EvoSuite: Automatic Test Suite Generation for Object-Oriented Software

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

To find defects in software, one needs *test cases* that execute the software systematically, and *oracles* that assess the correctness of the observed behavior when running these test cases. This paper presents EvoSuite, a tool that automatically generates test cases with assertions for classes written in Java code. To achieve this, EvoSuite applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion. For the produced test suites, EvoSuite suggests possible oracles by adding small and effective sets of assertions that concisely summarize the current behavior; these assertions allow the developer to detect deviations from expected behavior, and to capture the current behavior in order to protect against future defects breaking this behavior.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Experimentation

Keywords

Test case generation, assertion generation, search based software testing

1. INTRODUCTION

When writing programs in object-oriented languages such as Java, one typically performs unit testing to identify defects in the software and to capture the current behavior, such that future defects breaking the behavior are detected.

Test cases can be generated automatically, but there is the issue of the *oracle*, i.e., how to verify that the outputs of the test cases are the expected ones. Faults can sometimes be automatically detected if they lead to program crashes, deadlocks, or violate a formal specification. In many cases, however, there are no formal specifications, and the faults do not lead to program crashes or deadlocks. Therefore, there is the need to provide small test suites (in terms of both test data and assertions) to the user that can be manually verified, i.e., are the assert statements consistent with the semantics of the program?

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

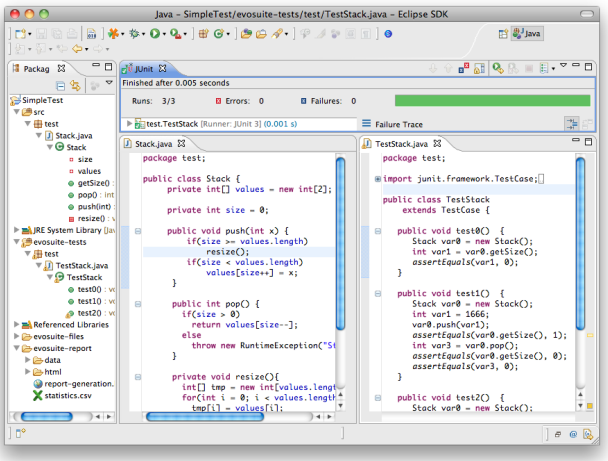

Figure 1: EvoSuite can be used as a command line tool or as an Eclipse plugin, producing coverage test suites for Java classes fully automatically.

EvoSuite is a tool that automates this task by systematically producing test suites that achieve high code coverage, are as small as possible, and provide assertions (see Figure 1). EvoSuite uses a search-based approach integrating state-of-the-art techniques such as for example hybrid search [9], dynamic symbolic execution [7] and testability transformation [8]. Furthermore, EvoSuite implements several novel techniques to efficiently achieve its objectives:

Whole test suite generation: EvoSuite uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time [3]. Optimizing with respect to a coverage criterion rather than individual coverage goals achieves that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals.

Mutation-based assertion generation: EvoSuite uses mutation testing to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the test cases [5]. These assertions highlight the

relevant aspects of the current behavior in order to support developers in identifying defects, and the assertions capture the current behavior to protect against regression faults.

EvoSuite fully automatically produces these test suites for individual classes or entire projects, without requiring complicated manual steps. The tool is freely available, and can be used on the command line, as a plugin to the Eclipse development platform, or through a web interface.

2. WHOLE TEST SUITE OPTIMIZATION

In white box testing, when no automated oracles are available, a common systematic approach to test generation is to select one coverage goal for a given coverage criterion at a time (e.g., a program branch or a control flow path), and to derive a test case that exercises this particular goal (e.g., [9,13]). 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.

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. 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. Furthermore, a test case targeting a particular coverage goal will mostly also satisfy further coverage goals by accident. 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. There is no efficient solution to predict collateral coverage or the difficulty of a coverage goal.

To overcome these problems, whole test suite generation [3] is an approach that does not produce individual test cases for individual coverage goals, but instead focuses on test *suites* targeting an entire coverage criterion. Optimizing with respect to a coverage criterion rather than individual coverage goals achieves 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.

EvoSuite implements whole test suite generation as a search based approach, improving significantly over previous search based approaches (see Figure 2). In evolutionary search, a population of candidate solutions is evolved using operators imitating natural evolution such as crossover and mutation. Individuals are selected for reproduction based on their fitness, i.e., an estimation of how close they are to the optimal solution, and with each generation the fitness improves until a solution is found or the allotted resources are used up.

A candidate solution in EvoSuite is a test suite, consisting of a variable number of individual test cases. Each of these test cases is a sequence of method calls again of variable length, exercising the unit under test (UUT) and setting up complex objects in order to do so. Crossover between two test suites exchanges test cases based on a randomly chosen crossover position. This type of crossover removes the difficulties of crossover on method sequences (e.g., [13]) due to dependencies between statements of a test case. Mutation of a test suite may add new test cases, or mutate individual

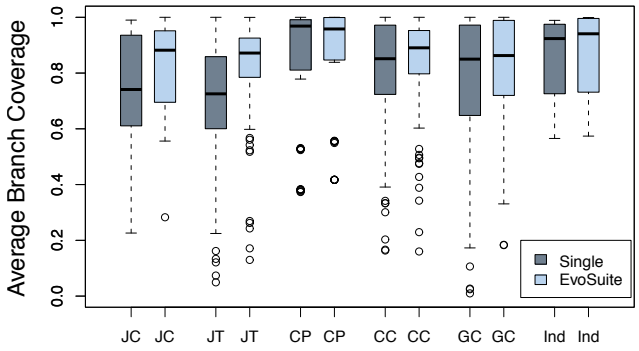

Figure 2: Average branch coverage achieved with EvoSuite on six case study libraries, compared to the standard approach of targeting one branch at a time ("Single"): Even with an evolution limit of 1,000,000 statements, EvoSuite achieves higher coverage [3].

tests. Test cases, in turn, are mutated by adding, deleting, or changing individual statements and parameters. To help the generation of appropriate input data, EvoSuite also makes use of focused local searches and dynamic symbolic execution after every predefined number of generations.

The fitness of individuals is measured with respect to an entire coverage criterion. For example, for branch coverage the fitness calculates the sum of the individual (normalized) branch distances of all the branches in the UUT. The branch distance is a commonly used estimation of how far an individual execution is from having a branch evaluate to true or to false; for example, the condition $x = 42$ with the value 10 for x has a branch distance of $|42 - 10| = 32$ of becoming true. Consequently, an individual has fitness 0 if it covers all branches of a UUT.

As longer method sequences make it easier to reach coverage goals, EvoSuite allows the search to dive into long sequences, but applies several bloat control techniques [4] to ensure that individuals do not become excessively large. At the end of the search, test suites are minimized such that only statements contributing to coverage remain.

3. MUTATION-BASED ASSERTION GENERATION

The aim of producing a coverage test suite is to provide the developer with a representative set of test cases. This allows the software engineer to check and capture the functional correctness of the UUT that cannot be captured with automated oracles. In order to do so, test cases need some kind of manual oracles, which in the case of unit tests typically are assertions. The test oracle problem is one of the main problems in traditional white-box test generation.

Given an automatically generated unit test, there is only a finite number of things one can assert — the choice of assertions is defined by the possible observations one can make on the public API of the UUT and its dependent classes. For example, one can write assertions on return values, compare objects with each other, or call inspector methods on objects. Consequently, synthesizing the possible assertions is easily possible [14]. However, presenting all assertions to

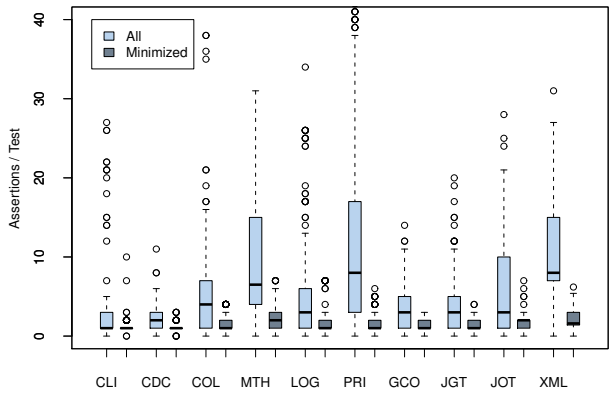

Figure 3: Effectiveness of the mutation-based assertion minimization, evaluated on 10 open source projects [5]. The boxplots summarize the statistics on assertions before ("All") and after ("Minimized") applying the mutation-based minimization.

the developer is problematic, as there might be too many of them, and many of them will be irrelevant: A fault in the current version of the program can only be detected if the developer verifies the correctness of the synthesized assertion and identifies a problem. Similarly, a test case might fail at a later point, indicating a regression failure, when in fact having too many assertions might simply overspecify the test case, leading to false alarms.

In order to determine the important and effective assertions, EvoSuite applies mutation testing. This was originally developed as part of the tool μTest [5], which is now a component of EvoSuite. In mutation testing, artificial defects (mutants) are seeded into a program, and test cases are evaluated with respect to how many of these seeded defects they can distinguish from the original program. A mutant that is not detected shows a deficiency in the test suite and indicates in most cases that either a new test case should be added, or that an existing test case needs a better test oracle. A mutant is detected if an assertion fails; conversely, if an assertion does not detect any of the mutants of a program, it is likely irrelevant to the test case, similarly in principle to a vacuously satisfied property in verification.

After the test case generation process EvoSuite runs each test case on the unmodified software as well as all mutants that are covered by the test case, while recording information on which mutants are detected by which assertions. Then, EvoSuite calculates a reduced set of assertions that is sufficient in order to detect all the mutants of the UUT that the given test case may reveal. Figure 3 shows how effective this reduction is, evaluated on a set of open source libraries. The theory of mutation testing suggests that using only these assertions is sufficient to detect most other possible faults in the UUT [10].

4. IMPLEMENTATION

To generate test suites for Java classes, EvoSuite requires only the Java bytecode of the class under test and its dependencies. The bytecode is analyzed and instrumented, and at the end of the search EvoSuite produces a JUnit test suite for a given class. EvoSuite works fully automatically; it can generate test suites for entire packages without requiring user intervention.

For a given package, EvoSuite considers one class at a time, and tries to produce a test suite maximizing branch coverage for this class. EvoSuite is not restricted to primitive datatypes, but can handle arrays and objects of any class. To produce instances of classes, EvoSuite considers all possible methods and constructors that produce an instance of a required type, and recursively tries to satisfy all dependencies. EvoSuite treats the `String` class specially, by replacing calls to string comparison methods with calls to its own helper methods that calculate string distances based on the Levenshtein distance. This allows the search to also evolve strings towards desired values to satisfy conditions on strings. As another improvement, EvoSuite uses a pool of constant primitive and string values it determines statically in the Java bytecode. However, EvoSuite is currently limited to single threading applications.

As test cases are repeatedly executed as part of the search to measure fitness values, interactions of the code under test with the environment can be unwanted and even dangerous: Random sequences accessing data on the filesystem can result in clutter in the best case, and data loss in the worst case; random accesses to networking or databases are usually not desirable either. EvoSuite provides its own security manager, which can be configured to prohibit unwanted access to the environment. To overcome the problem that random values can lead to very long execution times, EvoSuite uses timeouts and penalizes long executing test cases during the search. To overcome problems with unstoppable very long executions, EvoSuite applies a combination of bytecode instrumentation, thread handling and, in the worst case, restarting the virtual machine.

EvoSuite can be used as a command line tool that produces test suites for individual classes or entire packages, and produces HTML reports for further analysis. For convenience, EvoSuite can also be used with an experimental plugin to the Eclipse development platform (see Figure 1), where test cases can be created with a simple mouse click. Finally, EvoSuite can also be used through a webservice, allowing simple experimentation without any installation.

5. RELATED WORK

Recent advances allow modern testing tools to efficiently derive test cases for realistically sized programs fully automatically. Often, these test cases are generated randomly with the objective to detect program crashes [2] or to find contract violations [11]. To improve over random testing and its variants, which usually achieve low code coverage, techniques based on dynamic symbolic execution have been presented [7]. In contrast to such tools, EvoSuite not only tries to find test cases that violate the automated oracles, but it also aims at producing compact test suites achieving high code coverage, such that developers can use them to evaluate functional correctness.

When using code coverage, a common systematic approach is to select one coverage goal at a time (e.g., a program branch or a control flow path), and to derive a test case that exercises this particular goal (e.g., [9, 13]). In contrast, EvoSuite evolves entire test suites targeting all coverage

goals at the same time, leading to several advantages as discussed in Section 2.

A popular tool for unit testing C# code is Pex [12], which generates input values for parameterized test cases using dynamic symbolic execution. Pex can also produce assertions based on return values of methods, but is limited with respect to classes that require complex method sequences. There are several tools for Java producing JUnit test cases: Randoop [11] is well known for its ease of use, but in contrast to EvoSuite it randomly tests software without guidance in covering complex code structures, reporting violations of predefined contracts.

TestFul [1] and eToc [13] are perhaps the most related tools. Both use a search-based approach to generate JUnit test suites to maximize structural coverage. However, eToc is an old tool that has not been updated for several years, and therefore does not feature the most recent advances in test data generation. On the other hand, TestFul differs from EvoSuite in many critical details, and it is not fully automated. For example, TestFul requires the manual editing of XML files for each class under test, which led its empirical evaluation to be just on 15 classes. On other hand, because EvoSuite is fully automated like Randoop, it was possible to evaluate it on thousands of classes [3].

In terms of assertion generation, Randoop [11] allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra [14] generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. While such approaches can be used to derive efficient oracles, they do not serve to identify which of these assertions are actually useful, and such techniques are therefore only found in regression testing. In contrast, through the μTest tool, EvoSuite uses mutation testing to select an effective subset of assertions.

6. CONCLUSIONS

EvoSuite is a tool that automatically produces test suites for Java programs that achieve high code coverage and provide assertions. EvoSuite implements several novel techniques, leading to higher structural coverage and an efficient selection of assertions based on seeded defects, which is a critical feature that other Java tools miss.

EvoSuite currently supports branch coverage and mutation testing as test objectives, but we are also working on adding further criteria, such as criteria based on data flow, and a further focus of our research is to produce more readable test cases [6].

To learn more about EvoSuite, visit our Web site:

`http://www.st.cs.uni-saarland.de/evosuite/`

Acknowledgments. We thank Andrey Tarasevich, Andre Mis, Sebastian Steenbuck, and David Schuler for their support in developing EvoSuite. This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1, and by a Google Focused Research Award on "Test Amplification". 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.

7. REFERENCES

[1] L. Baresi and M. Miraz. Testful: automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 281–284, New York, NY, USA, 2010. ACM.

[2] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34:1025–1050, September 2004.

[3] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC'11: Proceedings of the 11th International Conference On Quality Software*, 2011.

[4] G. Fraser and A. Arcuri. It is not the length that matters, it is how you control it. In *ICST'11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 150–159. IEEE Computer Society, 2011.

[5] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, pages 147–158, New York, NY, USA, 2010. ACM.

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

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI'05: Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, USA, 2005. ACM.

[8] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[9] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[10] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1:5–20, January 1992.

[11] 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 applications companion*, pages 815–816, New York, NY, USA, 2007. ACM.

[12] N. Tillmann and J. N. de Halleux. Pex — white box test generation for .NET. In *TAP 2008: International Conference on Tests And Proofs*, volume 4966 of *LNCS*, pages 134 – 253. Springer, 2008.

[13] P. Tonella. Evolutionary testing of classes. In *ISSTA'04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, New York, USA, 2004. ACM.

[14] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006: Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.