Optimized Test Suite Generation using Memetic Algorithm: A Survey

Ankita A. Mundade
PG Student, Department of Computer Engineering, North Maharashtra University
SES's R. C. Patel Institute of Technology, Shirpur, MS, India

Tareek M. Pattewar
Assistant Prof., Department of Information Technology, North Maharashtra University
SES's R. C. Patel Institute of Technology, Shirpur, MS, India

ABSTRACT
For developing successful software, testing is a very important component. In software testing, providing input, executes it and check expected output. Many techniques which automatically produce inputs have been proposed over the years, and today are able to produce test suites with high code coverage. In software testing a common scenario is that test data are generated and a tester manually adds test cases. It is a difficult task to generate test cases manually but it is important to produce small representative test sets and this representativeness is typically measured using code coverage. But there is 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. For solving these problems, propose a novel paradigm which is generation of whole test suite based on search based testing. Instead of evolving each test case individually, evolve all the test cases in a test suite at the same time. At the end, the best resulting test suite is minimized.

General Terms
Algorithm, Testing, Optimization, search

Keywords
Length, software-based software engineering, branch coverage, Memetic Algorithm, local search.

1. INTRODUCTION
Software testing is an essential component in IT field of any successful software development process. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results. In software testing providing input, execute it and check expected output. However, in the general case one cannot assume the availability of automated test cases. This means that if we produce test inputs, then human tester needs to specify the test cases in terms of 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 test case generation as easy as possible. A common approach in literature is to generate a test case for each coverage goal and then to combine them in a single test suite. But the size of 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 coverage. Many techniques which are automatically produce inputs have been proposed over the year and today are able to produce test suites. These techniques are to generate a test case for each coverage goal consider example as branches in branch coverage, and then to combine them in a single test suite [9]. So it is difficult to find the size of result test suite. Test suites are usually generated by applying one test at a time.

Test case: Test case is nothing but a set of variables or conditions under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do.

Test suite: Test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions.

In this paper, we evaluate a novel approach for test data generation, which we call generation of whole test suite that improves upon the current approach of targeting one goal at a time. All the test cases are evolved in a test suite at the same time, and the fitness function considers all the testing goals simultaneously. The whole test suite generation improves the current approach of targeting one goal at a time. An evolutionary technique [6] is used in which, instead of evolving each test case individually all the test cases in a test suite are evolved at the same time. The technique starts with an initial population of randomly generated test suites, and then uses a Memetic Algorithm to optimize toward 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 a test suite. The EVOSUITE tool is used to implements this approach 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. It optimizes whole test suites toward a coverage criterion is superior to the traditional approach of targeting one coverage goal at a time.

2. RELATED WORK
In the literature is branch coverage, but in principle any other coverage criterion or related techniques such as mutation testing are amenable to automated test generation. Metaheuristic search techniques have been used as an alternative to symbolic execution-based approaches [6].Search-based techniques have also been applied to test object-oriented software using method sequences [43] or strongly typed genetic programming [8], [10]. When

generating test cases for object-oriented software, since the early work of Tonella [9] , 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.

Any other coverage criterion is amenable to automated test generation. For example, mutation testing is often considered a worthwhile test goal and has been used in a search-based test generation environment. Recently, Harman et al. [5] proposed a search based multi-objective approach in which, although each goal is still targeted individually. And there is the secondary objective of maximizing the number of collateral targets that are accidentally covered.

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[1] and Baresi et al. [2] 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. [3] In that work, test suites are optimized with a search algorithm with respect to mutation analysis.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 [7] tool.

3. METHODOLOGY

3.1 Local Search on Method Call Sequence

Here local search aim is optimizing the values in one particular test case of a test suite. When local search is applied to a test case, EvoSuite iterates over its sequence of statements from the last to the first. And then applies a local search for each statement dependent on different type of the statement. Following are types of statements on which Local search is performed: primitive statements, array statements, method statements, field statements and constructor statements.

3.1.1 Primitive Statements

Here Booleans and Enumerations are considered. For Boolean variables the only two values are considered and option is to flip the values. For enumerations, an exploratory move consists of replacing the enum value with any other value. And we iterate over all enumeration values if the exploratory move was successful. In integer datatypes, for integer variables which includes int, char byte, short, long, the possible exploratory moves are +1 and −1. The exploratory move decides the direction of the pattern move. If an exploratory move to +1 was successful, then with every iteration I of the pattern search we add δ to the variable. If +1 was not successful, −1 is used as exploratory move, and if successful, subsequently ∂ is subtracted. Handle the floating point number with AVM. Exploratory moves are performed for a range of precision values p, where the precision ranges from 0–7 for float variables. When an exploratory move was successful, pattern moves are made by increasing *I* when calculating δ. Exploratory moves are slightly more complicated for string variables. For determining local search on a string variable, first apply n random mutations on the string1. If any of the n probing mutations changed the fitness, then the string has some effect on it, regardless of whether the change resulted in an improvement or not. String values affect the fitness through a range of Boolean conditions. These Boolean conditions are used in branches, and these conditions are transformed such that the branch distance also gives guidance on strings. If the probing on a string showed that it affects the fitness, and then applies a systematic local search on the string.

3.1.2 Array Statements

When we consider local search on array , it concerns the length of an array and also consider values which are assigned to the block of the array. For getting efficient search on the array length, first step of local search is , try to remove assignments to array block. Consider if array of length of n, then we first try to remove the assignments at slot n-1. When fitness value does not change, try or remove assignment at slot n-2 and continue until we find the highest index n' foe which an assignment positively contribute to the fitness value. Then apply a regular integer-based local search on the array length value ,but the length does not get smaller than n'+1. When search has found the best length, expand the test case by with assignments to all slots of the array . These slots of the array are not already assigned in the test case , such assignments may be deleted as part of the regular search. Then apply local search on each assignment to the array, depending on the component type of the array.

3.1.3 Reference Type Statements

Statements (such as method statement, field statement, constructor statement) which are related to reference values do not allow traditional local search in terms of primitive values. The neighborhood of a complex type in a sequence of calls is huge (e.g., all possible calls on an object with all possible parameter combinations, etc.) such that exhaustive search is not a viable option. Therefore, we apply randomized hill climbing on such statements. This local search consists of repeatedly applying random mutations to the statement, and it is stopped if there are R consecutive mutations that did not improve the fitness. We use the following mutations for this randomized hill climbing:

1. Replace the statement with a random call returning the same type.

2. Replace a parameter (for method and constructor statements) or the receiving object with any other value of the same type available in the test case.

3. If the call creates a non-primitive object, add a random method.

3.2 Concept of Memetic Algorithm

Given the ability to perform local search on the individuals of a global optimization there is the question of how to integrate these techniques. Often, MAs are implemented such that individuals can perform Lamarckian or Baldwinian learning immediately after reproduction This, however, raises the questions of how often to apply the individual learning, on which individuals it should be applied, and how long it should be done. Because local search can be very expensive, we would like to direct the learning towards the better individuals of the population, such that newly generated genetic material is more likely to directly contribute towards the solution. In EvoSuite, local search is applied at regular intervals; the rate at which it is applied is the first parameter of local search. When local search is applied, we iterate over the population ranked by their fitness, such that the first individual to be improved is the best individual of the search, then the second

best, and so on. Thus, as a second parameter, there is a search budget for this local search.

4. OBJECTIVES

Our objective is generating whole test suite toward a coverage criteria. This is superior to the traditional approach of targeting one coverage goal at a time. For optimizing the result extended the Genetic Algorithm used in the EVOSUITE test generation tool to Memetic Algorithm. Here we will define a set of local search operator for getting optimized result. And our next objective is to find parameter which makes the local search adaptive.

5. CONCLUSIONS

In this paper, we have described the whole test suite generation technique. Coverage criteria are a standard technique to automate test generation. The EVOSUITE tool implements the approach presented in this paper for generating JUnit test suites. Then optimize the chosen coverage criterion test suite. We use a search algorithm, namely, a Memetic Algorithm (MA) that is applied on a population of test suites.

6. ACKNOWLEDGMENTS

Our thanks to the experts who have contributed towards development of the template.

7. REFERENCES

[1] Arcuri and X. Yao, "Search Based Software Testing of Object- Oriented Containers," Information Sciences, vol. 178, no. 15, pp. 3075-3095, 2008.

[2] L. Baresi, P.L. Lanzi, and M. Miraz, "Testful: An Evolutionary Test Approach for Java," Proc. IEEE Int'l Conf. Software Testing, Verification and Validation, pp. 185-194, 2010.

[3] Baudry, F. Fleurey, J.-M. Je´ze´quel, and Y. Le Traon, "Automatic Test Cases Optimization: A Bacteriologic Algorithm," IEEE Software, vol. 22, no. 2, pp. 76-82, Mar./Apr. 2005.

[4] G. Fraser and A. Arcuri, "Evosuite: Automatic Test Suite Generation for Object-Oriented Software," Proc. 19th ACM SIGSOFT Symp. and the 13th European Conf. Foundations of Software Eng., 2011.

[5] 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 CostProblem," Proc. Third Int'l Conf. Software Testing, Verification, and Validation Workshops, 2010.

[6] P. McMinn, "Search-Based Software Test Data Generation: A Survey," Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105-156, 2004

[7] Pacheco and M.D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," Proc. Companion to the 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems and Application, pp. 815-816, 2007.

[8] J.C.B. Ribeiro, "Search-Based Test Case Generation for Object- Oriented Java Software Using Strongly-Typed Genetic Programming," Proc. GECCO Conf. Companion