Evolutionary Generation of Whole Test Suites

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—Recent advances in software testing allow automatic derivation of tests that reach almost any desired point in the source code. There is, however, a fundamental problem with the general idea of targeting one distinct test coverage goal at a time: Coverage goals are neither independent of each other, nor is test generation for any particular coverage goal guaranteed to succeed. We present EVOSUITE, a search-based approach that optimizes *whole test suites* towards satisfying a coverage criterion, rather than generating distinct test cases directed towards distinct coverage goals. Evaluated on five open source libraries and an industrial case study, we show that EVOSUITE achieves up to 18 times the coverage of a traditional approach targeting single branches, with up to 44% smaller test suites.**

***Keywords*-Search based software engineering, length, branch coverage, genetic algorithm**

I. INTRODUCTION

In structural testing, tests are generated from source code with the aim of satisfying a coverage criterion. Recent advances allow modern testing tools to efficiently derive test cases for realistically sized programs fully automatically. A common approach is to select one coverage goal at a time (e.g., a program branch), and to derive a test case that exercises this particular goal (e.g., [14], [25]). Although feasible, there is a major flaw in this strategy, as it assumes that all coverage goals are equally important, equally difficult to reach, and independent of each other. Unfortunately, none of these assumptions holds.

This problem manifests itself in several ways: Many coverage goals are simply infeasible, meaning that there exists no test that would exercise them; this is an instance of the undecidable infeasible path problem [11]. For example, consider the stack implementation in Figure 1: The false branch of the `if` condition in Line 7 is infeasible. Targeting this goal will per definition fail and the effort was wasted. The general strategy seems to be to accept this fact and get over it.

Even if feasible, some coverage goals are simply more difficult to satisfy than others. Therefore, given a limited amount of resources for testing, a lucky choice of the order of coverage goals can result in a good test suite, whereas an unlucky choice can result in all the resources being spent on only few test cases. 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, a test case targeting a particular coverage goal will mostly also satisfy further coverage goals by accident

```
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. A simple stack implementation: Some branches are more difficult to cover than others, can lead to coverage of further branches, and some branches can be infeasible.

(collateral coverage, also called serendipitous coverage [13]). Again the order in which goals are chosen influences the result – even if all coverage goals are considered, collateral coverage can influence the resulting test suite. For example, 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. There is no efficient solution to predict collateral coverage or the difficulty of a coverage goal.

We introduce EVOSUITE, a novel approach that overcomes these problems by *optimizing an entire test suite at once* towards satisfying a coverage criterion, instead of considering distinct test cases directed towards satisfying distinct coverage goals. The main contributions of this paper are:

Whole test suite optimization: EVOSUITE optimizes whole test suites with respect to an entire coverage criterion

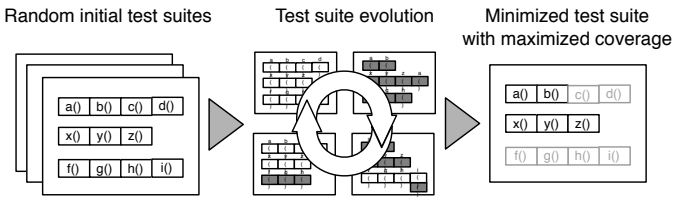


Fig. 2. The EVOSUITE process: A set of randomly generated test suites is evolved to maximize coverage, and the best result is minimized.

```
Stack var0 = new Stack();
int var1 = 0;
try { var1 = var0.pop(); }
catch(EmptyStackException e) {}
int var2 = −434;
var0.push(var2);
int var4 = var0.pop();
```

```
int var0 = −20;
Stack var1 = new Stack();
var1.push(var0);
var1.push(var0);
var1.push(var0);
var1.push(var0);
```

Fig. 3. Test suite consisting of two tests, produced by EVOSUITE for the `Stack` class shown in Figure 1: All feasible branches are covered.

at the same time. This means that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals. In addition, the concept of collateral coverage disappears as all coverage is intentional.

Search-based testing: EVOSUITE uses a search-based approach that evolves a population of test suites. This approach improves over the state of the art in search-based testing by (1) handling dependencies among predicates, (2) handling test case length dynamically without applying exploration impeding constraints, and (3) giving guidance towards reaching test goals in private functions.

Figure 2 illustrates the main steps in EVOSUITE: It starts by randomly generating a set of initial test suites, which are evolved using evolutionary search towards satisfying a chosen coverage criterion (see Section III). At the end, the best resulting test suite is minimized, giving us a test suite as shown in Figure 3 for the `Stack` example from Figure 1. We demonstrate the effectiveness of EVOSUITE by applying it to five open source libraries and an industrial case study (Section IV); to the best of our knowledge, this is the largest evaluation of search-based testing of object-oriented software to date.

II. 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 (e.g., mutation testing [16]) is amenable to automated test generation.

Solving path constraints generated with symbolic execution is a popular approach to generate test data [29] or unit tests [30], and dynamic symbolic execution as an extension can overcome a number of problems by combining concrete executions with symbolic execution (e.g., [10], [21]). This idea has been implemented in tools like DART [10] and CUTE [21], and is also applied in Microsoft's parametrized unit testing tool PEX [24] or the object-oriented testing framework Symstra [30].

Meta-heuristic search techniques have been used as an alternative to symbolic execution based approaches [17]. Search-based techniques have also been applied to test object oriented software using method sequences [**?**], [25] or strongly typed genetic programming [20], [27]. A promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [15]), alleviating some of the problems both approaches have.

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 [16] is a worthwhile test goal, and has been used in a search-based test generation environment [**?**].

When each testing target is sought individually, 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 [5]. Recently, Harman et al. [13] 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 [6] and Baresi et al. [7] 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. [8]. 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 are not directly aimed to achieve code coverage, as for example Randoop [19]. In that work, sequences of function calls are generated incrementally using an extension of random testing (for details, see [19]), and the goal is to find test sequences for which the SUT fails. But this 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 get as output only the test cases for which failures are triggered.

A similar approach is used for example in DART [10] or CUTE [21], in which although path coverage is targeted, an automated oracle (e.g., does the SUT crash?) is used to check the generated test cases. This step is essential because, apart from trivial cases, the test suite 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 in [10], [19]. 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 manually verified, then the generated test suites should 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.

III. 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.

A. 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 III-D).

B. 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. Note, that in this paper we only consider the problem of deriving test inputs, which in unit testing are commonly sequences of method calls. In practice, a test case also contains a test oracle, e.g., in terms of test assertions. This problem is addressed elsewhere (e.g., [**?**]. Given $|T| = n$, we have $T = \{t_1,t_2,\ldots,t_n\}$.

Algorithm 1 The genetic algorithm applied in EvoSuite

```
1  current_population ← generate random population
2  repeat
3    Z ← elite of current_population
4    while |Z| ≠ |current_population| do
5      P1,P2 ← select two parents with rank selection
6      if crossover probability then
7        O1,O2 ← crossover P1,P2
8      else
9        O1,O2 ← P1,P2
10     mutate O1 and O2
11     fP = min(fitness(P1),fitness(P2))
12     fO = min(fitness(O1),fitness(O2))
13     lP = length(P1) + length(P2)
14     lO = length(O1) + length(O2)
15     TB = best individual of current_population
16     if fO < fP ∨ (fO = fP ∧ lO ≤ lP) then
17       for O in {O1,O2} do
18         if length(O) ≤ 2 × length(TB) then
19           Z ← Z ∪ {O}
20         else
21           Z ← Z ∪ {P1 or P2}
22     else
23       Z ← Z ∪ {P1,P2}
24   current_population ← Z
25 until solution found or maximum resources spent
```

In a unit testing scenario, a test case t essentially is a program that executes the software under test (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 [**?**], [25]: A test case is a sequence of statements $t = \langle s_1,s_2,\ldots,s_l\rangle$ of length l. The length of a test suite is defined as the sum of the lengths of its test cases, i.e., $length(T) = \sum_{t\in T} l_t$.

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. There are four different types of statements:

Primitive statements represent numeric variables, e.g., `int var0 = 54`. Value and type of the statement are defined by the primitive variable.

Constructor statements generate new instances of any given class; e.g., `Stack var1 = 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 var2 = var1.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 var3 = var1.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.
For a given software under test, the *test cluster* 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 size from 0 to N (i.e., $n \in [0,N]$). 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.

C. 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. Note that we consider branch coverage at byte-code level. Because all high level branch statements in Java (e.g., predicates in loop conditions and switch statements) are transformed into simpler `if` statements in the byte-code, EvoSuite is able to handle all of them.

Let B denote the set of branches of the SUT, two for every control structure. 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 although potentially feasible if the entire grammar of target language is used. 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 [17]. The branch distance for any given execution of a predicate can be calculated by applying a recursively defined set of rules (see [17] for details). For example, for predicate $x \geq 10$ and $x = 5$, the branch distance to the true branch is $10 - 5 + k$, with $k \geq 1$. In practice, to determine the branch distance each predicate of the SUT is instrumented to evaluate and 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:

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

Here, $\nu(x)$ is a normalizing function in $[0,1]$; we use the normalization function [1]: $\nu(x) = x/(x+1)$. This results in the following fitness function to minimize:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k,T) \qquad (1)$$

D. Bloat Control

A variable size representation could lead to *bloat*, which is a problem that is very typical for example in Genetic Programming [23]: 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 [23].

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 **[?]**), then bloat might occur.

In previous work [9], we have studied several bloat control methods from the literature of Genetic Programming [23]

applied in the context of testing object oriented software. However, our previous study [9] covered only the case of targeting one branch at a time. In EVOSUITE we use the same methods analyzed in that study [9], 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:

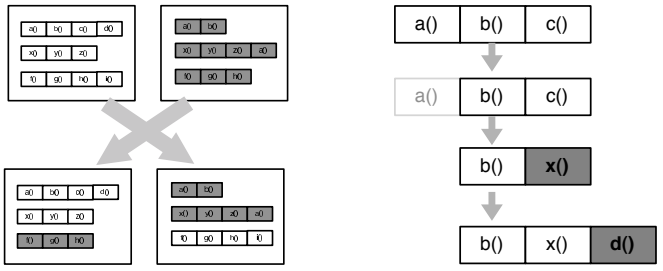
- We put a limit N on the maximum number of test cases and limit L for their maximum length. Even if we expect the length and number of test cases of an optimal test suite to have low values, we still need to choose comparatively larger N and L. In fact, allowing the search process to dive in longer test sequences and then reduce their length during/after the search can provide staggering improvements in term of achieved coverage [2].
- In our GA we use rank selection [28] 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 [6], [7]), 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 in the new generations if their size is bigger than that of the parents (for the details, see Algorithm 1).
- We use a dynamic limit control conceptually similar to the one presented by Silva and Costa [23]. 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 T_B (see Line 18 in Algorithm 1).

E. 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 the chosen solution representation (see Section III-B). 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.

1) Crossover: The crossover operator (see Figure 4(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.



(a) Crossover (b) Mutation

Fig. 4. Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level, while mutation is applied to test cases and test suites.

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 4(b)), then three types of operations are applied in order: *remove*, *change* and *insert*. Each of them 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 $t = \langle s_1, s_2, \ldots, s_n \rangle$ with length n, each statement s_i is deleted with probability $1/n$. As the value $v(s_i)$ might be used as a parameter in any of the statements $s_{i+1}, \ldots, s_n$, the test case needs to be repaired to remain valid: For each statement s_j, $i < j \leq n$, 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 $t = \langle s_1, s_2, \ldots, s_n \rangle$ with length n, each statement s_i is changed with probability $1/n$. If s_i is a primitive statement, then the numeric value represented by s_i is changed by a random value in $\pm[0,\Delta]$, where Δ is a constant. 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 unit under test 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 unit under test. 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 statement 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) 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 III-B), 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 [5]). 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 III-E2 until the test case has a length $\geq r$.

IV. 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.

A. Implementation Details

Our EvoSuite prototype is implemented in Java, and generates JUnit test suites. It does not require the source code of the SUT, as it collects all necessary information for the test cluster from the bytecode via Java Reflection.

During test generation, EvoSuite considers one top-level class at a time. The class and all its anonymous and member classes are instrumented at bytecode level to keep track of called methods and branch distances during execution. Note that the number of branches at bytecode level is usually larger than at source code level, as complex predicates are compiled into simpler bytecode instructions.

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 [2] 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 so difficulties can arise when method signatures are imprecise: For example, container classes [22] often declare `Object` as parameter or return type. Because the search operators have no access to dynamic type information, casting to the appropriate class is difficult. In particular, this problem also 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, we always put `Integer` objects into container classes, such that we can also cast returned `Object` instances back to `Integer`.

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.

B. Single Branch Strategy

To allow a fair comparison with the traditional single branch approach (e.g., [25]), 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 [25] and Fraser and Zeller [**?**]: Offspring is generated using the single point crossover function described in Section III-E1, 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, trying to satisfy dependencies

TABLE I
NUMBER OF CLASSES, BRANCHES, AND LINES OF CODE IN THE CASE STUDY SUBJECTS

Case Study		#Classes		#Branches	LOC[1]
		Public	All		
JC	Java Collections	30	118	3,531	6,337
JT	Joda Time	131	199	7,834	18,007
CP	Commons Primitives	210	213	2,874	7,008
CC	Commons Collections	244	418	8,491	19,041
GC	Google Collections	91	331	3,549	8,586
Ind	Industrial	21	29	373	809
Σ		7271,3all 08		26,652	59,788

with existing values, but generating new values as done during statement insertion, if necessary.

The fitness function in the single branch strategy also needs to be adapted: We use the traditional *approach level* plus normalized branch distance fitness function, which is commonly used in the literature [14], [17].

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 branch where the control flow diverges from the target branch.

Test cases are only generated for branches that have not been covered previously by other test cases. As the order in which branches are targeted can influence the results, we selected the branches in random order. Notice that, in the literature, often no order is specified (e.g., [13], [15], [25]); also, the goal of this paper is not to study the impact of different orders, but rather to compare with current practice.

C. Case Study Subjects

For the evaluation, we chose five open source libraries. 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.* [4]. This results in a total of 1,308 classes, which were tested by only calling the API of the 727 public classes (the remaining classes are private and anonymous member classes).

The projects were chosen with respect to their testability: For experiments of this size, it is necessary that the units are testable without complex interactions with external resources (e.g., databases, networks and filesystem) and are not multi-threaded. In fact, each experiment should be run in an independent way, and there might be issues if automatically generated test cases do not properly de-allocate resources. Notice that this is a problem common to practically all automated testing techniques in the literature. Table I summarizes the properties of these case study subjects.

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

D. Experimental Setup

As witnessed in Section III, 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 $\sigma' = 0.5$. Although longer test cases are better in general [2], 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.

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 [3].

For each of the 727 public classes, we ran EVOSUITE against the single branch strategy to compare their achieved coverage. Each experiment comparison was repeated 100 times with different seeds for the random number generator.

TABLE II
$\hat{A}_{12}$ 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.

Case Study	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$
JC	2	9	19
JT	11	1	119
CP	11	146	53
CC	30	105	109
GC	27	11	53
Ind	0	17	4
Σ	81	289	357

E. Results

Due to space constraints we cannot provide full information of the analyzed data [3], 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 [26]. 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 100 runs of EVOSUITE, we obtained coverage values higher than the ones obtained in *all* of the 100 runs of the single branch strategy.

The box-plot in Figure 5 compares the actual obtained coverage values (averaged out of the 100 runs) of EVOSUITE and the single branch strategy (Single). In total, the coverage improvement of EVOSUITE ranged up to 18 times that of the single branch strategy (averaged over 100 runs). Figure 6 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 II 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 329 out of 438 comparisons in which $\hat{A}_{12} \neq 0.5$. Notice that in many cases 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.

For those cases where $\hat{A}_{12} = 0.5$ for the coverage, Figure 7 compares the obtained test suite size values (averaged out of the 100 runs) of EVOSUITE and the single branch strategy (Single). In the best case, we obtained a test suite size (averaged out of the 100 runs) that for EVOSUITE was 44% smaller.

Figure 8 shows $\hat{A}_{12}$ for the length of the resulting test suites. For length comparisons, we obtained p-values lower than 0.05 in 158 out of 289 comparisons. 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. But collateral coverage of difficult to reach branches 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, 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 cover that difficult branch, then that sequence could 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 easier to cover the other nested branches. 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 those nested branches would be harmed by the fact that covering their difficult parent should be done from scratch again.

Because our empirical analysis employes a very large case study (1,308 classes for a total of 26,652 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.

V. THREATS TO VALIDITY

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

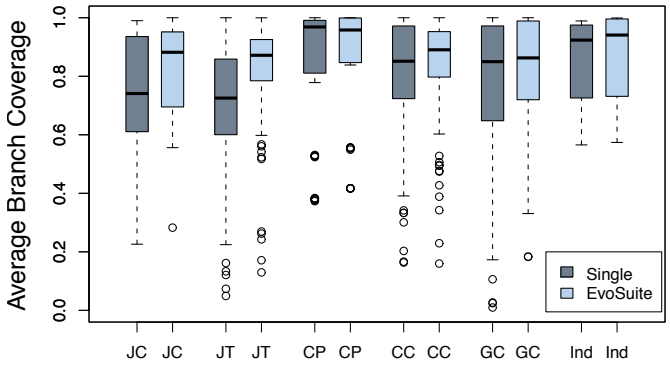


Fig. 5. Average branch coverage: Even with an evolution limit of 1,000,000 statements, EVOSUITE achieves higher coverage.

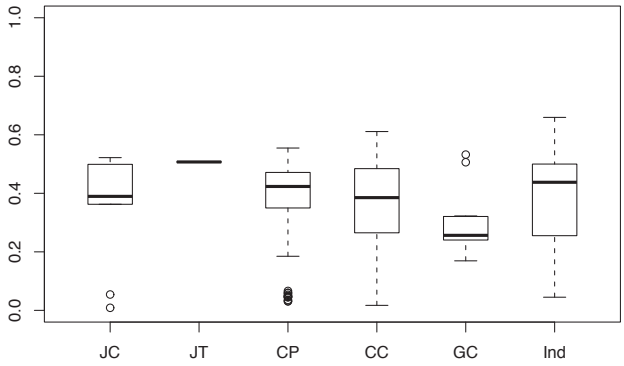


Fig. 8. $\hat{A}_{12}$ for length: EVOSUITE has a low probability of generating longer test suites than a single branch approach.

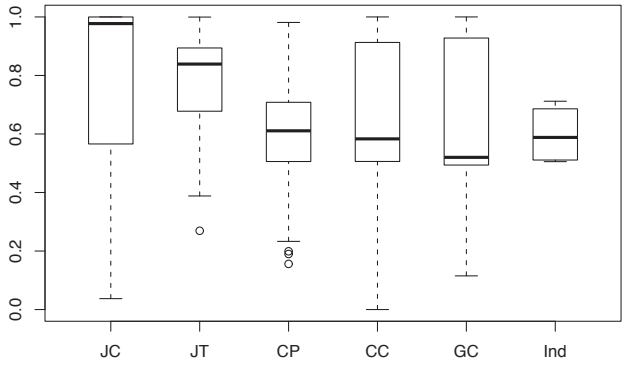


Fig. 6. $\hat{A}_{12}$ for coverage: EVOSUITE has a high probability of achieving higher coverage than a single branch approach.

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).

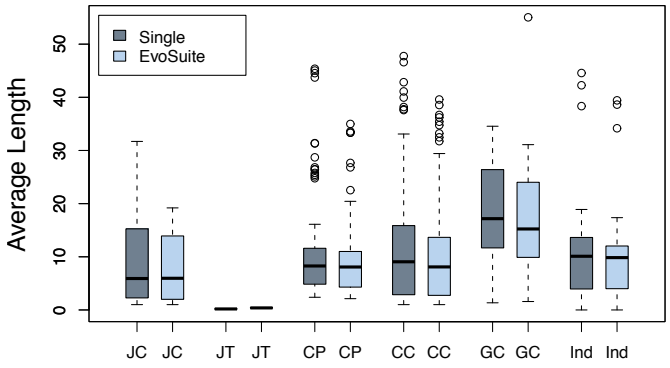


Fig. 7. Average length values: Even after minimization, EVOSUITE test suites tend to be smaller than those created with a single branch strategy (shown for cases with identical coverage).

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 100 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.

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.

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.

VI. 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, and can also reduce the resources needed to generate these 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 [16].

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, EVOSUITE currently has only basic support for arrays and textual inputs, and does not implement all available techniques in search-based testing (e.g., Testability Transformation [12], [18]). Implementing these improvements will further improve the results and make EVOSUITE applicable to a wider range of applications.

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.

Acknowledgments. We thank Valentin Dallmeier, Yana Mileva, Andrzej Wasylkowski and Andreas Zeller for comments on earlier versions of this paper. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. Andrea Arcuri is funded by the Norwegian Research Council.

REFERENCES

[1] 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.

[2] ——, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering*, 2011, http://doi.ieeecomputersociety.org/10.1109/TSE.2011.44.

[3] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *IEEE International Conference on Software Engineering (ICSE)*, 2011.

[4] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *ICTSS'10: Proceedings of the IFIP International Conference on Testing Software and Systems*. Springer, 2010, pp. 95–110.

[5] ——, "Formal analysis of the effectiveness and predictability of random testing," in *ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 219–229.

[6] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.

[7] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for Java," in *ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 185–194.

[8] 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.

[9] G. Fraser and A. Arcuri, "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.

[10] 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.

[11] 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.

[12] 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.

[13] 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 *SBST'10: Proceedings of the International Workshop on Search-Based Software Testing*. IEEE Computer Society, 2010, pp. 182–191.

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

[15] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 297–306.

[16] 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.

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

[18] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, pp. 1–27, 2009.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] P. Tonella, "Evolutionary testing of classes," in *ISSTA'04: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 119–128.

[26] 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.

[27] 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.

[28] D. Whitley, "The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best," in *ICGA'89: Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1989, pp. 116–121.

[29] 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.

[30] 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.