testing, the whole test suite approach significantly improves performance of mutation testing.

RQ2*: Whole test suite generation improves mutation testing significantly over targeting individual mutants, with average effect size $\hat{A}_{12}$ up to 78%.*

The results for **RQ2** shows that the whole test suite approach improves mutation testing. But is this improvement enough to make mutation testing a viable option, and to overcome the issues identified in **RQ1**? To see how mutation testing compares to branch coverage testing (**RQ3**), we once more look at the data (coverage, mutation score and size) presented in Table 2. In addition, statistical analyses on mutation score values are presented in more details in Table 4.

In Table 4 we can see that whole test suite approach for mutation testing leads to higher mutation scores (average effect sizes $\hat{A}_{12}$ are equal to 0.61 for weak mutation and 0.78 for strong mutation). However, the data in Table 2 show that coverage is lower, albeit only by a very small amount when compared to the results for **RQ1**. Would a practitioner prefer using weak mutation testing if that leads to a loss of 2% of code coverage to obtain a +4% of mutation score? What about losing 4% of code coverage with strong mutation testing if that leads to more than double mutation score (i.e., 29% instead of 12%)? Again these are questions that we cannot really answer without controlled empirical studies with human practitioners. However, compared to the traditional approach of targeting individual mutants the benefits in terms of mutation score are much higher, and the drawback in terms of branch coverage is much lower. In practice, this makes it more likely that mutation testing would be a criterion that practitioners could consider to use.

RQ3*: With 2 minutes search budget, whole test suite generation significantly increases mutation scores, but only slightly reduces coverage.*

5.5 Increasing the Search Budget with Whole Test Suite Generation

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 was low enough such that the overhead of mutation testing affected the achieved branch coverage. However, 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, lets the experiments finish in reasonable time.

The data discussed so far was based on the first set of experiments, which used a two minute timeout for the test data generation phase. But this represents only one possible scenario of usage for a test data generation tool such as EVOSUITE. Informal feedback from practitioners that use EVOSUITE in industry (and common sense) pointed out that, although a two minute timeout is a common scenario while implementing a new class, there are many others. Typical examples are, e.g., leaving EVOSUITE running during a lunch break (half an hour), or during a meeting (one hour). Other cases are leaving EVOSUITE running overnight and over the weekend. As the time EVOSUITE is left running does impact the performance, it is important to study how branch coverage and mutation score relate for higher search budgets.

Figure 1 shows the data of the second set of experiments. The default configuration of EVOSUITE (target only branch coverage) is compared with the strong mutation configuration for increasing values of the search budget, going from two to 64 minutes. The boxplots show that, for four minutes, strong mutation configuration leads to slightly lower branch coverage, whereas for higher search budgets there is practically no difference. On the other hand, the mutation score of strong mutation tends to increase for higher budgets (i.e., the upper quartile increases from roughly 60% to 80%, while the median value increases by 4%).

On one hand, for small search budgets such as two minutes, mutation testing does not fully outperform test suite generated targeting only branch coverage. It is up to the final user to decide which test criterion he deems more important. On the other hand, for higher search budgets, mutation testing achieves the same code coverage, but higher mutation scores. In both cases, the novel techniques presented in this paper make mutation testing a viable option to use for the final practitioners.

RQ4*: In our experiments, 4 minutes search budget were sufficient to avoid any loss in code coverage during mutation test generation.*

5.6 Influence of the Difficulty of Testing a Class

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

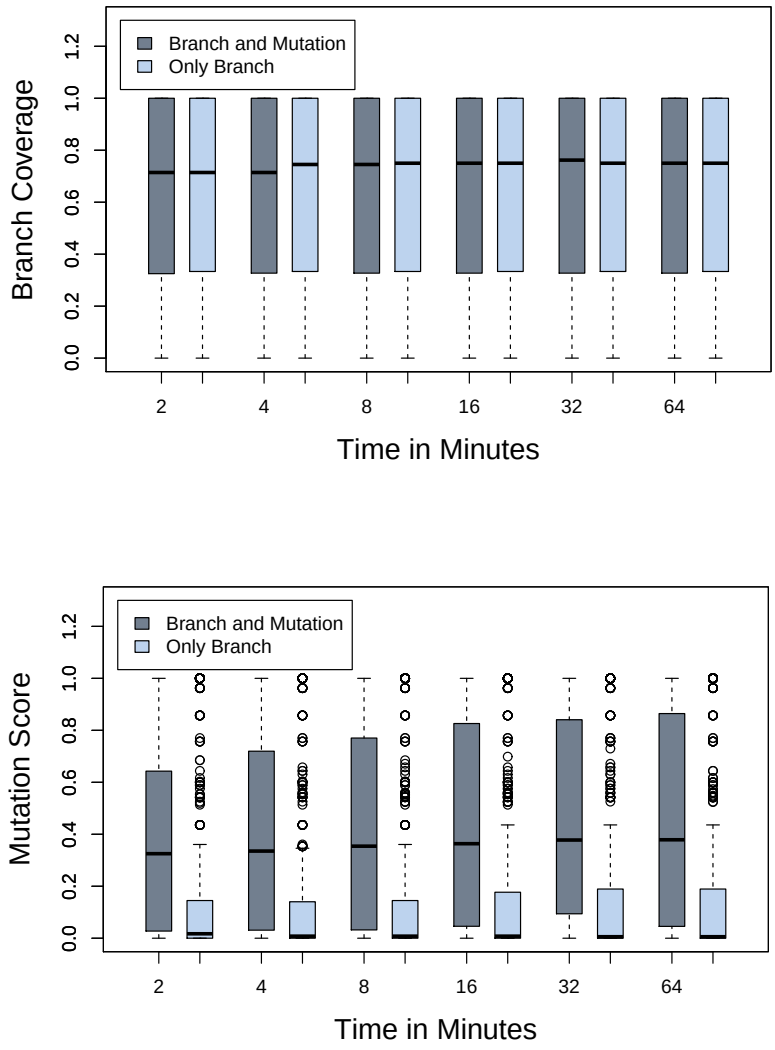

Fig. 1 Branch coverage and mutation scores for different search budgets (expressed in minutes, from 2 to 64). Data is based on 100 classes randomly selected from SF100. Two configurations are compared: the default one using only branch coverage, and strong mutation with whole test suite approach.

experiments (Fraser and Arcuri, 2012b) we know that there are several classes for which EvoSuite 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 EvoSuite does not support yet. To better understand how the difficulty of a class affects mutation testing using the traditional and whole test suite generation, we consider the branch coverage achieved on a class as a proxy measurement of the

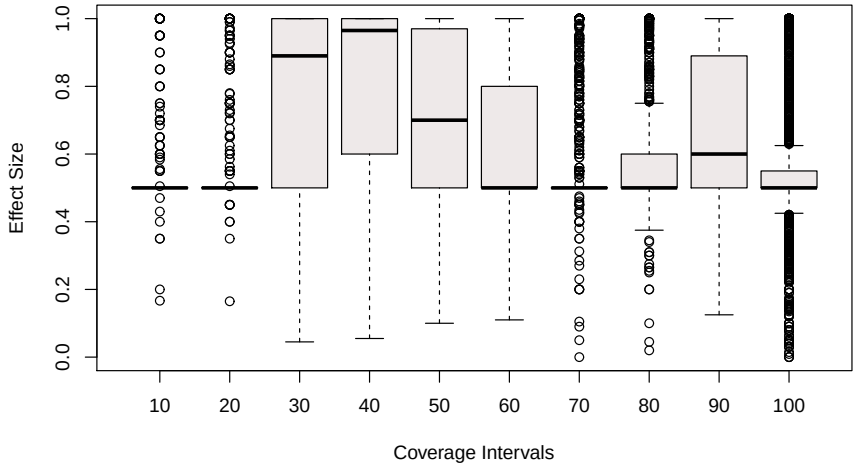

Fig. 2 Effect sizes $\hat{A}_{12}$ for the whole test suite generation approach for weak mutation testing. For each 10% code coverage interval, we draw a boxplot of the $\hat{A}_{12}$ effect sizes for each class within that interval. Labels show the upper limit (inclusive). For example, the group 40% represent all the classes with average coverage greater than 30% and lower or equal to 40%.

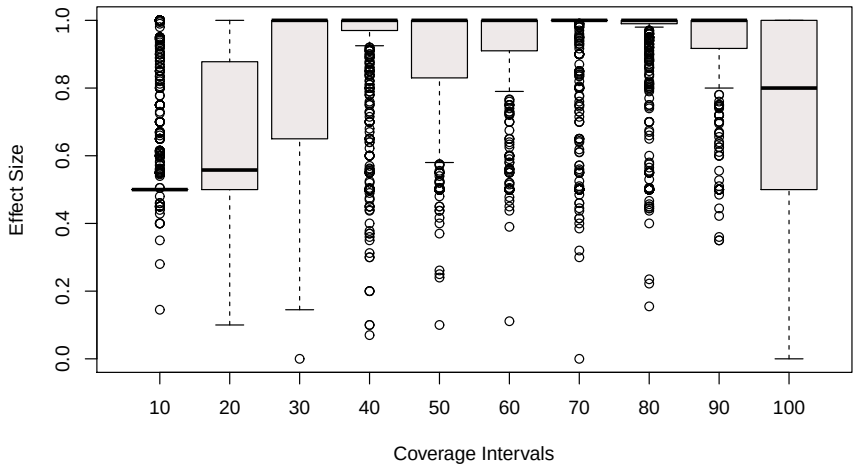

Fig. 3 Effect sizes $\hat{A}_{12}$ for the whole test suite generation approach for strong mutation testing. For each 10% code coverage interval, we draw a boxplot of the $\hat{A}_{12}$ effect sizes for each class within that interval. Labels show the upper limit (inclusive). For example, the group 40% represent all the classes with average coverage greater than 30% and lower or equal to 40%.

difficulty of a class. Thus, **RQ5** studies and compares the mutation score based on the achieved branch coverage of the whole test suite approach.

Figure 2 shows a boxplot comparing the whole test suite approach for weak mutation score and the traditional approach of targeting each mutant separately, and Figure 3 shows the same boxplot for strong mutation. In other words, the $8{,}771$ classes were divided in 10 groups, based on the achieved branch coverage. As branch cover-

age is different between weak and strong mutation testing, the partitions in Figure 2 and Figure 3 are not necessarily the same.

As we can see in Figure 2 and Figure 3, there is not much difference in mutation score on the classes for which EVOSUITE achieves low coverage, e.g., below 10%. This is as expected: If the coverage is this low for a class, then it represents some problem that EVOSUITE cannot overcome; if only few parts of a class are executed, there is no chance to increase the mutation score. Similarly, the majority of classes for which we achieve very high coverage can be simply trivial, and so for many of them it would not be possible to generate challenging enough mutants that would be able to distinguish among different EVOSUITE configurations.

On one hand, the graph in Figure 3 for strong mutation testing follows a kind of regular pattern. It starts with nearly no difference (median effect size close to 0.5 for coverage less than 10%), and soon increases up to very strong effect size (median effect size 1 already for coverage 40%). The effect size stays strong up to coverage 90%, where it finally decreases (but it is still higher than 0.5). This is consistent with the above explanation, i.e., on trivial classes there is little to gain.

On the other hand, the graph in Figure 2 sees a drop in the improvement around 60% coverage, and from there upwards there is an improvement, but it is clearly smaller than before that. Our conjecture is that the improvement gained from using whole test suite generation is generally higher the more difficult the problem at hand is. Easier classes likely result in higher branch coverage even with low search budgets, so the 60% upwards range in Figure 2 can be considered easier classes, where the effect size is smaller. Furthermore, strong mutation testing is more difficult by construction (i.e., more expensive fitness function), which is why the improvement seen in Figure 3 is higher than for weak mutation testing. This is also confirmed by the overall $\hat{A}_{12}$ values of 0.6 for weak mutation versus 0.78 for strong mutation testing, as seen in Table 3.

RQ5*: The more difficult a class, the larger the improvement gained by whole test suite generation.*

5.7 Correlations with the 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 relation, which, in the long run, would lead to obtain a more reliable body of knowledge to study, analyze and investigate novel, more performant testing techniques.

To investigate the correlation between branch coverage and mutation score in whole test suite generation (**RQ6**) we calculated the average branch coverage and mutation score achieved by EVOSUITE for each class when using strong mutation testing and the whole test suite approach. Figure 4 displays this data as a scatter plot.

There are two notable clusters of classes around the corners of this plot. There are classes with very low branch coverage and mutation scores (bottom-left corner);

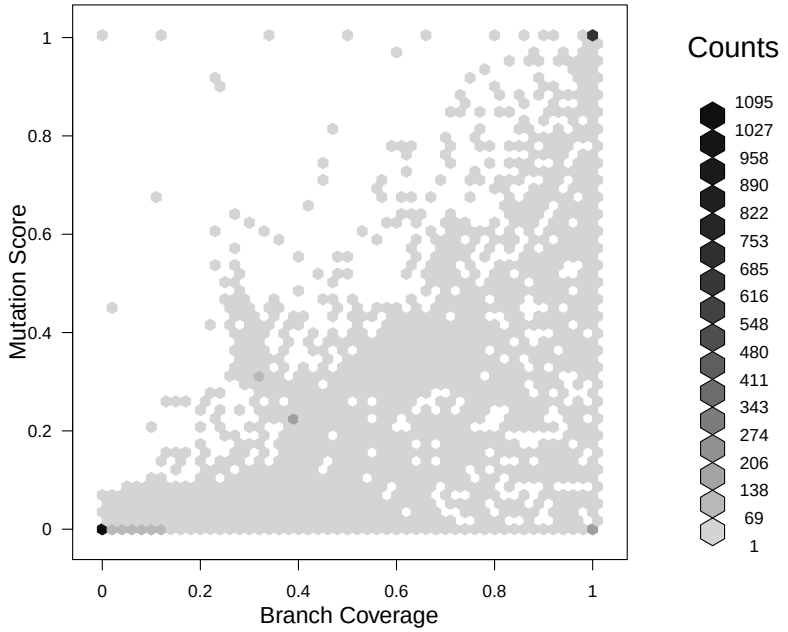

Fig. 4 Scatter plot of average branch coverage vs. average mutation score for each class.

these are classes which represent problems EVOSUITE cannot handle. Given the low search budget we used during the experiments, classes with branch coverage and mutation score around 100% (top-right corner) are likely trivial classes that pose no problems.

For classes outside these two extreme cases, we see that usually branch coverage is higher than the mutation score, which shows that achieving state infection and 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 4). In these cases it is likely that the API of the CUT simply is not rich enough (e.g., not enough observer methods) to reveal sufficient internal details that would allow to produce assertions, or EVOSUITE'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 4, we studied the correlation between the mutation score and branch coverage. There is a 0.65 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 $[0.64,0.66]$.

To provide more insight, Figure 5 shows the correlation between mutation score and number of mutants in the CUT. As one would expect, more mutants means lower mutation score. In numbers, there is a -0.48 linear correlation (p-value very close to zero, and confidence interval $[-0.49, -0.46]$). One could have expected a stronger correlation (e.g., close to -1) but, as shown in Figure 5, there are three different clusters. As discussed above, there are cases which EVOSUITE simply is not able

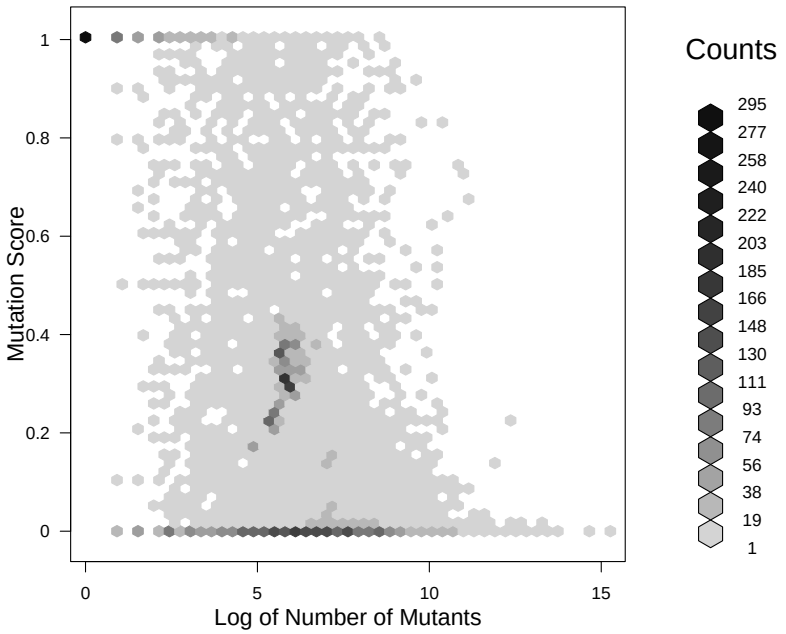

Fig. 5 Scatter plot of average branch coverage vs. average mutation score for each class.

to handle, and others that are trivial for EvoSuite. In both these types of classes, the number of mutants is kind of irrelevant for the mutation score. The third cluster in the middle of the plot represents "regular" behavior, with 100-1000 mutants and mutation scores in the range of 20-40%.

These results point out that there are still many open problems in mutation testing, as just executing more code does not always mean killing more mutants (correlation 65%), even with advanced techniques as implemented inside EvoSuite. Furthermore, more mutants does not always mean a more difficult mutation testing problem (correlation -48%). Characterizing the difficulty of a case study only based on the number of generated mutants is a too coarse grained option.

RQ6*: There is a* 0.65 *positive correlation between mutation score and branch coverage, and a negative* -0.48 *with number of mutants.*

5.8 Low Mutation Scores

One interesting thing to note about Table 2 is that average mutation scores are in the range of 12%–29%. Even for larger search budgets (e.g., 64 minutes instead of two), the average mutation score does not go over 46% (for a subset of 100 randomly chosen classes from SF100). This is in contrast with previous reports (e.g.,

summarized by Harman et al (2011)) that 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 (Fraser and Arcuri, 2012b); this is reflected by the also lower branch coverage values.

As discussed by Harman et al (2011), most of the previous work on mutation testing has been evaluated mainly on simple examples or relatively small systems of no more than a few hundreds lines of code (for details, see Table 1 in (Harman et al, 2011)). For example, Offutt and Lee (1994) used 11 Fortran subroutines, where the longest was only 29 lines of code. Although large empirical studies were not common in the 90s (e.g., due to much more limited computational resources), this poses problems when meta-analyses across studies (especially across decades) are carried out.

Comparing mutation score results in the literature with what EVOSUITE achieves on SF100 would be misleading, for at least two main reasons. First, SF100 is composed of 100 real-world projects, which involve GUI widgets, accesses to databases, network connections (e.g., TCP and UDP), writing/deleting files, multi-threading, etc. The largest project in SF100 is composed of 2,189 classes, where the most complex class has 2,480 bytecode level branches (see (Fraser and Arcuri, 2012b) for details). It is not necessary for a test generation tool to address all these types of problems to be useful, but it is necessary to bear in mind what types of programs tools are intended to work on and what they are evaluated on. Results on real, complex software are obviously going to be worse than results on toy examples. Second, mutation scores depend on the quality of the test suites. In many empirical studies (e.g., (Baker and Habli, 2012; Mateo et al, 2012)), the test suites are already present (e.g., manually created, provided with the project). Comparing the test suites automatically generated by EVOSUITE with such manually written test suites is not the same as comparing EVOSUITE with other automated test case generation tools.

Of particular interest is the work of Harman et al (2011) and the work of Just et al (2012). Harman et al (2011) developed and evaluated an automated tool to generate test cases. The evaluation was carried out on 17 C programs, ranging from 35 to 9,564 lines of code, and the average mutation score was 69%. This higher mutation score, compared to what is reported in this paper, has at least three (complementary) explanations: (1) procedural code (e.g., C) is different from object-oriented software (e.g., Java), as in the latter there is the extra problem of calling observers to assert the internal state of the objects; (2) the case study, although reasonable and sufficient for the addressed research questions, was hand-picked, which leads to several of the problems previously discussed; (3) much higher search budget was used, up to an average 2,762 minutes (i.e., nearly two days), compared to the two to 64 minutes used in this paper. The latter two arguments also hold when comparing to earlier work of Fraser and Zeller (2012) on Java code (40.43% mutation score on average).

In their work, Just et al (2012) used 10 medium-size Java projects, ranging from 3,647 to 116,750 lines of code. Mutation scores were calculated based on the test cases shipped with those projects. In total the mutation score on all projects was 43.5%, but there was large variance between projects, ranging from 8.2% (*trove*)

to 94.7% (*num4j*). To see how easy the choice of case studies influences the result, consider that simply omitting the two projects with lowest scores (*trove* and *itext*) would have increased the mutation score from 43.5 to 74%! This is a further example of the difficulties of trying to generalize and compare results from empirical analyses based on hand-picked case studies[3].

In Table 2 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., (Offutt and Lee, 1994)), 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.

Indeed, killing a mutant involved the three steps of reachability, infection, propagation (the RIP model (DeMillo and Offutt, 1991)). The achieved branch coverage gives us an estimate for how much reachability is a cause of low mutation scores. With an average branch coverage of 60% we see that clearly already reachability is difficult on real software. Note that the branch coverage value does not allow us to exactly quantify reachability as neither branches nor mutants are uniformly distributed. Infection requires a local state change; this is what weak mutation testing requires. When targeting weak mutation testing using whole test suite generation, EvoSuite achieved an average *weak* mutation score of 35.8% (not shown in the tables); in contrast, the best achieved *strong* mutation score using the same search budget is 29%. We can therefore estimate that, for up to $\frac{35.8-29}{35.8} = 19\%$ of the mutants that infected the state, propagation was an issue. Consequently, further research in all three aspects, reachability, infection, and propagation, is called for.

When an empirical study provides evidence that is in contrast to what is reported in the literature, there is always a threat that the empirical study is flawed. EvoSuite has been carefully tested, but it is not possible to guarantee its correctness. Or it might just be that search-based testing is not well suited for mutation testing. To shed light on these potential threats, we look at the raw data of the SBST'13 tool competition (Bauersfeld et al, 2013; Fraser and Arcuri, 2013a) that EvoSuite won. In the SBST'13 competition, the competitors did not have access to the chosen benchmark (everything was run remotely by the competition organizers). Code coverage and mutation scores were calculated with external tools, i.e., Cobertura and Javalanche (note that EvoSuite does not use any external tools for these calculations). The data of that competition strongly confirm the results presented in this paper. Although EvoSuite could achieve reasonable branch coverage (57%), the mutation score was much lower (13%). The second best tool (Randoop) achieved 39% branch coverage, but only 6% mutation score. Manual tests that were present in the benchmark had high branch coverage (81%), but comparatively low mutation score (only 17%).

To better understand why our results on the SF100 corpus are different from the literature, Table 5 shows the results of our third empirical study. This study is based on the same set of 40 Java classes used by Deng et al (2013). On this set of

[3] This has nothing to do with whether what proposed in (Just et al, 2012) is valuable or not. In particular, it is important to stress out that, compared to the literature, the case study in (Just et al, 2012) is among the largest and most variegated

Table 5 Average branch coverage, mutation score and size for each class in (Deng et al, 2013).

Name	Coverage	Mutation Score	Size
BoundedQueue	0.94	0.87	84.30
BoyerMoore	0.97	0.64	49.30
cal	0.82	0.78	16.00
Calculation	0.97	0.85	46.20
checkIt	0.86	0.79	4.50
CheckPalindrome	0.80	0.93	4.80
Count2s	0.93	0.78	12.50
countPositive	0.80	0.95	16.50
findLast	0.67	0.70	24.70
findVal	0.67	0.71	19.70
Gaussian	0.93	0.93	29.30
GaussianElimination	0.94	0.94	83.00
Heap	0.90	0.69	105.10
lastZero	0.80	1.00	30.20
LRS	0.91	0.96	10.40
MergeSort	0.92	0.93	27.40
Node	1.00	1.00	1.10
numZero	0.80	0.95	13.50
oddOrPos	0.86	0.95	20.00
org.apache.lucene.analysis.CharArraySet	0.64	0.47	287.00
org.apache.lucene.document.Document	0.82	0.65	150.75
org.apache.lucene.document.NumberTools	0.78	0.83	12.44
org.apache.lucene.index.FieldInfos	0.61	0.35	188.00
org.apache.lucene.index.IndexModifier	0.30	0.16	43.14
org.apache.lucene.index.Term	0.81	0.85	85.44
org.apache.lucene.search.BooleanQuery	0.40	0.45	250.50
org.apache.lucene.search.function.CustomScoreProvider	0.88	0.83	67.60
org.apache.lucene.search.TermRangeQuery	0.51	0.78	58.20
org.joda.time.field.MillisDurationField	0.93	0.93	70.00
power	0.80	0.97	6.50
printPrimes	0.91	0.76	3.90
Queue	1.00	0.90	75.00
QuickSort	0.95	0.81	43.00
RecursiveSelectionSort	0.88	0.94	26.40
SearchString	0.89	0.66	91.10
Stack	0.91	0.89	91.90
stats	0.80	0.84	14.60
sum	0.67	0.96	13.00
TestPat	1.00	0.93	57.20
trashAndTakeOut	0.86	0.96	13.30
twoPred	0.86	0.94	7.00
Average	0.83	0.82	53.38

classes, EVOSUITE performed well, with an average branch coverage of 83%, and an average mutation score of 82%. This is in clear contrast with the values 53% (branch coverage) and 29% (mutation score) reported in Table 2. The employed tool was the same (i.e.,EVOSUITE), the only difference was the employed case study. This confirms our conjectures on the influence of the underlying case study, and that commonly used examples in the literature do not include some of the problems that can be found in real-world code.

RQ7*: Mutation score results in the literature over-estimate what can be obtained today on real-world software.*

6 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 (Fraser and Arcuri, 2012b). 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 EVOSUITE, and we did not compare with other tools. The results of the SBST'13 tool competition provides support that using EVOSUITE is sufficient to address **RQ1**. Answering **RQ2**, **RQ3**, and **RQ4** require to implement all the novel techniques within the same tool, as to minimize the effects of external confounding factors. Finally, for answering **RQ5**, **RQ6** and **RQ7**, there is the need of a tool that can handle SF100. We are aware of no other tool that can be *automatically* and *safely* applied to SF100. Extending existing tools to apply to SF100 is not just a matter of adding a security manager, as there are many details to consider (see (Fraser and Arcuri, 2013b) for more discussions on some of the technical challenges involved).

All analyses in this paper are automated. This means that we do not know whether any live mutants remaining are equivalent, or if EVOSUITE just did not manage to generate a killing test case. As discussed in Section 2.2, manual analysis would be necessary to decide on equivalence. However, our research questions are independent on whether the remaining mutants are equivalent or not. Nevertheless, investigating the remaining mutants on equivalence, or potential for improving EVOSUITE, would be interesting avenues for future research.

To allow reproducibility of the results presented in this paper, both the SF100 corpus and EVOSUITE are freely available from our webpage at *http://www.evosuite.org/*.

7 Conclusions

Mutation testing can be used to evaluate how good a test suite is at revealing possible faults, and it can also be used during automated test case generation to create test suites that are better than those optimized only for code coverage. Mutation testing is a promising technique, but it 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* (Fraser and Arcuri, 2013c) 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 this is also the case for mutation testing, by performing an empirical study on 100 Java projects randomly selected from SourceForge, i.e., the SF100 corpus (consisting of 8,963 classes, for a total of more than two million lines of code). Besides the whole test suite approach, EVOSUITE also includes several novel optimizations for mutation testing, such as the use of infection conditions, optimized mutation operators, and prioritized test execution.

Our results show that using standard mutation testing in test case generation would not scale up to the complexity of real-world software. Given a fixed same search budget, mutation testing is so expensive that no time is left to achieve high code coverage (e.g., branch coverage dropped from 57% to 23%). On the other hand, our novel techniques make mutation testing a viable option for software practitioners as they make possible, within the same search budget, to achieve same/similar code coverage but much higher mutation score. Test suites that have higher mutation scores are of practical value, as they provide better help at detecting faults.

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 bytecode 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. Finally, the results on SF100 clearly pointed out that further research in test generation is necessary in order to achieve improvements with respect to reachability, infection, and propagation.

To learn more about EVOSUITE and SF100, visit our Web site:

`http://www.evosuite.org/`

Acknowledgements

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

References

Acree AT (1980) On mutation. Phd thesis, Georgia Institute of Technology, Atlanta, Georgia

Adamopoulos K, Harman M, Hierons RM (2004) How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Genetic and Evolutionary Computation Conference (GECCO), Seattle, Washington, USA, pp 1338–1349

Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th international conference on Software engineering, ACM, ICSE ’05, pp 402–411

Arcuri A (2013) It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (STVR) 23(2):119–147

Arcuri A, Briand L (2012) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) DOI: 10.1002/stvr.1486

Arcuri A, Fraser G (2013) Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering (EMSE) pp 1–30, dOI: 10.1007/s10664-013-9249-9

Ayari K, Bouktif S, Antoniol G (2007) Automatic mutation test input data generation via ant colony. In: Genetic and Evolutionary Computation Conference (GECCO), ACM, New York, USA, pp 1074–1081

Baker R, Habli I (2012) An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Transactions on Software Engineering (TSE)

Baldwin D, Sayward FG (1979) Heuristics for determining equivalence of program mutations. techreport 276, Yale University, New Haven, Connecticut

Baudry B, Fleurey F, Jzquel JM, Le Traon Y (2005) Automatic test cases optimization: a bacteriologic algorithm. IEEE Software 22(2):76–82

Bauersfeld S, Vos T, Lakhotia K, Poulding S, Condori N (2013) Unit testing tool competition. In: International Workshop on Search-Based Software Testing (SBST), pp 414–420

Bottaci L (2001) A genetic algorithm fitness function for mutation testing. In: SEMINAL 2001: International Workshop on Software Engineering using Metaheuristic Inovative Algorithms, a workshop at 23rd Int. Conference on Software Engineering, pp 3–7

Budd TA (1980) Mutation analysis of program test data. Phd thesis, Yale University, New Haven, Connecticut

DeMillo RA, Offutt AJ (1991) Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9):900–910

DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. Computer 11(4):34–41

Deng L, Offutt J, Li N (2013) Empirical evaluation of the statement deletion mutation operator. In: IEEE International Conference on Software Testing, Verification and Validation (ICST)

Fleyshgakker VN, Weiss SN (1994) Efficient mutation analysis: A new approach. In: ISSTA '94: Proceedings of the International Symposium on Software Testing and Analysis, Seattle, Washington, pp 185–195

Frankl PG, Weiss SN, Hu C (1997) All-uses vs mutation testing: an experimental comparison of effectiveness. Journal of Systems and Software 38(3):235–253

Fraser G, Arcuri A (2011a) EvoSuite: Automatic test suite generation for object-oriented software. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp 416–419

Fraser G, Arcuri A (2011b) It is not the length that matters, it is how you control it. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 150 – 159

Fraser G, Arcuri A (2012a) The seed is strong: Seeding strategies in search-based software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp 121–130

Fraser G, Arcuri A (2012b) Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp 178–188

Fraser G, Arcuri A (2013a) Evosuite at the SBST 2013 tool competition. In: International Workshop on Search-Based Software Testing (SBST), pp 406–409

Fraser G, Arcuri A (2013b) EvoSuite: On the challenges of test case generation in the real world (tool paper). In: IEEE International Conference on Software Testing, Verification and Validation (ICST)

Fraser G, Arcuri A (2013c) Whole test suite generation. IEEE Transactions on Software Engineering 39(2):276–291

Fraser G, Zeller A (2012) Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering (TSE) 28(2):278–292

Fraser G, Arcuri A, McMinn P (2013) Test suite generation with memetic algorithms. In: Genetic and Evolutionary Computation Conference (GECCO)

Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In: PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, pp 213–223

Hamlet RG (1977) Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering 3(4):279–290

Harman M, Jia Y, Langdon WB (2011) Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, ESEC/FSE '11, pp 212–222

Hierons RM, Harman M, Danicic S (1999) Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability 9(4):233–262

Howden WE (1982) Weak mutation testing and completeness of test sets. IEEE Trans Softw Eng 8(4):371–379

Jia Y, Harman M (2009) Higher order mutation testing. Journal of Information and Software Technology 51(10):1379–1393

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (TSE) 37(5):649–678

Just R, Kapfhammer GM, Schweiggert F (2012) Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In: Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering, IEEE Computer Society, ISSRE '12, pp 11–20

Just R, Ernst MD, Fraser G (2013) Using state infection conditions to detect equivalent mutants and speed up mutation analysis. arXiv preprint arXiv:13032784

Korel B (1990) Automated software test data generation. IEEE Transactions on Software Engineering pp 870–879

Mateo PR, Usaola MP, Aleman JLF (2012) Validating 2nd-order mutation at system level. IEEE Transactions on Software Engineering (TSE)

McMinn P (2004) Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2):105–156

Offutt AJ (1992) Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology 1(1):5–20

Offutt AJ, Craft WM (1994) Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability 4(3):131–154

Offutt AJ, Lee SD (1991) How strong is weak mutation? In: Proceedings of the symposium on Testing, analysis, and verification, ACM, TAV4, pp 200–213

Offutt AJ, Lee SD (1994) An empirical evaluation of weak mutation. IEEE Trans Softw Eng 20(5):337–344

Offutt AJ, Pan J (1997) Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability 7(3):165–192

Offutt AJ, Untch RH (2001) Mutation testing for the new century. Kluwer Academic Publishers, Norwell, MA, USA, chap Mutation 2000: uniting the orthogonal, pp 34–44

Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: ICSE '93: Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, pp 100–107

Offutt AJ, Ma YS, Kwon YR (2004) An experimental mutation system for Java. ACM SIGSOFT Software Engineering Notes 29(5):1–4

Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for Java. In: OOPSLA'07: Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Application, ACM, pp 815–816

Papadakis M, Malevris N (2010) Automatic mutation test case generation via dynamic symbolic execution. In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, pp 121–130

Patrick M, Alexander R, Oriol M, Clark JA (2013) Using mutation analysis to evolve subdomains for random testing. In: International Workshop on Mutation Analysis

Schuler D, Zeller A (2010) (Un-)Covering equivalent mutants. In: ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation, IEEE Computer Society, pp 45–54

Staats M, Whalen MW, Heimdahl MP (2011) Programs, tests, and oracles: the foundations of testing revisited. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp 391–400

Untch RH (1992) Mutation-based software testing using program schemata. In: Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92), Raleigh, North Carolina, pp 285–291

Walsh PJ (1985) A measure of test case completeness (software, engineering). PhD thesis, State University of New York at Binghamton, Binghamton, NY, USA

Wong WE, Mathur AP, Maldonado JC (1995) Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In: Software Quality and Productivity: Theory, practice and training, Chapman & Hall, Ltd., London, UK, UK, pp 258–265

Zhang L, Xie T, Zhang L, Tillmann N, de Halleux J, Mei H (2010) Test generation via dynamic symbolic execution for mutation testing. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, IEEE Computer Society,

ICSM ’10, pp 1–10