Effective Test Suite Generation for Bugs
A. Brahmani Devi
Dept. of IT, Sir C.R. Reddy College of Engineering, Eluru, AP, India

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 sensitive 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.

Keywords
Search Based Software Engineering, Length, Branch Coverage, Genetic Algorithm, Infeasible Goal, Collateral Coverage

I. Introduction
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.

II. Background
Coverage criteria are commonly used to guide test generation. A coverage criterion represents a finite set of coverage goal s, 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 [13] are amenable to automated test generation.

Solving path constraints generated with symbolic execution is a popular approach to generate test data [5] or unit tests [8], and dynamic symbolic execution as an extension can overcome a number of problems by combining concrete executions with symbolic execution (e.g., [2], [13]). This idea has been implemented in tools like DART [10] and CUTE [3], and is also applied in...

III. Testsuite 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.

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 of test cases, i.e., length(T) = \sum_{i=1}^{l} l_i. 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., [11]). Each statement in a test case represents one value v(s_t), which has a type τ(v(s_t)) ∈ T, where 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 []).
- 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) | 0 ≤ k < i\}.
- Assignment statements assign values to array indices or to public member variables of objects, e.g., vart[0] = new Object() or var2.size = 0.
- 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) | 0 ≤ 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) | 0 ≤ k < i\}.

Algorithm 1 The genetic algorithm applied in EVOSUITE

1. current population ≤ generate random population
2. Z ← elite of current population
3. while |Z| ≤ 6 \{current population\} do
4. P_1, P_2 ← select two parents with rank selection
5. if crossover probability then
6. O_1, O_2 ← crossover P_1, P_2
7. else
8. O_1, O_2 ← P_1, P_2
9. mutate O_1 and O_2
10. f_o = min(fitness(P_1),fitness(P_2))
11. f_z = min(fitness(O_1),fitness(O_2))
12. l_o = length(P_1) + length(P_2)
13. l_z = length(O_1) + length(O_2)
14. T_q = best individual of current population
15. if f_z < f_o ∨ (f_z = f_o ∧ l_z < l_o) then
16. for O in \{O_1, O_2\} do
17. if length(O) < 2 × length(T_q) then
18. Z ← Z ∪ \{O\}
19. else
20. Z ← Z ∪ \{P_1 or P_2\}
21. else
22. Z ← Z ∪ \{P_1, P_2\}
23. current population ← Z
24. until solution found or maximum resources spent

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) | 0 ≤ 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) | 0 ≤ 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., vart[0] = new Object() or var2.size = 0. Assignment statements do not define new values.

For a given SUT, the test cluster [7] defines the set of available classes, their public constructors, methods, and fields. Note that the chosen representation has variable size. Not only can the number n of test cases in a test suite 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 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.

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 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 n of the GA.

IV. 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.
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 data types in byte-code need transformation to carry a distance measurement 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, EVO-SUITE 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.

A. 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 [13]) and C# (e.g., PEX [12]). Similarly, tool prototypes are often targeted for specific operating system s; e.g., PEX [2] and Dsc [9] only work with Windows. Second, although there are popular testing tools for Java, as for example Randoop [6], 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 [5] 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 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”).

V. 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).

VI. Conclusion
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 [9].

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., [12-13]), and search optimizations such as testability transformation [4] or local search [6] 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 [12]. 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 [11] assertions, and to ease understanding we try to make the produced test cases more readable [10].

References


