Whole Test Suite Generation

Gordon Fraser, *Member, IEEE* and Andrea Arcuri, *Member, IEEE.*

Abstract—Not all bugs lead to program crashes, and not always is there a formal specification to check the correctness of a software test's outcome. A common scenario in software testing is therefore that test data is generated, and a tester manually adds test oracles. As this is a difficult task, it is important to produce small yet representative test sets, and this representativeness is typically measured using code coverage. There is, however, a fundamental problem with the common approach of targeting one coverage goal at a time: Coverage goals are not independent, not equally difficult, and sometimes infeasible – the result of test generation is therefore dependent on the order of coverage goals and how many of them are feasible. To overcome this problem, we propose a novel paradigm in which *whole test suites* are evolved with the aim of covering *all* coverage goals at the same time, while keeping the total size as small as possible. This approach has several advantages, as for example its effectiveness is not affected by the number of infeasible targets in the code. We have implemented this novel approach in the EvoSuite tool, and compared it to the common approach of addressing one goal at a time. Evaluated on open source libraries and an industrial case study for a total of 1,741 classes, we show that EvoSuite achieved up to 188 times the branch coverage of a traditional approach targeting single branches, with up to 62% smaller test suites.

Index Terms—Search based software engineering, length, branch coverage, genetic algorithm, infeasible goal, collateral coverage

1 Introduction

It is widely recognized that software testing is an essential component of any successful software development process. A software test consists of an input that executes the program and a definition of the expected outcome. Many techniques to automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. Yet, the problem of the expected outcome persists, and has become known as the *oracle problem*. Sometimes, essential properties of programs are formally specified, or have to hold universally such that no explicit oracles need to be defined (e.g., programs should normally not crash). However, in the general case one cannot assume the availability of an automated oracle. This means that, if we produce test inputs, then a human tester needs to specify the oracle in terms of the expected outcome. To make this feasible, test generation needs to aim not only at high code coverage, but also at small test suites that make oracle generation as easy as possible.

• *Gordon Fraser is with Saarland University – Computer Science, Saarbrücken, Germany, email: fraser@cs.uni-saarland.de. Andrea Arcuri is with the Certus Software V&V Center at Simula Research Laboratory, P.O. Box 134, Lysaker, Norway, email: arcuri@simula.no*

```
 1 public class Stack {
 2   int[] values = new int[3];
 3   int size = 0;
 4   void push(int x) {
 5     if(size >= values.length)       ⇐ Requires a full stack
 6       resize();
 7     if(size < values.length)      ⇐ Else branch is infeasible
 8       values[size++] = x;
 9   }
10   int pop() {
11     if(size > 0)⇐ May imply coverage in push and resize
12       return values[size−−];
13     else
14       throw new EmptyStackException();
15   }
16   private void resize(){
17     int[] tmp = new int[values.length ∗ 2];
18     for(int i = 0; i < values.length; i++)
19       tmp[i] = values[i];
20     values = tmp;
21   }
22 }
```

Fig. 1. Example stack implementation: Some branches are more difficult to cover than others, some lead to coverage of further branches, and some can be infeasible.

A common approach in the literature is to generate a test case for each coverage goal (e.g., branches in branch coverage), and then to combine them in a single test suite (e.g., see [43]). However, the size of a resulting test suite is difficult to predict, as a test case generated for one goal may implicitly also cover any number of further coverage goals. This is usually called collateral or serendipitous coverage (e.g., [25]). For example, consider the stack implementation in Figure 1: Covering the true branch in Line 11 is necessarily preceded by the true branch in Line 7, and may or may not also be preceded by the true branch in Line 5. In fact, the order in which each goal is select can thus play a major role, as there can be dependencies among goals. Although there have been attempts to exploit collateral coverage to optimize test generation (e.g., [25]), to the best of our knowledge there is no conclusive evaluation in the literature of their effectiveness.

```
Stack stack0 = new Stack();
try {
  stack0.pop();
} catch(EmptyStackException e) {
}
```

```
Stack stack0 = new Stack();
int int0 = −510;
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.push(int0);
stack0.pop();
```

Fig. 2. Test suite consisting of two tests, produced by EvoSuite for the `Stack` class shown in Figure 1: All feasible branches are covered.

There are further issues to the approach of targeting one test goal at a time: Some targets are more difficult to cover than others. For example, covering the true branch in Line 5 of the stack example is more difficult than covering the false branch of the same line, as the true branch requires a `Stack` object which has filled its internal array. Furthermore, coverage goals can be *infeasible*, such that there exists no input that would cover them. For example, in Figure 1 the false branch of the `if` condition in Line 7 is infeasible. Even if this particular infeasible branch may be easy to detect this is not true in general (it is, in fact, an undecidable problem [23]), and thus targeting infeasible goals will per definition fail and the effort would be wasted. This leads to the question of how to properly allocate how much of the testing budget (e.g., the maximum total time allowed for testing by the user) is used for each target, and how to redistribute such budget to other uncovered targets when the current target is covered before its budget is fully consumed. Although in the literature there has been preliminary work based on software metrics to predict the difficulty of coverage goals in procedural code [31], its evaluation and usefulness on object-oriented software is still an open research question.

In this paper we evaluate a novel approach for test data generation, which we call *whole test suite generation*, that improves upon the current approach of targeting one goal at a time. We use an evolutionary technique [1], [34] in which, instead of evolving each test case individually, we evolve all the test cases in a test suite at the same time, and the fitness function considers all the testing goals simultaneously. The technique starts with an initial population of randomly generated test suites, and then uses a Genetic Algorithm to optimize towards satisfying a chosen coverage criterion, while using the test suite size as a secondary objective. At the end, the best resulting test suite is minimized, giving us a test suite as shown in Figure 2 for the `Stack` example from Figure 1. With such an approach, most of the complications and downsides of the one target at a time approach either disappear or become significantly reduced. The technique is implemented as part of our testing tool EvoSuite [18], which is freely available online.

This novel approach was first described in [17], and this paper extends that work in several directions, by for example using a much larger and variegated case study, verifying that the presence of infeasible branches has no negative impact on performance, and by providing theoretical analyses to shed more lights on the properties of the proposed approach. In particular, we demonstrate the effectiveness of EvoSuite by applying it to 1,741 classes coming from open source libraries and an industrial case study (Section 5); to the best of our knowledge, this is the largest evaluation of search-based testing of object-oriented software to date. Because to effectively address the problem of test suite generation we had to develop specialized search operators, there would be no guarantee on the *convergence* property of the resulting search algorithm. To cope with this problem, we formally prove the convergence of our proposed technique.

The results of our experiments show strong statistical evidence that the EvoSuite approach yields significantly better results (i.e., either higher coverage or, if same coverage, then smaller test suites) compared to the traditional approach of targeting each testing goal independently. In some cases, EvoSuite achieved up to 188 times higher coverage on average, and test suites that were 62% smaller while maintaining the same structural coverage. Furthermore, running EvoSuite with a constrained budget (one million statement executions during the search, up to a maximum 10 minutes timeout) resulted in an impressive 83% of coverage on average on our case study.

The paper is organized as follows. Section 2 provides background information. The novel approach of evolving whole test suites is described in Section 3, and the details of our EvoSuite tool follow in Section 4. The empirical study we conducted to validate our approach is presented and discussed in Section 5. Convergence is formally proven in Section 6. Threats to validity of our study are analyzed in Section 7, and finally, Section 8 concludes the paper.

2 Background

Coverage criteria are commonly used to guide test generation. A coverage criterion represents a finite set of coverage goals, and a common approach is to target one such goal at a time, generating test inputs either symbolically or with a search-based approach. The predominant criterion in the literature is branch coverage, but in principle any other coverage criterion or related techniques such as mutation testing [29] are amenable to automated test generation.

Solving path constraints generated with symbolic execution is a popular approach to generate test data [50] or unit tests [51], and dynamic symbolic execution as an extension can overcome a number of problems by combining concrete executions with symbolic execution (e.g., [22], [39]). This idea has been implemented in tools like DART [22] and CUTE [39], and is also applied in Microsoft's parametrized unit testing tool PEX [42] or in the Dsc [28] tool.

Meta-heuristic search techniques have been used as an alternative to symbolic execution based approaches (see [1], [34] for surveys on this topic). The application of search for test data generation can be traced as back to the 70s [35], where the key concepts of *branch distance* [30] and *approach level* [48] were introduced to help search techniques in generating the right test data. A promising avenue also seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [12], [27], [33]), alleviating some of the problems both approaches have.

Search-based techniques have also been applied to test object-oriented software using method sequences [21], [43] or strongly typed genetic programming [37], [47]. When generating test cases for object-oriented software, since the early work of Tonella [43], authors have tried to deal with the problem of handling the length of the test sequences, for example by penalizing the length directly in the fitness function. However, longer test sequences can lead to achieve higher code coverage [5], yet properly handling their growth/reduction during the search requires special care [19].

Most approaches described in the literature aim to generate test suites that achieve as high as possible branch coverage. In principle, any other coverage criterion is amenable to automated test generation. For example, mutation testing [29] is often considered a worthwhile test goal, and has been used in a search-based test generation environment [21]. When test cases are sought for individual targets in such coverage based approaches, it is important to keep track of the accidental collateral coverage of the remaining targets. Otherwise, it has been proven that random testing would fare better under some scalability models [10]. Recently, Harman et al. [25] proposed a search-based multi-objective approach in which, although each goal is still targeted individually, there is the secondary objective of maximizing the number of collateral targets that are accidentally covered. However, no particular heuristic is used to help covering these other targets.

All approaches mentioned so far target a single test goal at a time – this is the predominant method. There are some notable exceptions in search-based software testing. The works of Arcuri and Yao [11] and Baresi et al. [13] use a single sequence of function calls to maximize the number of covered branches while minimizing the length of such a test case. A drawback of such an approach is that there can be conflicting testing goals, and it might be impossible to cover all of them with a single test sequence regardless of its length.

Regarding the optimization of an entire test suite in which all test cases are considered at the same time, we are aware of only the work of Baudry et al. [14]. In that work, test suites are optimized with a search algorithm with respect to mutation analysis. However, in that work there is the strong limitation of having to manually choose and fix the length of the test cases, which does not change during the search.

In the literature of testing object-oriented software, there are also techniques that do not directly aim at code coverage, as for example implemented in the Randoop [36] tool. In that work, sequences of function calls are generated incrementally using an extension of random testing (for details, see [36]), and the goal is to find test sequences for which the system under test (SUT) fails. This, however, is feasible if and only if automated oracles are available. Once a sequence of function calls is found for which at least one automated oracle is not passed, that sequence can be reduced to remove all the unnecessary function calls to trigger the failure. The software tester would usually get as output only the test cases for which failures are triggered. Notice that achieving higher coverage likely leads to higher probability of finding faults, and so recent extensions such as Palus [52] aim to achieve this.

Although targeting for path coverage, tools like DART [22] or CUTE [39] have a similar objective, assuming the availability of an automated oracle (e.g., does the SUT crash?) to check the generated test cases. This step is essential because, apart from trivial cases, the test suites generated following a path coverage criterion would be far too large to be manually evaluated by software testers in real industrial contexts.

The testing problem we address in this paper is very different from the one considered by tools such as Randoop, DART, or CUTE: Our goal is to target difficult faults for which automated oracles are not available – which is a common situation in practice. Because in these cases the outputs of the test cases have to be verified manually, the generated test suites need to be of manageable size. There are two contrasting objectives: the "quality" of the test suite (e.g., measured in its ability to trigger failures once manual oracles are provided) and its size. The approach we follow in this paper can be summarized as: Satisfy the chosen coverage criterion (e.g., branch coverage) with the smallest possible test suite.

3 Test Suite Optimization

To evolve test suites that optimize the chosen coverage criterion, we use a search algorithm, namely a Genetic Algorithm (GA), that is applied on a population of test suites. In this section, we describe the applied GA, the representation, genetic operations, and the fitness function.

3.1 Genetic Algorithms

Genetic Algorithms (GAs) qualify as meta-heuristic search technique and attempt to imitate the mechanisms of natural adaptation in computer systems. A population of chromosomes is evolved using genetics-inspired operations, where each chromosome represents a possible problem solution.

The GA employed in this paper is depicted in Algorithm 1: Starting with a random population, evolution is performed until a solution is found that fulfills the coverage criterion, or the allocated resources (e.g., time, number of fitness evaluations) have been used up. In each iteration of the evolution, a new generation is created and initialized with the best individuals of the last generation (*elitism*). Then, the new generation is filled up with individuals produced by rank selection (Line 5), crossover (Line 7), and mutation (Line 10). Either the offspring or the parents are added to the new generation, depending on fitness and length constraints (see Section 3.4).

3.2 Problem Representation

To apply search algorithms to solve an engineering problem, the first step is to define a representation of the valid solutions for that problem. In our case, a solution is a *test suite*, which is represented as a set T of test cases t_i. Given $|T| = n$, we have $T = \{t_1, t_2, \ldots, t_n\}$.

In a unit testing scenario, a test case t essentially is a program that executes the SUT. Consequently, a test case requires a reasonable subset of the target language (e.g., Java in our case) that allows one to encode optimal solutions for the addressed problem. In this paper, we use a test case representation similar to what has been used previously [21], [43]: A test case is a sequence of statements $t = \langle s_1, s_2, \ldots, s_l \rangle$

Algorithm 1 The genetic algorithm applied in EvoSuite

```
$current\_population \leftarrow$ generate random population
repeat
    $Z \leftarrow$ elite of $current\_population$
    while $|Z| \neq |current\_population|$ do
        $P_1,P_2 \leftarrow$ select two parents with rank selection
        if crossover probability then
            $O_1,O_2 \leftarrow$ crossover $P_1,P_2$
        else
            $O_1,O_2 \leftarrow P_1,P_2$
        mutate $O_1$ and $O_2$
        $f_P = min(fitness(P_1),fitness(P_2))$
        $f_O = min(fitness(O_1),fitness(O_2))$
        $l_P = length(P_1) + length(P_2)$
        $l_O = length(O_1) + length(O_2)$
        $T_B =$ best individual of $current\_population$
        if $f_O < f_P \vee (f_O = f_P \wedge l_O \leq l_P)$ then
            for $O$ in $\{O_1,O_2\}$ do
                if $length(O) \leq 2 \times length(T_B)$ then
                    $Z \leftarrow Z \cup \{O\}$
                else
                    $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
        else
            $Z \leftarrow Z \cup \{P_1,P_2\}$
    $current\_population \leftarrow Z$
until solution found or maximum resources spent
```

of length l. The length of a test suite is defined as the sum of the lengths of its test cases, i.e., $length(T) = \sum_{t \in T} l_t$. Note, that in this paper we only consider the problem of deriving test inputs. In practice, a test case usually also contains a test oracle, e.g., in terms of test assertions; the problem of deriving such oracles is addressed elsewhere (e.g., [21].

Each statement in a test case represents one value $v(s_i)$, which has a type $\tau(v(s_i)) \in \mathcal{T}$, where $\mathcal{T}$ is the finite set of types. We define five different kinds of statements:

Primitive statements represent numeric, Boolean, String, and enumeration variables, as for example `int var0 = 54`. Furthermore, primitive statements can also define arrays of any type (e.g., `Object[] var1 = new Object[10]`). The value and type of the statement are defined by the primitive variable. In addition, an array definition also implicitly defines a set of values of the component type of the array, according to the length of the array.

Constructor statements generate new instances of any given class; e.g., `Stack var2 = new Stack()`. Value and type of the statement are defined by the object constructed in the call. Any parameters of the constructor call are assigned values out of the set $\{v(s_k) \mid 0 \leq k < i\}$.

Field statements access public member variables of objects, e.g., `int var3 = var2.size`. Value and type of a field statement are defined by the member variable. If the field is non-static, then the source object of the field has to be in the set $\{v(s_k) \mid 0 \leq k < i\}$.

Method statements invoke methods on objects or call static methods, e.g., `int var4 = var2.pop()`. Again, the source object or any of the parameters have to be values in $\{v(s_k) \mid 0 \leq k < i\}$. Value and type of a method statement are defined by its return value.

Assignment statements assign values to array indices or to public member variables of objects, e.g., `var1[0] = new Object()` or `var2.maxSize = 10`. Assignment statements do not define new values.

For a given SUT, the *test cluster* [47] defines the set of available classes, their public constructors, methods, and fields.

Note that the chosen representation has *variable size*. Not only the number n of test cases in a test suite can vary during the GA search, but also the number of statements l in the test cases. The motivation for having a variable length representation is that, for a new software to test, we do not know its optimal number of test cases and their optimal length a priori – this needs to be searched for.

The entire search space of test suites is composed of all possible sets of sizes from 1 to N (i.e., $n \in [1,N]$). Each test case can have a size from 1 to L (i.e., $l \in [1,L]$). We need to have these constraints, because in the context addressed in this paper we are not assuming the presence of an automated oracle. Therefore, we cannot expect software testers to manually check the outputs (i.e., writing assert statements) of thousands of long test cases. For each position in the sequence of statements of a test case, there can be from I_{min} to I_{max} possible statements, depending on the SUT and the position (later statements can reuse objects instantiated in previous statements). The search space is hence extremely large, although finite because N, L and I_{max} are finite.

3.3 Fitness Function

In this paper, we focus on *branch coverage* as test criterion, although the EvoSuite approach can be generalized to any test criterion. A program contains control structures such as `if` or `while` statements guarded by logical predicates; branch coverage requires that each of these predicates evaluates to true and to false. A branch is *infeasible* if there exists no program input that evaluates the predicate such that this particular branch is executed.

Let B denote the set of branches of the SUT, two for every control structure. For simplicity, we treat switch/case constructs such that each case is treated like an individual if condition with a true and false branch. A method without any control structures consists of only one branch, and therefore we require that each method in the set of methods M is executed at least once.

An optimal solution T_o is defined as a solution that covers all the feasible branches/methods and it is minimal in the total number of statements, i.e., no other test suite with the same coverage should exist that has a lower total number of statements in its test cases. Depending on the chosen test case representation some branches might never be covered, even though they are potentially reachable if the entire grammar of the target language was used. As a very simple example, if the chosen representation allows only to create instances of the SUT and none of other classes, then it might not be possible to reach the branches in the methods of the SUT that take as input instances of other classes. Because without a formal proof it is not possible to state that a representation is fully adequate, for sake of simplicity we tag those branches as infeasible for the given representation.

In order to guide the selection of parents for offspring generation, we use a fitness function that rewards better coverage. If two test suites have the same coverage, the selection mechanism rewards the test suite with less statements, i.e., the shorter one.

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 M_T as well as the minimal *branch distance* $d_{min}(b,T)$ for each branch $b \in B$. 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 [30], [34]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [30], [34] 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 SUT is instrumented to keep track of the distances for each execution.

The fitness function 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:

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

Here, $\nu(x)$ is a normalizing function in $[0,1]$; we use the normalization function [4]: $\nu(x) = x/(x+1)$. Notice that there is a non-trivial reason behind the choice of $d(b,T) = \nu(d_{min}(b,T))$ applied only when the predicate is executed at least twice [11]. For example, assume the case in which it is always applied. If the predicate is reached, and branch b is not covered, then we would have $d(b,T) > 0$, while the opposite branch b_{opp} would be covered, and so $d(b_{opp},T) = 0$. The search algorithm might be able to follow the gradient given by $d(b,T) > 0$ until b is covered, i.e., $d(b,T) = 0$. However, in that case b_{opp} would not be covered any more, and so its branch distance would increase, i.e., $d(b_{opp},T) > 0$. Now, the search would have a gradient to cover b_{opp} but, if it does cover it, then necessarily b would not be covered any more (the predicate is reached only once) – and so on. Forcing a predicate to be evaluated at least twice, before assigning $\nu(d_{min}(b,T))$ to the distance of the non-covered branch, avoids this kind of circular behavior.

Finally, the resulting fitness function to minimize is as follows:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k,T)$$

3.4 Bloat Control

A variable size representation could lead to *bloat*, which is a problem that is very typical for example in Genetic Programming [41]: For instance, after each generation, test cases can become longer and longer, until all the memory is consumed, even if shorter sequences are better rewarded. Notice that bloat is an extremely complex phenomenon in evolutionary computation, and after many decades of research it is still an open problem whose dynamics and nature are not completely understood [41].

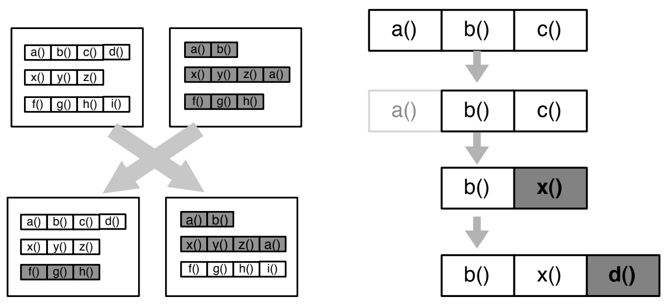
Bloat occurs when small negligible improvements in the fitness value are obtained with larger solutions. This is very typical in classification/regression problems. When in software testing the fitness function is just the obtained coverage, then we would not expect bloat, because the fitness function would assume only few possible values. However, when other metrics are introduced with large domains of possible values (e.g., branch distance and also for example mutation impact [21]), then bloat might occur.

In previous work [19], we have studied several bloat control methods from the literature of Genetic Programming [41] applied in the context of testing object-oriented software. However, our previous study [19] covered only the case of targeting one branch at a time. In EvoSuite we use the same methods analyzed in that study [19], although further analyses are required to study whether there are differences in their application to handle bloat in evolving test suites rather than single test cases. The employed bloat control methods are:

- We put a limit N on the maximum number of test cases and limit L for their maximum length. Even if we expect the length and number of test cases of an optimal test suite to have low values, we still need to choose comparatively larger N and L. In fact, allowing the search process to employ longer test sequences and then reduce their length during/after the search can provide staggering improvements in terms of achieved coverage [5].
- In our GA we use rank selection [49] based on the fitness function (i.e., the obtained coverage and branch distance values). In case of ties, we assign better ranks to smaller test suites. Notice that including the length directly in the fitness function (as for example done in [11], [13]), might have side-effects, because we would need to put together and linearly combine two values of different units of measure. Furthermore, although we have two distinct objectives, coverage is more important than size.
- Offspring with non-better coverage are never accepted for the next generation if they are larger than their parents (for the details, see Algorithm 1).
- We use a dynamic size limit conceptually similar to the one presented by Silva and Costa [41]. If an offspring's coverage is not better than that of the best solution T_B in the current entire GA population, then it is not accepted in the new generations if it is longer than twice the length of T_B (see Line 18 in Algorithm 1).

3.5 Search Operators

The GA code depicted in Algorithm 1 is at high level, and can be used for many engineering problems in which variable size representations are used. To adapt it to a specific engineering problem, we need to define search operators that manipulate



(a) Test Suite Crossover (b) Test Case Mutation

Fig. 3. Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level; mutation is applied to test cases and test suites.

the chosen solution representation (see Section 3.2). In particular we need to define the crossover and mutation operators for test suites. Furthermore, we need to define how random test cases are sampled when we initialize the first population of the GA.

3.5.1 Crossover

The crossover operator (see Figure 3(a)) generates two offspring O_1 and O_2 from two parent test suites P_1 and P_2. A random value α is chosen from $[0,1]$. On one hand, the first offspring O_1 will contain the first $\alpha|P_1|$ test cases from the first parent, followed by the last $(1-\alpha)|P_2|$ test cases from the second parent. On the other hand, the second offspring O_2 will contain the first $\alpha|P_2|$ test cases from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent.

Because the test cases are independent among them, this crossover operator always yields valid offspring test suites. Furthermore, it is easy to see that it decreases the difference in the number of test cases between the test suites, i.e., $abs(|O_1|-|O_2|) \leq abs(|P_1|-|P_2|)$. No offspring will have more test cases than the largest of its parents. However, it is possible that the total sum of the length of test cases in an offspring could increase.

3.5.2 Mutation

The mutation operator for test suites is more complicated than that used for crossover, because it works both at test suite and test case levels. When a test suite T is mutated, each of its test cases is mutated with probability $1/|T|$. So, on average, only one test case is mutated. Then, a number of new random test cases is added to T: With probability σ, a test case is added. If it is added, then a second test case is added with probability σ^2, and so on until the ith test case is not added (which happens with probability $1-\sigma^i$). Test cases are added only if the limit N has not been reached, i.e., if $n < N$.

If a test case is mutated (see Figure 3(b)), then three types of operations are applied in order: *remove*, *change* and *insert*. Each is applied with probability $1/3$. Therefore, on average, only one of them is applied, although with probability $(1/3)^3$ all of them are applied. These three operations work as follows:

Remove: For a test case $t = \langle s_1, s_2, \ldots, s_l \rangle$ with length l, each statement s_i is deleted with probability $1/l$. As the value $v(s_i)$ might be used as a parameter in any of the statements $s_{i+1}, \ldots, s_l$, the test case needs to be repaired to remain valid: For each statement s_j, $i < j \leq l$, if s_j refers to $v(s_i)$, then this reference is replaced with another value out of the set $\{v(s_k) \mid 0 \leq k < j \wedge k \neq i\}$ which has the same type as $v(s_i)$. If this is not possible, then s_j is deleted as well recursively.

Change: For a test case $t = \langle s_1, s_2, \ldots, s_l \rangle$ with length l, each statement s_i is changed with probability $1/l$. If s_i is a primitive statement, then the numeric value represented by s_i is changed by a random value in $[-\Delta, \Delta]$, where Δ is a constant. If the primitive value is a string, then the string is changed by deleting, replacing, or inserting characters in a way similar to how sequences of method calls are mutated. In the case of an array, the length is changed by a random value in $[-\Delta', \Delta']$ such that no accesses to the array are invalidated. In an assignment statement, either the variable on the left or the right hand side of the assignment is replaced with a different variable of the same type. If s_i is not a primitive statement, then a method, field, or constructor with the same return type as $v(s_i)$ and parameters satisfiable with the values in the set $\{v(s_k) \mid 0 \leq k < i\}$ is randomly chosen out of the test cluster.

Insert: With probability σ', a new statement is inserted at a random position in the test case. If it is added, then a second statement is added with probability σ'^2, and so on until the ith statement is not inserted. A new statement is added only if the limit L has not been reached, i.e., if $l < L$. For each insertion, with probability $1/3$ a random call of the class under test or its member classes is inserted, with probability $1/3$ a method call on a value in the set $\{v(s_k) \mid 0 \leq k < i\}$ for insertion at position i is added, and with probability $1/3$ a value $\{v(s_k) \mid 0 \leq k < i\}$ is used as a parameter in a call of the class under test or its member classes. Any parameters of the selected call are either reused out of the set $\{v(s_k) \mid 0 \leq k < i\}$, set to `null`, or randomly generated.

If after applying these mutation operators a test case t has no statements left (i.e., all have been removed), then t is removed from T.

To evaluate the fitness of a test suite, it is necessary to execute all its test cases and collect the branch information. During the search, on average only one test case is changed in a test suite for each generation. This means that re-executing all test cases is not necessary, as the coverage information can be carried over from the previous execution.

3.5.3 Random Test Cases

Random test cases are needed to initialize the first generation of the GA, and when mutating test suites. Sampling a test case at random means that each possible test case in the search space has a non-zero probability of being sampled, and these probabilities are independent. In other words, the probability of sampling a specific test case is constant and it does not depend on the test cases sampled so far.

When a test case representation is complex and it is of variable length (as it happens in our case, see Section 3.2), it is often not possible to sample test cases with uniform distribution (i.e., each test case having the same probability of being sampled). Even when it would be possible to use a uniform distribution, it would be unwise (for more details on this problem, see [10]). For example, given a maximum length L, if each test case was sampled with uniform probability, then sampling a short sequence would be extremely unlikely. This is because there are many more test cases with long length compared to the ones of short length.

In this paper, when we sample a test case at random, we choose a value r in $1 \leq r \leq L$ with uniform probability. Then, on an empty sequence we repeatedly apply the insertion operator described in Section 3.5.2 until the test case has a length $\geq r$.

4 The EvoSuite Tool

The EvoSuite tool implements the approach presented in this paper for generating JUnit test suites for Java code. EvoSuite works on the byte-code level and collects all necessary information for the test cluster from the byte-code via Java Reflection. This means that it does not require the source code of the SUT, and in principle is also applicable to other languages that compile to Java byte-code (such as Scala or Groovy, for example). Note that we also consider branch coverage at the byte-code level. Because high level branch statements in Java (e.g., predicates in loop conditions) are transformed into simpler statements similar to atomic `if` statements in the byte-code, EvoSuite is able to handle all language constructs. Furthermore, EvoSuite treats each case of a switch/case construct like an individual `if`-condition. The number of branches at byte-code level is thus usually larger than at source code level, as complex predicates are compiled into simpler byte-code instructions.

EvoSuite instruments the byte-code with additional statements to collect the information necessary to calculate fitness values, and also performs some basic transformations to improve testability: To allow optimizations of String values, branches based on String methods like `String.equals` are transformed such that they act on the edit distance [2]. Similarly, comparisons on double, float, and long datatypes in byte-code need transformation to carry a distance measurements to the branches.

During test generation, EvoSuite considers one top-level class at a time. The class and all its anonymous and member classes are instrumented at byte-code level to keep track of called methods and branch distances during execution. To produce test cases as compilable JUnit source code, EvoSuite accesses only the public interfaces for test generation; any subclasses are also considered part of the unit under test to allow testing of abstract classes. To execute the tests during the search, EvoSuite uses Java Reflection. Before presenting the result to the user, test suites are minimized using a simple minimization algorithm [5] which attempts to remove each statement one at a time until all remaining statements contribute to the coverage; this minimization reduces both the number of test cases as well as their length, such that removing any statement in the resulting test suite will reduce its coverage.

The search operators for test cases make use of only the type information in the test cluster, and although type information is updated during execution difficulties can arise when method signatures are imprecise. In particular, this problem exists for all classes using Java Generics, as type erasure removes much of the useful information during compilation and all generic parameters look like `Object` for Java Reflection. To overcome this problem for container classes (and in general for any class with methods that take as input and return `Object` instances), we always put `Integer` objects into container classes, such that we can also cast returned `Object` instances back to `Integer`. Currently, container classes need to be identified manually, but future versions of EvoSuite will determine suitable types automatically.

Test case execution can be slow, and in particular when generating test cases randomly, infinite recursion can occur (e.g., by adding a container to itself and then calling the `hashCode` method). Therefore, we chose a timeout of five seconds for test case execution. If a test case times out, then the test suite with that test case is assigned the maximum fitness value, which is $|M| + |B|$, the sum of methods and branches to be covered.

Test cases are executed in their own threads, but as Java does not allow to forcefully stop threads it may happen that after the timeout such a thread survives. To overcome this problem, we instrument the byte-code with additional statements that yield the execution of a test when the thread is interrupted. If this also shows no effect, the thread is given a low priority and its execution is ignored. Aggregations of such stale threads consume memory, and so EvoSuite uses a client/server architecture where the client monitors its free memory, and asks the server for a process restart which resumes at the same point in the search, if it runs out of memory.

EvoSuite also employs a custom, configurable security manager which can be used to control what permissions are granted during the test execution. For example, it is usually not desirable to have the tests open random networking connections or access the filesystem in random ways. The use of a custom security manager was an essential feature for the large empirical study we conduct in this paper. In fact, thanks to that we were able to select a large number of SUTs from different sources without needing to worry whether their behavior was safe under all inputs. However, this means that EvoSuite currently can only cover code that has no environmental dependencies. Furthermore, EvoSuite assumes that the code under test is deterministic, such that executing the same test case twice will yield the same results.

4.1 Single Branch Strategy

To allow a fair comparison with the traditional single branch approach (e.g., [43]), we implemented this strategy on top of EvoSuite. In the single branch strategy, an individual of the search space is a single test case. The identical mutation operators for test cases can be used as in EvoSuite, but crossover needs to be done at the test case level. For this, we used the approach also applied by Tonella [43] and Fraser and Zeller [21]: Offspring is generated using the single point crossover function described in Section 3.5.1, where the first part of the sequence of statements of the first parent is merged with the second part of the second parent, and vice versa.

Because there are dependencies between statements and values generated in the test case, this might invalidate the resulting test case, and we need to repair it: The statements of the second part are appended one at a time similarly to the insertion described in Section 3.5.2, except that whenever possible dependencies are satisfied using existing values.

The fitness function in the single branch strategy also needs to be adapted: We use the traditional *approach level* [48] plus normalized branch distance fitness function, which is commonly used in the literature (e.g., see [26], [34]). The approach level is used to guide the search toward the target branch. It is determined as the minimal number of control dependent edges in the control dependency graph between the target branch and the control flow represented by the test case. The branch distance is calculated as in EvoSuite, but taken for the closest control dependent branch where the control flow diverges from the target branch.

While implementing this approach, we tried to derive a faithful representation of current practice, which means that there are some optimizations proposed in the literature which we did not include:

- New test cases are only generated for branches that have not already been covered through collateral coverage of previously created test cases. However, we do not evaluate the collateral coverage of all individuals during the search, as this would add a significant overhead, and it is not clear what effects this would have given the fixed timeout we used in our experiments.
- When applying the one target at a time approach, a possible improvement could be to use a *seeding* strategy [48]. During the search, we could store the test data that have good fitness values on targets that are not covered yet. These test data can then be used as starting point (i.e., for seeding the first generation of a GA) in the successive searches for those uncovered targets. However, we decided not to implement this, as reference [48] does not provide sufficient details to reimplement the technique, and there is no conclusive data regarding several open questions; for example, potentially a seeding strategy could reduce diversity in the population, and so in some cases it might in fact reduce the overall performance of the search algorithm.
- The order in which coverage goals are selected might also influence the result. As in the literature usually no order is specified (e.g., [25], [27], [43]), we selected the branches in random order. However, in the context of procedural code approaches to prioritize coverage goals have been proposed, e.g., based on dynamic information [48]. However, the goal of this paper is neither to study the impact of different orders, nor to adapt these prioritization techniques to object-oriented code and prioritization techniques.
- In practice, when applying a single goal strategy, one might also bootstrap an initial random test suite to identify the trivial test goals, and then use a more sophisticated technique to address the difficult goals; here, a difficult question is when to stop the random phase and start the search. In contrast, EvoSuite has this initial random phase integrated into its own search process.

TABLE 1
Number of classes, branches, and lines of code in the case study subjects

Case Study		#Classes Public	#Classes All	#Branches	LOC[1]
COL	Colt	135	298	10,795	20,741
CCL	Commons CLI	14	15	662	1,078
CCD	Commons Codec	21	22	1,369	2,205
CCO	Commons Collections	246	421	8,683	19,190
CMA	Commons Math	247	306	10,503	23,881
CPR	Commons Primitives	210	231	2,874	7,008
GCO	Google Collections	85	370	4,214	9,886
ICS	Industrial Casestudy	21	29	373	809
JCO	Java Collections	30	118	3,531	6,339
JDO	JDom	57	61	4,098	6,452
JGR	JGraphT	137	193	2,467	5,924
JTI	Joda Time	131	199	8,681	18,003
NXM	NanoXML	1	1	310	661
NCS	Numerical Casestudy	11	11	209	421
REG	Java Regular Expressions	3	91	1,922	3,020
SCS	String Casestudy	12	12	607	606
TRO	GNU Trove	205	591	10,585	24,297
XEN	Xmlenc	7	7	1,645	788
XOM	XML Object Model	165	185	11,794	23,814
ZIP	Java ZIP Utils	3	4	219	441
Σ		1,741	3,165	85,541	175,564

5 Experiments

The independence of the order in which test cases are selected and the collateral coverage are inherent to the EvoSuite approach, therefore the evaluation focuses on the improvement over the single branch strategy.

5.1 Case Study Subjects

For the evaluation, we chose a total of 19 open source libraries and programs. For example, among those there are several widely used libraries developed by Google and the Apache Software Foundation. Furthermore, to analyze in more details some specific types of software, we also used a translation of the String case study subjects employed by Alshraideh and Bottaci [2], and we also used a set of numerical applications from [6]. To avoid a bias caused by considering only open source code, we also selected a subset of an industrial case study project previously used by Arcuri et al. [9]. This results in a total of 3,165 classes, which were tested by only calling the API of the 1,741 public classes (the remaining classes are member classes).

The choice of a case study is of paramount importance for any empirical analysis in software engineering. To address this problem, in this paper we consider several types of software, as for example container classes, numerical applications, and software with high use of strings and arrays processing. Table 1 summarizes the properties of these case study subjects.

To avoid bias in analyzing the results, we present and discuss the results of our empirical study grouped by project. In fact, different testing techniques can have comparatively different performance on different types of SUT. For example,

1. LOC stands for non-commenting lines of source code, calculated with JavaNCSS (http://javancss.codehaus.org/)

random testing has been shown to be very effective in testing container classes [40]. If one only chooses container classes as case study and ignores for example numerical applications, then random testing could be misleadingly advantaged in technique comparisons. In fact, even if one uses several kinds of software as SUTs, then it all depends on the proportion of the SUT types (e.g., if in the case study there are many more container classes than numerical applications). Therefore, aggregated statistics on all the artifacts of a case study need to be interpreted with care, as the proportion of different kinds of software types could lead to misleading results. Unfortunately, how to define a representative case study for test data generation is still an open research question.

5.2 Experimental Setup

As witnessed in Section 3, search algorithms are influenced by a great number of parameters. For many of these parameters there are "best practices": For example, we chose a crossover probability of $3/4$ based on past experience. In EvoSuite, the probabilities for mutation are largely determined by the individual test suite size or test case length; the initial probability for test case insertion was set to $\sigma = 0.1$, and the initial probability for statement insertion was set to $\sigma' = 0.5$. The maximum value for the perturbation of integral primitive types Δ was set to 20. Although longer test cases are better in general [5], we limited the length of test cases to L = 80 because we experienced this to be a suitable length at which the test case execution does not take too long. The maximum test suite size was set to N = 100, although the initial test suites are generated with only two test cases each. The population size for the GA was chosen to be 80.

Different settings (e.g., population size) of an algorithm would lead to different performance. Unfortunately, on a new search problem (such as generation of test suites in this paper) it is not possible to know beforehand which are the best settings to use. A tuning phase could lead to find settings for which EvoSuite performs better, but tuning phases are computational expensive. As even with common settings based on the literature it is possible to achieve reasonable results [8], we postponed the tuning investigation to future work, and preferred to focus on having a larger case study with the aim of reducing threats to external validity.

Search algorithms are often compared in terms of the number of fitness evaluations; in our case, comparing to a single branch strategy would not be fair, as each individual in EvoSuite represents several test cases, such that the comparison would favor EvoSuite. As the length of test cases can vary greatly and longer test cases generally have higher coverage, we decided to take the number of *executed statements* as execution limit. This means that the search is performed until either a solution with 100% branch coverage is found, or k statements have been executed as part of the fitness evaluations. In our experiments, we chose $k = 1{,}000{,}000$.

For the single branch strategy, the maximum test case length, population size, and any probabilities are chosen identical to the settings of EvoSuite. At the end of the test generation, the resulting test suite is minimized in the same way as in EvoSuite.

The stopping condition for the single branch strategy is chosen the same as for EvoSuite, i.e., maximum 1,000,000 statements executed. To avoid that this budget is spent entirely on the first branch if it is difficult or infeasible, we apply the following strategy: For $|B|$ branches and an initial budget of X statements, the execution limit for each branch is $X/|B|$ statements. If a branch is covered, some budget may be left over, and so after the first iteration on all branches there is a remaining budget X'. For the remaining uncovered branches B' a new budget $X'/|B'|$ is calculated and a new iteration is started on these branches. This process is continued until the maximum number of statements (1,000,000) is reached.

EvoSuite and search-based testing are based on randomized algorithms, which are affected by chance. Running a randomized algorithm twice will likely produce different results. It is essential to use rigorous statistical methods to properly analyze the performance of randomized algorithms when we need to compare two or more of them. In this paper, we follow the guidelines described in [7].

For each of the 1,741 public classes, we ran EvoSuite against the single branch strategy to compare their achieved coverage. Each experiment comparison was repeated 30 times with different seeds for the random number generator. When one can have an arbitrarily large number of artifacts for a case study (e.g., in our case we could download and use as SUT as many projects as we wanted from open source repositories, because EvoSuite is fully automated and has a customized security manager to avoid undesired side effects), then there is the trade-off between the number of runs per artifact and size of the case study. In general, it is advisable to have enough runs (e.g., 30) to detect statistical difference of algorithm performances on single artifacts, and then have a case study as large and variegated as possible to reduce threats to external validity. Notice that, in our empirical analysis, even with the use of a large cluster of computers it took several days to run all the experiments.

5.3 Results

Due to space constraints we cannot provide full information of the analyzed data [7], but just show the data that are sufficient in claiming the superiority of the EvoSuite technique. Statistical difference has been measured with the Mann-Whitney U test. To quantify the improvement in a standardized way, we used the Vargha-Delaney $\hat{A}_{12}$ effect size [44]. In our context, the $\hat{A}_{12}$ is an estimation of the probability that, if we run EvoSuite, we will obtain better coverage than running the single branch strategy. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that, in *all* of the 30 runs of EvoSuite, we obtained coverage values higher than the ones obtained in *all* of the 30 runs of the single branch strategy.

The box-plot in Figure 4 compares the actual obtained coverage values (averaged out of the 30 runs) of EvoSuite and the single branch strategy (Single). In total, the coverage improvement of EvoSuite ranged up to 188 times that of the single branch strategy. This happened for a particular SUT in which EvoSuite obtained a coverage of 70.2% (averaged

TABLE 2
$\hat{A}_{12}$ measure values in the coverage comparisons: $\hat{A}_{12} < 0.5$ means EVOSUITE resulted in less, $\hat{A}_{12} = 0.5$ equal, and $\hat{A}_{12} > 0.5$ better coverage than a single branch approach. In brackets "()" the number of times the effect size is statistically significant at level 0.05.

Case Study	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$
COL	13(9)	30	92(79)
CCL	2(1)	6	6(4)
CCD	2(1)	13	6(5)
CCO	19(5)	137	90(81)
CMA	24(10)	100	123(103)
CPR	23(10)	150	37(19)
GCO	4(2)	31	50(42)
ICS	0(0)	17	4(3)
JCO	2(1)	10	18(17)
JDO	3(2)	27	27(25)
JGR	2(1)	88	47(41)
JTI	41(28)	28	62(41)
NXM	0(0)	0	1(1)
NCS	1(0)	10	0(0)
REG	0(0)	1	2(2)
SCS	4(4)	6	2(1)
TRO	2(1)	73	130(124)
XEN	0(0)	4	3(3)
XOM	22(11)	92	51(45)
ZIP	0(0)	1	2(2)
Σ	164(86)	824	753(638)

over 30 runs), while for the single branch strategy the average coverage was only 0.3%. Calculated on all the SUTs in the case study, EVOSUITE obtained an average coverage of 83%, whereas the single branch strategy obtained 77%.

Figure 5 shows a box-plot of the results of the $\hat{A}_{12} \neq 0.5$ measure for the coverage grouped by case study subject; this figure illustrates the strong statistical evidence that EVOSUITE achieves higher coverage. In many cases, EVOSUITE is practically certain to achieve better coverage results, even when we take the randomness of the results into account.

Whole test suite generation achieves **higher coverage** *than single branch test case generation.*

Table 2 shows for the coverage comparisons how many times we obtained $\hat{A}_{12}$ values equal, lower and higher than 0.5. We obtained p-values lower than 0.05 in 724 out of 917 comparisons in which $\hat{A}_{12} \neq 0.5$. Among the 20 projects, there is one for which EVOSUITE gave worse results, i.e., SCS. Without an in depth analysis on that project, it is difficult to conjecture why that is the case. The project SCS contains a set of *artificial* classes, where all methods are static, take as input (and manipulate) string objects, and have no infeasible branches. It might be that using a hybrid algorithm in which EVOSUITE is enhanced with local search (e.g., see [26]) could be effective for this type of software. But the use of local search for test suite evolution is still an unexplored field, although promising (see for example [11], [13], [26]). At any rate, although the single branch strategy is statistically better on four SUTs and achieves an average coverage of 86.8%, even if EVOSUITE is statistically better on only one SUT, its average coverage is actually higher, i.e., 87.7%. This is not a contradiction. It could be explained by the fact that, when EVOSUITE is worse on a specific SUT, it is only worse by little, whereas when it is better, it is better by a larger quantity. This is clearly visible in the boxplot in Figure 4, where although the median value for EVOSUITE is lower, then however its lower quartile and minimum value is much higher.

Notice that in many cases (i.e., 824) we have $\hat{A}_{12} = 0.5$. This did not come as a surprise: For some "easy" classes, a budget of 1,000,000 statements executions would be more than enough to cover all the feasible branches with very high probability. In these cases, it is important to analyze what is the resulting size of the generated test suites. When the coverage is different, analyzing the resulting test suite sizes is not reasonable (e.g., a test suite with higher coverage likely has to be larger).

For those cases where $\hat{A}_{12} = 0.5$ for the coverage, Figure 6 compares the obtained test suite size values (averaged out of the 30 runs) of EVOSUITE and the single branch strategy (Single). In the best case, we obtained a test suite size (averaged out of the 30 runs) that for EVOSUITE was 62% smaller. In particular, EVOSUITE has an average length of 3.23 statements, whereas the length of the test suites generated with the single branch strategy led to an average 8.36 length.

Figure 7 shows $\hat{A}_{12}$ for the length of the resulting test suites, but only when p-values are lower than 0.05. Recall that for both EVOSUITE and the single branch strategy we use the *same* post-processing technique to reduce the length of the output test suites. When we obtain full coverage of all the feasible branches, then EVOSUITE has a low probability of generating larger test suites.

Whole test suite generation produces **smaller test suites** *than single branch test case generation.*

The results obtained with EVOSUITE compared to the traditional approach (of targeting each branch separately) are simply staggering. How is it possible to achieve such large improvements? There can be several explanations. First, in case of infeasible branches, all the effort spent by a single branch at a time strategy would be wasted, apart from possible collateral coverage. Collateral coverage of difficult to reach branches, however, would be quite unlikely. Second, the traditional fitness function would have problems in guiding the search toward private methods that are difficult to execute. For example, consider the case of a private method that is called only once in a public method, but that method call is nested in a branch whose predicate is very complicated to satisfy. Unless the fitness function is extended to consider all possible methods that can lead to execute the private methods (as for example done in [46]), then there would be no guidance to execute those private methods. Third, assume that there is a difficult branch to cover, and nested to that branch there are several others. Once EVOSUITE is able to generate a test sequence that covers that difficult branch, this sequence can be extended (e.g., by adding function calls at its end) or copied in another test case in the test suite (e.g., through the crossover operator) to make it easier to cover the other nested branches.

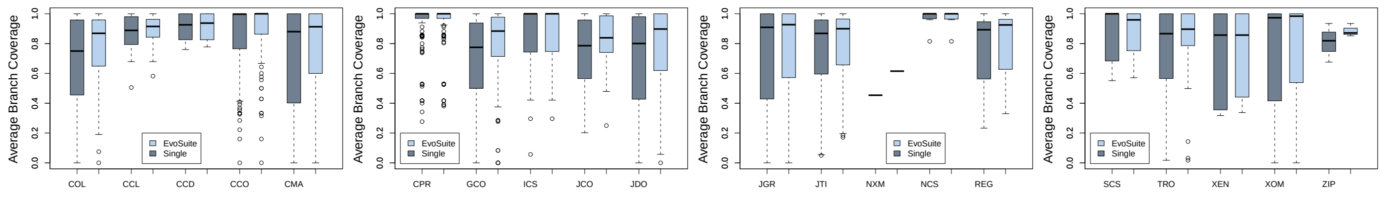


Fig. 4. Average branch coverage: Even with an evolution limit of 1,000,000 statements, EvoSuite achieves higher coverage.

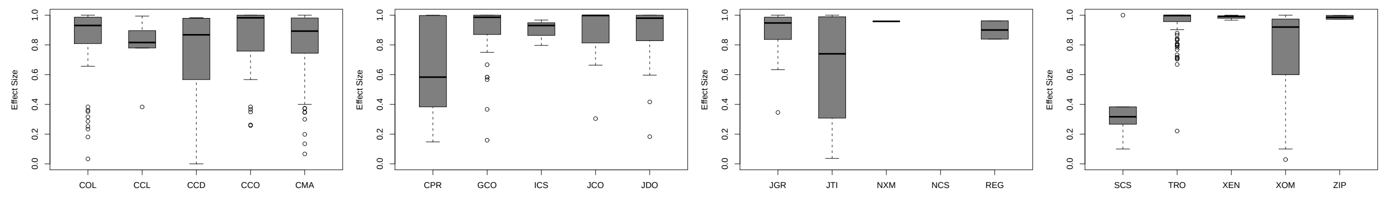


Fig. 5. $\hat{A}_{12}$ for coverage: EvoSuite has a high probability of achieving higher coverage than a single branch approach.

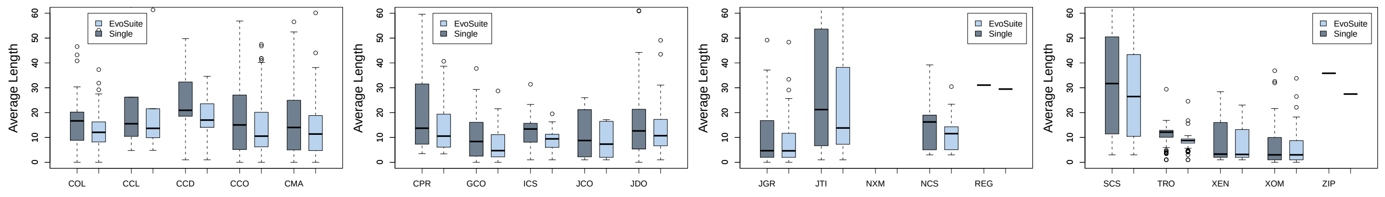


Fig. 6. Average test suite length: Even after minimization, EvoSuite test suites tend to be smaller than those created with a single branch strategy (shown for cases with identical coverage).

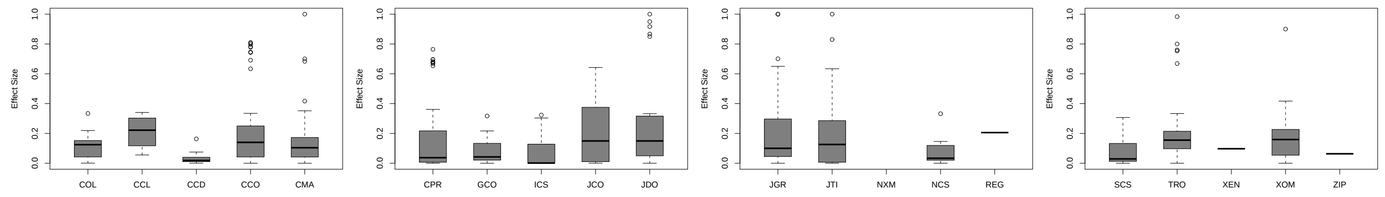


Fig. 7. $\hat{A}_{12}$ for test suite length: EvoSuite has a low probability of generating longer test suites than a single branch approach.

On the other hand, in the traditional approach of targeting one branch at a time, unless smart seeding strategies are used based on previously covered branches, the search to cover nested branches would be harmed by the fact that covering their difficult parent needs to be done from scratch again.

Because our empirical analysis employs a very large case study (1,741 classes for a total of 85,503 byte-code level branches), we cannot analyze all of these branches to give an exact explanation for the better performance of EvoSuite. However, the three possible explanations we gave are plausible, although further analyses (e.g., on artificial software that we can generate with known number of infeasible branches) would be required to shed light on this important research question. To this goal, we created an artificial problem to experiment with, whose analysis is discussed later in the paper.

5.3.1 *Difficult Branches*

Looking at the results in Table 2 we see that EvoSuite does not achieve 100% coverage for all classes. To some extent, this is due to infeasible branches. Due to the large amount of classes empirically investigated in this paper, it is not possible to distinguish between all infeasible and difficult branches by hand. However, identifying some of the difficult cases is helpful to understand the results and guide future research.

Not all infeasible branches are as obvious as the example in Figure 1. Other examples of infeasible branches are given by private methods that are not called in any public methods, dead code, or methods of abstract classes that are overridden in all concrete subclasses without calling the abstract super-class.

Some difficult branches we could identify are the following:

Environment dependencies such as databases or the filesystem are not currently handled by EVOSUITE. Using the custom security manager one can avoid that interactions with the environment cause damage, but branches depending, for example, on network connections or file contents usually cannot be covered.

Methods called by native code such as `readObject` and `writeObject` of the Java `Serializable` interface cannot be directly called.

Anonymous and private classes are more difficult to cover than top level classes, as they can only be handled indirectly via the owner classes' public interface. An even more difficult variant of this problem are abstract classes that are only instantiated by anonymous classes – there is no way to directly test the abstract class, except by creating stubs.

Multi-threaded code is also difficult for a search-based approach, as the test generation would need to handle thread creation and termination.

Static and private constructors are only executed once after a class is loaded, and private constructors are often used in singleton instances where the constructor is again only called once. To achieve full coverage in such cases one would need to unload classes after each test execution; EVOSUITE uses a faster but less precise approach introduced in the JCrasher [15] tool, where static constructors are duplicated in callable methods, and then re-executed before test execution.

5.4 Infeasible Test Goals

A coverage goal is infeasible if there exists no test that would exercise it. For some simple cases, there could be techniques that are able to identify infeasible targets; for example, dead-code detection might reveal some infeasible branches, such as the one listed in Figure 1. However, in general it is an undecidable problem whether a particular coverage goal or program path is feasible [23] or if a mutant is equivalent. Furthermore, a branch might be infeasible for other reasons (e.g., environmental dependencies), and applying techniques to detect infeasible goals on several targets might have a non negligible computational overhead, which would reduce the time budget for test data generation (e.g., less generations in an evolutionary algorithm). Depending on how effective these techniques are, their use might thus either increase or even decrease the overall performance of the testing tool.

When targeting each coverage goal individually, any resources invested for an infeasible goal are per definition wasted. Depending on how the available resources are distributed across the individual goals, the more infeasible goals there are the fewer of the resources will be spent on the feasible goals, thus leading to overall worse results.

In contrast, EVOSUITE does not focus on individual coverage goals, and so the infeasible goals have no effect on the achieved coverage of the feasible goals. To demonstrate this effect, we ran a set of experiments on the example code listed in Figure 8. In the byte-code representation, this code contains six feasible branches. At the position labelled in the source code we iteratively inserted infeasible branches (`if(x * x == y * y + 2)`) and analyzed the behavior of EVOSUITE and the single branch strategy over the number of inserted

```
class Infeasible {
  void infeasibleGoals(int x, int y) {
    if(x > 0 && y > 0 && 2 * x == Math.sqrt(y)) {
    }
    // Infeasible branches are added here
  }
}
```

Fig. 8. A simple example to demonstrate the effect of infeasible coverage goals. In the default case, there are three predicates, leading to six (feasible) branches in the compiled byte-code.

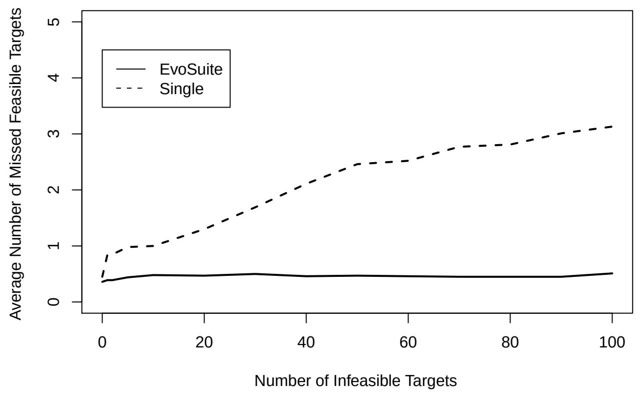


Fig. 9. The effect of infeasible goals on the coverage of the feasible goals in the code: The number of unsatisfied coverage goals rises with the number of infeasible test goals in the single branch strategy, while EVOSUITE is not influenced at all by the infeasible goals.

infeasible branches. For each version of the example code, we performed 100 runs of EVOSUITE and 100 runs of the single branch strategy with different random seeds and a search limit of 40,000 statements. Figure 9 impressively demonstrates how EVOSUITE is oblivious to the introduced infeasible branches, while the performance of the single branch strategy degrades.

To give more soundness to this analysis, we also applied a Kruskal-Wallis test to verify the impact of the number of infeasible targets on the number of missed (i.e., uncovered) feasible branches. For the single branch strategy, we obtained a p-value lower than 2.2×10^{-16}, with $\chi^2 = 932.5$. This gives strong statistical evidence that confirms the trend in Figure 9. On the other hand, for EVOSUITE we obtained a very high p-value, i.e. 0.73, where $\chi^2 = 9.44$. Notice that a high p-value does not mean that we have strong statistical evidence to claim that infeasible targets have no impact on EVOSUITE. To make analyses as precise and sound as possible, one should quantify the effect sizes and then use power analysis to calculate the probability of Type II error [7]. However, the differences in coverage in Figure 9 look so small that, even if higher number of runs *might* decrease the p-value of the test, then anyway the differences would be so small to be of little practical interest.

5.5 Comparisons with Other Tools

In this paper, we have carried out a large empirical analysis to show that the EvoSuite strategy of evolving whole test suites is generally better than the traditional approach of searching for only one target at the time. To provide further evidence on the effectiveness of EvoSuite, and to get more insight on the dynamics of test data generation, it would be important to compare its performance against other tool prototypes in the literature. Unfortunately, this was not possible. In this section, we describe the challenges and shortcomings that would be faced in tool comparisons.

First, because our tool handles Java byte-code, we could only compare it with others that handle languages also compiling to Java byte-code. For example, this precludes (or it makes them hard) comparisons with testing tools supporting C (e.g., CUTE [39]) and C# (e.g., PEX [42]). Similarly, tool prototypes are often targeted for specific operating systems; e.g., PEX [42] and Dsc [28] only work with Windows.

Second, although there are popular testing tools for Java, as for example Randoop [36], those do not address our testing problem (i.e., generating high coverage test suites that are small, so non-automated oracles can be manually verified by the software engineers). Third, some old tool prototypes are no longer supported, and can give problems when used on new versions of Java and/or SUTs with specific features (e.g., we did not manage to run jCUTE on several of the SUTs we experimented with). Fourth, some tool prototypes are simply not publicly available, and re-implementing them would be too time consuming and prone to errors and misunderstanding in the implementation.

Another important point is that many testing tools are only semi-automatic and, for example, require the user to writing drivers and ad hoc generators for specific type of objects. This is a kind of problem we faced when we tried to compare EvoSuite with tools such as JPF [45] and TestFul [13], and which makes large empirical studies difficult. Another problem for empirical studies on testing is that the tools need to guarantee that the test code does not break anything, e.g., by running it in a sandbox like EvoSuite– this is usually not the case (e.g. the Randoop documentation[1] states: "WARNING: Testing code that modifies your file system might result in Randoop generating tests that modify your file system! Be careful when choosing classes and methods to test".).

Often research prototypes have known limitations. For example, Dsc [28] clearly indicates in its documentation[2] that it does not support floating-point numbers and has only basic support for strings. Such limitations would put these prototypes in disadvantage in tool comparisons, although they might feature novel algorithms that are very useful for the type of program constructs (e.g., constraints on integer variables) that they can handle.

To the best of our knowledge, we have not found any other test data generation tool in the literature that satisfies all the above requirements and that we could use for comparisons with EvoSuite. There are however commercial tools that are likely to satisfy those constraints, as for example the ones developed by companies such as Parasoft[3] and Agitar[4]. Unfortunately, comparisons with commercial tools have their own set of challenges. For example, usually the details of the underlying technologies of commercial tools are not disclosed. Therefore, it would be hard to understand why they behave in a particular way on some SUTs, and so explaining differences in results would be infeasible.

6 Convergence

Search algorithms can be run for any arbitrary amount of time (or fitness evaluations, depending on the chosen stopping criterion). The more time it is allowed for the search, the better results we would expect on average. If given enough time, will a GA find an optimal solution? Or is there the possibility that it will be stuck forever in a sub-optimal area of search space? The answer to these questions lies in the *convergence* analysis of search algorithms [38].

Standard evolutionary algorithms using elitism are proven to converge to optimal solutions [38]. This means that, if left running for an infinite time, they will *always* find an optimal solution. Formally, given $\mathcal{P}_o(i)$ the probability of finding an optimal solution within i steps of the search, we would have $\lim_{i\to\infty} \mathcal{P}_o(i) = 1$. Convergence in infinite time might be of little practical interest, as it is not feasible to run a search algorithm for infinite time. Furthermore, given the bounds N and L, an exhaustive enumeration of the entire search space would also find an optimal solution in finite time. However, there is a major motivation for proving convergence: Convergence should be a pre-requisite of any algorithm – if an algorithm does not guarantee an optimal solution even if run for infinite time, then its use is questionable.

To use search algorithms for testing object-oriented software, new search operators are often designed in the literature, because the standard ones based on bit-strings cannot be used (e.g., [43]). But once non-standard operators are used, the theoretical results coming from the literature of evolutionary computation cannot be applied. If new search operators are designed, it is hence important to prove the convergence of the resulting new algorithms, and we do that for the GA we use in this paper (the results can be directly extended to all the other search algorithms that use the same mutation operator described in Section 3.5.2). To the best of our knowledge, this is the first time this type of theoretical results is provided for the problem of test data generation.

Let us define the number of "iterations" of a search algorithm as the number of fitness evaluations that are computed (this is a typical procedure in the formal analysis of search algorithms). The *runtime* R of a search algorithm is a random variable describing the number of iterations it needs to find an optimal solution. Notice that the notation $\mathcal{P}_o(i)$ is equivalent to $\mathcal{P}(R \leq i)$. Let us assume that, before a solution is evaluated, it is always mutated with an operator μ. It does not matter if other search operators are applied before μ (as for example

1. http://randoop.googlecode.com/hg/doc/index.html, accessed August 2011.

2. http://ranger.uta.edu/∼csallner/dsc/index.html, accessed August 2011.

3. http://www.parasoft.com, accessed August 2011.

4. http://www.agitar.com, accessed August 2011.

a crossover). The best solution seen so far is always stored (i.e., elitism). Let us call $\mathcal{A}$ the class of algorithms for which all the above conditions hold. Notice that the GA used in this paper (depicted in Algorithm 1) belongs to the class $\mathcal{A}$. The following lemma provides *sufficient* conditions for which a search algorithm in $\mathcal{A}$ converges. This lemma is a simplification and adaptation of similar theoretical results in the literature of search algorithms [38].

Lemma *1:* For the class of search algorithms $\mathcal{A}$, if μ is able to sample an optimal solution with probability p lower bounded by a constant (i.e., $p \geq c$ for some constant c), then the search algorithms using μ converge, i.e., $\lim_{i\to\infty} \mathcal{P}(R \leq i) = 1$, where $\mathcal{P}(R \leq i)$ is the probability of finding an optimal solution within i iterations. Furthermore, $E[R] \leq 1/c$, i.e., the expected number of iterations is upper bounded by the constant $1/c$.

Proof: Because we are interested in the convergence for $i \to \infty$, we do not need to study the exact dynamics of the search algorithms. We just provide loose lower bounds to the effectiveness that are high enough to prove this theorem. We can focus only on the mutation operator μ and in its ability of sampling an optimal solution with probability $p \geq c$. The process of sampling an optimal solution with μ at iteration i can be described as a geometric distribution with parameter greater or equal than c (see [16]). Therefore:

$$\lim_{i\to\infty} \mathcal{P}(R \leq i) \geq \lim_{i\to\infty} 1 - (1-c)^i = 1 \ .$$

The expected value E of a geometric distribution with parameter p is equal to $1/p$ (see [16]). Therefore, $E[R] \leq 1/c$.

Because the algorithms in $\mathcal{A}$ always store the best solution seen so far, the convergence is hence proven. □

To prove that our GA converges, it is sufficient to prove that our mutation operator described in Section 3.5.2 is always able to sample an optimal solution with probability lower bounded by a constant.

Proposition *1:* The mutation operator described in Section 3.5.2 is able to sample an optimal solution T_o with probability $p \geq c$, for some constant c, when applied to any test suite T whose number of test cases and their length is bounded in N and L, respectively.

Proof: To prove this proposition, we do not need to provide tight bounds (i.e., the highest possible constant c). Very loose bounds will be sufficient. It will be enough to prove that with constant probability we can remove and add any number of test cases in T.

An optimal solution T_o is composed of $n_o \leq N$ test cases, each one with length up to L. With probability that is at least $(1/N)^N$, all the test cases are mutated. With probability $(1/3)\cdot(2/3)^2$, only the remove operation is applied on a test case. With probability at least $(1/L)^L$, all the statements in a test case are removed. Therefore, all the test cases are removed with probability *at least*:

$$\rho = \left(\frac{1}{N}\frac{2^2}{3^3}\frac{1}{L^L}\right)^N \ .$$

After removing all the test cases in T, exactly n_o new random test cases will be added with probability $\psi = (1 - \sigma^{n_o+1}) \times \prod_{j=1}^{n_o} \sigma^j = (1-\sigma^{n_o+1}) \times \sigma^{n_o(n_o+1)/2}$ (see Section 3.5.2). In fact we need the first insertion with probability σ^1, then second with probability σ^2, and so on until the last one with probability σ^{n_o}. Then, for the (n_o+1)*th* test case, we do not insert it with probability $1-\sigma^{n_o+1}$.

When we generate a new test case at random, each possible test case in the chosen representation (Section 3.2) can be sampled with non-zero probability, although the sampling is not uniform (Section 3.5.3). Let Ω be the lowest probability for a test case to be sampled. Because N, L and I_{max} are finite, then Ω is a constant.

In an optimal test suite of size n_o, there would be n_o distinct test cases. If any of these are equal, then the duplicates could be removed because they do not increase the coverage (in fact, the execution of test cases is independent, so none of them can have effects on the others). There can be several optimal test suites with size n_o, but, as a lower bound, we can consider just one. Because the order in which the test cases are sampled is not important, we can consider all their $n_o!$ permutations. Therefore, sampling the right test cases has probability at least $\Psi = n_o! \times \Omega^{n_o}$.

Finally, we can prove that the probability p of sampling an optimal solution is at least $p \geq c$, where $c = \rho \times \psi \times \Psi$. □

Although the conditions to apply Lemma 1 are very general, and proving whether they hold is rather straightforward (see the proof of Proposition 1), for many mutation operators proposed in the software testing literature (e.g., [43]) it does not seem that Lemma 1 is applicable. However, Lemma 1 is a sufficient condition, and so it might not be necessary. In other words, it could well be that all the techniques that have been proposed so far in the literature do converge, even if Lemma 1 does not apply to them. However, if a technique does not converge, there might be cases for which it might never find a solution even if left running for infinite time. Without a formal proof that is valid for all software (as for example Lemma 1), it would not be possible to know beforehand whether a technique would converge on any particular addressed problem instance.

7 Threats to Validity

The focus of this paper is on comparing the approach "entire test suite" to "one target at the time".

Threats to *construct validity* are on how the performance of a testing technique is defined. We gave priority to the achieved coverage, with the secondary goal of minimizing the length. This yields two problems: (1) in practical contexts, we might not want a much larger test suite if the achieved coverage is only slightly higher, and (2) this performance measure does not take into account how difficult it will be to manually evaluate the test cases for writing assert statements (i.e., checking the correctness of the outputs).

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 30 times, and we followed rigorous statistical procedures to evaluate their results.

Another possible threat to internal validity is that we did not study the effect of the different configurations for the employed GA. In this paper we claim that EvoSuite is superior to the common approach of focusing on only one target at the time. However, in theory it might be possible that there exist parameter settings for which the one target at the time approach is better than any configuration of EvoSuite. To shed light on this possible issue, we would need to carry out large tuning phases on both the two approaches. However, as already explained earlier in the paper, we preferred to use the computational time of the experiments to have a much larger case study rather than applying tuning phases.

Although we used both open source projects and industrial software as case studies, there is the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis. Furthermore, we evaluated the optimization of entire test suites against going to each testing target individually only by using a GA. The superiority of an EvoSuite-like approach might not hold when other testing techniques are employed (e.g., other types of search algorithms such as Simulated Annealing).

Our EvoSuite prototype might not be superior to all existing testing tools; this, however, is not our claim: We have shown that whole test suite generation is superior to a traditional strategy targeting one test goal at a time. Basically, this insight can be used to improve any existing testing tool, independent of the underlying test criterion (e.g., branch coverage, mutation testing, ...) or test generation technique (e.g., search algorithm), although such a generalization to other techniques will of course need further evidence.

8 Conclusions

Coverage criteria are a standard technique to automate test generation. In this paper, we have shown that optimizing whole test suites towards a coverage criterion is superior to the traditional approach of targeting one coverage goal at a time. In our experiments, this results in significantly better overall coverage with smaller test suites.

While we have focused on branch coverage in this paper, the findings also carry over to other test criteria. Consequently, the ability to avoid being misled by infeasible test goals can help overcoming similar problems in other criteria, for example, the equivalent mutant problem in mutation testing [29].

Even though the results achieved with EvoSuite already demonstrate that whole test suite generation is superior to single target test generation, there is ample opportunity to further improve our EvoSuite prototype. For example, there is potential in combining search-based test generation with dynamic symbolic execution (e.g., [12], [33]), and search optimizations such as testability transformation [24] or local search [26] should further improve the achieved coverage. Furthermore, there are general enhancements in the literature of search algorithms that we could integrate and evaluate in EvoSuite, as for example island models (e.g., see the recent [3]) and adaptive parameter control [32].

In our empirical study, we targeted object-oriented software. However, the EvoSuite approach could be easily applied to procedural software as well, although further research is needed to assess the potential benefits in such a context.

The approach presented in this paper aims at producing small test suites with high coverage, such that the developer can add test oracles in terms of assertions. Although keeping the test suites small is helpful in this respect, the oracle problem is still very difficult. In this respect, we are investigating ways to support the developer by automatically producing effective [21] assertions, and to ease understanding we try to make the produced test cases more readable [20].

To learn more about EvoSuite, visit our Web site:

http://www.evosuite.org

Acknowledgments. We thank Valentin Dallmeier, Yana Mileva, Andrzej Wasylkowski and Andreas Zeller for comments on earlier versions of this paper. This work is funded by a Google Focused Research Award on "Test Amplification" and the Norwegian Research Council.

References

[1] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 6, pp. 742–762, 2010.
[2] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators: Research articles," *Software Testing, Verification, and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.
[3] L. Araujo and J. Merelo, "Diversity through multiculturality: Assessing migrant choice policies in an island model," *Evolutionary Computation, IEEE Transactions on*, no. 99, pp. 1–14, 2011.
[4] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability (STVR)*, 2011, http://dx.doi.org/10.1002/stvr.457.
[5] ——, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering (TSE)*, 2011.
[6] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2011.
[7] ——, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.
[8] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011, pp. 33–47.
[9] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *IFIP International Conference on Testing Software and Systems (ICTSS)*, 2010, pp. 95–110.
[10] ——, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 2, pp. 258–277, 2012.
[11] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.
[12] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic search-based testing," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.
[13] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for java," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.
[14] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "Automatic test cases optimization: a bacteriologic algorithm," *IEEE Software*, vol. 22, no. 2, pp. 76–82, Mar. 2005.
[15] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, pp. 1025–1050, 2004.
[16] W. Feller, *An Introduction to Probability Theory and Its Applications, Vol. 1*, 3rd ed. Wiley, 1968.

[17] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” in *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40.

[18] ——, “Evosuite: Automatic test suite generation for object-oriented software.” in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.

[19] ——, “It is not the length that matters, it is how you control it,” in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 150 – 159.

[20] G. Fraser and A. Zeller, “Exploiting common object usage in test case generation,” in *ICST’11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2011, pp. 80–89.

[21] ——, “Mutation-driven generation of unit tests and oracles,” *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2011.

[22] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *PLDI’05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 213–223.

[23] A. Goldberg, T. C. Wang, and D. Zimmerman, “Applications of feasible path analysis to program testing,” in *ISSTA’94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1994, pp. 80–94.

[24] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, “Testability transformation,” *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[25] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo., “Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem,” in *International Workshop on Search-Based Software Testing (SBST)*, 2010.

[26] M. Harman and P. McMinn., “A theoretical and empirical study of search based testing: Local, global and hybrid search.” *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 2, pp. 226–247, 2010.

[27] K. Inkumsah and T. Xie, “Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution,” in *ASE’08: Proc. of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering*, 2008, pp. 297–306.

[28] M. Islam and C. Csallner, “Dsc+mock: A test case + mock class generator in support of coding against interfaces,” in *International Workshop on Dynamic Analysis (WODA)*, 2010, pp. 26–31.

[29] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” CREST Centre, King’s College London, London, UK, Technical Report TR-09-06, September 2009.

[30] B. Korel, “Automated software test data generation,” *IEEE Transactions on Software Engineering*, pp. 870–879, 1990.

[31] F. Lammermann, A. Baresel, and J. Wegener, “Evaluating evolutionary testability for structure-oriented testing with software measurements,” *Applied Soft Computing*, vol. 8, no. 2, pp. 1018–1028, 2008.

[32] F. Lobo, C. Lima, and Z. Michalewicz, *Parameter setting in evolutionary algorithms*. Springer Verlag, 2007, vol. 54.

[33] J. Malburg and G. Fraser, “Combining search-based and constraint-based testing,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

[34] P. McMinn, “Search-based software test data generation: A survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[35] W. Miller and D. L. Spooner, “Automatic generation of floating-point test data,” *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[36] C. Pacheco and M. D. Ernst, “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, 2007, pp. 815–816.

[37] J. C. B. Ribeiro, “Search-based test case generation for object-oriented Java software using strongly-typed genetic programming,” in *GECCO’08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*. ACM, 2008, pp. 1819–1822.

[38] G. Rudolph, “Convergence analysis of canonical genetic algorithms,” *IEEE transactions on Neural Networks*, vol. 5, no. 1, pp. 96–101, 1994.

[39] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *ESEC/FSE-13: Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*. ACM, 2005, pp. 263–272.

[40] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, “Testing container classes: Random or systematic?” in *Fundamental Approaches to Software Engineering (FASE)*, 2011.

[41] S. Silva and E. Costa, “Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories,” *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 141–179, 2009.

[42] N. Tillmann and J. N. de Halleux, “Pex — white box test generation for .NET,” in *TAP’08: International Conference on Tests And Proofs*, ser. LNCS, vol. 4966. Springer, 2008, pp. 134 – 253.

[43] P. Tonella, “Evolutionary testing of classes,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[44] A. Vargha and H. D. Delaney, “A critique and improvement of the CL common language effect size statistics of McGraw and Wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[45] W. Visser, C. S. Pasareanu, and S. Khurshid, “Test Input Generation with Java PathFinder,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 97–107.

[46] S. Wappler and I. Schieferdecker, “Improving evolutionary class testing in the presence of non-public methods,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2007, pp. 381–384.

[47] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in *GECCO’05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*. ACM, 2005, pp. 1053–1060.

[48] J. Wegener, A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[49] D. Whitley, “The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best,” in *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, 1989, pp. 116–121.

[50] N. Williams, B. Marre, P. Mouy, and M. Roger, “PathCrawler: automatic generation of path tests by combining static and dynamic analysis,” in *EDCC’05: Proceedings ot the 5th European Dependable Computing Conference*, ser. LNCS, vol. 3463. Springer, 2005, pp. 281–292.

[51] T. Xie, D. Marinov, W. Schulte, and D. Notkin, “Symstra: A framework for generating object-oriented unit tests using symbolic execution,” in *TACAS’05: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 365–381.

[52] S. Zhang, D. Saff, Y. Bu, and M. Ernst, “Combined static and dynamic automated test generation,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2011.

PLACE PHOTO HERE

Gordon Fraser is a post-doc researcher at Saarland University. He received a PhD in computer science from Graz University of Technology, Austria, in 2007, and his research concerns the prevention, detection, and removal of defects in software. He develops techniques to generate test cases automatically, and to guide the tester in validating the output of tests by producing test oracles and specifications.

PLACE PHOTO HERE

Andrea Arcuri received a BSc and a MSc degree in computer science from the University of Pisa, Italy, in 2004 and 2006, respectively. He received a PhD in computer science from the University of Birmingham, England, in 2009. Since then, he has been a research scientist at Simula Research Laboratory, Norway. His research interests include search based software testing and analyses of randomized algorithms.