Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution

Juan Pablo Galeotti  
Saarland University – Computer Science  
Saarbrücken, Germany

Gordon Fraser  
University of Sheffield  
Dep. of Computer Science  
Sheffield, UK

Andrea Arcuri  
Simula Research Laboratory  
P.O. Box 134, 1325 Lysaker, Norway

ABSTRACT

Automatic unit test generation aims to support developers by alleviating the burden of test writing. Different techniques have been proposed over the years, each with distinct limitations. To overcome these limitations, we present an extension to the EvoSuite unit test generator that combines two of the most popular techniques for test case generation: Search-Based Software Testing (SBST) and Dynamic Symbolic Execution (DSE). A novel integration of DSE as a step of local improvement in a genetic algorithm results in an adaptive approach, such that the best test generation technique for the problem at hand is favoured, resulting in overall higher code coverage.

1. INTRODUCTION

Software testing is a difficult and expensive task [11]. Automatic test generation tools aim at lowering the cost of writing tests by enabling users to derive tests automatically. Among the available techniques for generating test cases at the unit-level, Search-Based Software Testing [16] (SBST) and Dynamic Symbolic Execution [12] (DSE) have both been demonstrated to be capable of efficiently achieving high code coverage in some testing scenarios. Nevertheless, limitations are present in both approaches.

SBST is based on heuristics that require frequent test execution, and can therefore become very inefficient if test execution time is high. For example, covering any of the string-based branches in method coverMe in Figure 1 would require a search-based approach to execute many candidate tests with slight variations in the input strings, each time estimating how close the search is to reaching the target branch for a new input string.

On the other hand, DSE can be very efficient when applied on problems which the underlying constraint solver is able to address, yet its scope is much more limited. Despite tremendous recent progress, DSE may still struggle with floating point arithmetics, string datatypes, mixed constraints (e.g., floating points and strings in the same constraint), and sequences of function calls.

To avoid some of these problems, DSE tools often resort to mechanisms such as reflection, where inputs are not created in terms of the existing method interface, but by manipulating internal states directly. This can lead to object states that are difficult to reproduce using the method interface, or even infeasible. Such infeasible object states ultimately lead to test cases that are hard to read and understand, and are useless for debugging purposes. For example, in the Foo class in Figure 2, tools relying on reflection would explore a single entry-point (in this case, method coverMe), and would set the value of x from 5 to 0 without actually invoking the method inc five times. A test case like the one shown in Figure 2 is not only less elegant, it is also infeasible: Given the Foo class definition, no instance of this class can have a value different than "bar", which makes the instance infeasible at the unit-level.

This paper describes a tool that implements the combination of SBST and DSE proposed in detail in [10], which aims to overcome these limitations. The tool extends the search-based EvoSuite unit test generation tool. The details of the original test generation...
and assertion generation is described in [8], and implementation details can be found in [7]. In the new extension of EvoSuite, the genetic algorithm allows test suites to be optimized with a DSE-like approach. This combination allows EvoSuite to successfully generate test cases for the example in Figure 1 as shown in Figure 3. The sequence of method invocations properly handles the object state, no unrealistic object state is ever built, yet high coverage is achieved.

2. TEST GENERATION BACKGROUND

EvoSuite is a search-based test generation tool that applies whole test suite generation in order to find test suites that achieve high code coverage on target units. In this section, we describe this approach as well as DSE.

2.1 Whole Test Suite Generation

SBST describes the use of efficient search algorithms for the task of generating test cases. Genetic algorithms (GA) are one of the most commonly applied search algorithms. GAs mimic the natural process of evolution: An initial population of usually randomly produced candidate solutions is evolved using search-operators that resemble natural processes. In the context of whole test suite generation, this population is made of different test suites. After the randomly produced candidates are ready, the fittest parents are selected for reproduction, mimicking the natural phenomena of survival of the fittest. Crossover combines different parents to generate offspring. The offspring is then mutated with certain probabilities (e.g., new statements are added, existing statements are modified or deleted). Once reproduction is finished, a new generation is ready to be used as new parents for selection. This process continues until either the target coverage criterion has been met (e.g., all branches are covered) or the search budget has been exhausted (e.g., timeout or maximum number of generations).

The traditional approach to SBST is to optimize a test case for each coverage objective in isolation. In this context, choosing how to distribute a limited amount of computational resources over the set of coverage objectives is paramount. What is more, the existence of infeasible coverage goals (like unreachable code or infeasible branches) means that any computational resources invested on trying to achieve them are wasted. We bypass this problematic issue by evolving whole test suites instead of individual test cases: Whole test suite generation [8] optimizes an entire test suite at once towards satisfying a coverage criterion, instead of considering distinct test cases directed towards satisfying distinct coverage goals. This means that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals.

2.2 Dynamic Symbolic Execution

DSE is able to achieve high code coverage by intertwining concrete executions, collecting symbolic conditions and solving constraint systems. Typically DSE starts executing an initial input (e.g., default values for all program inputs). During the execution of the program, DSE keeps a symbolic state mapping each program variable to symbolic values, and a symbolic path condition representing the logical constraints on the inputs forcing the program execution to follow that particular path. Whenever a change to the concrete program state occurs, the symbolic state is updated appropriately to reflect those changes. This symbolic state is used to append branch conditions to the symbolic path condition. Every time the concrete execution follows a particular branch condition, the predicate is evaluated symbolically using the symbolic state. This results in a symbolic condition that is appended to the current path condition. For example, if the statement \( x++ \) is executed, and the symbolic value of \( x \) is \( X_0 \), then the symbolic value of \( x \) after executing the statement is update to \( X_0 + 1 \). Similarly, if there is a branch on the input variable \( x \): if \( x == 5 \) and the symbolic value of \( x \) is \( X + 1 \), then the path condition at this point will be extended with either \( X + 1 == 5 \) or \( X + 1 + 1 \neq 5 \), depending on the actual evaluation of the predicate during the concrete run.

When the concrete execution is finished, DSE has produced a path condition \( P = p_1 \land p_2 \land \ldots \land p_n \) for the initial input. By negating an individual \( p_i \) for \( i \leq n \) DSE can produce a new path condition \( P' = p_1 \land \ldots \land p_{i-1} \land \neg p_i \land \ldots \land p_n \), such that an input that satisfies \( P' \) leads to execution of a path different than \( P \). This is done systematically until no further branches can be negated, i.e., all paths have been explored. This approach has been popularized in particular by the recent development of powerful SMT solvers.

3. COMBINING SBST AND DSE

Our approach of combining SBST and DSE is based on the observation that DSE can be seen as a special case of local search. A local search algorithm [9] explores the neighborhood of a candidate solution during the search, whereas a global search algorithm (e.g., a genetic algorithm) uses a population-based approach to explore larger parts of the search space. Harman and McMinn [13] analyzed the effects of global and local search, and concluded that hybrid approaches (known as memetic algorithms) achieve better performance than global search and local search. Figure 4 depicts the hybrid approach implemented in EvoSuite: At a high level, the hybrid algorithm in EvoSuite conducts global search in terms of sets of sequences of method invocations, where search operators such as crossover and mutation are applied to explore the search space. In addition, however, individuals can be improved with a DSE step, where DSE is applied as a local search step on the primitive values contained in these sequences.

Technically, the integration of DSE into the search on test suites works by choosing an individual test case on which to apply DSE, and converting this test case to a parameterized test case. In a parameterized test, the actual test code is separated from its inputs. This means that any values that can be optimized with a constraint solver directly (e.g., primitives such as float and int values, but also String values) become parameters of the test. For example in [Figure 5] a parameterized unit test is obtained by replacing the “bar” constant value with the argument string0. Then DSE is
4. EVOSUITE: AN SBST+DSE TOOL

EVOSUITE is a unit test generation tool for Java that implements the hybrid approach described in this paper. It is fully automated, i.e., it requires no manual test drivers, entry functions, or parametrized unit tests, and can be used with its command-line interface or as an Eclipse plugin (see Figure 3). EVOSUITE is a mature research prototype that has been successfully applied to many open source projects [5], and has recently won the first two instances of the SBST unit testing tool competition [6]. We have evaluated [10] the performance of the hybrid approach on a set of 38 classes considered to be difficult (e.g., classes depending on complex string input), demonstrating an increase of the branch coverage from 71% of EVOSUITE using only search to 82% using the hybrid approach. We compared different configurations in these experiments, and determined that the best performance is achieved if DSE is applied every 30s, optimizing all primitive values for which a previous mutation has shown them to influence the fitness value. Figure 6 further shows the performance of EVOSUITE on the Roops1 benchmark in comparison to EVOSUITE with only search (GA) and the DSE tool DSC. Finally, to reduce threats to external validity we have applied EVOSUITE with DSE on the SF100 corpus of classes [4], where we observed an overall coverage increase of 1.2%.

Besides the focus on achieving high coverage, EVOSUITE aims to produce tests that are readable and effective at finding faults. To improve readability, EVOSUITE uses a range of post-processing steps after the search, for example in order to minimize sequences of method calls and to minimize primitive values used inside these sequences. In order to balance readability against fault finding effectiveness, EVOSUITE tries to select only assertions that are effective at detecting mutants [9].

4.1 Constraint Extraction

DSE is implemented in EVOSUITE using an instrumentation approach, based on the bytecode instrumentation framework of the DSC tool [14]. To ensure platform independence and require no installation beyond downloading a JAR file, EVOSUITE uses its own constraint representation for which it supplies a solver. However, we are experimenting with the integration of other well-established SMT solvers such as Z3 [2].

4.2 Constraint Solving

One particular advantage of the solver provided in EVOSUITE is that it can handle mixed constraints on integers, floating point numbers, and strings. The solver applies the Alternating Variable Method (AVM) [15], which optimizes each input variable in isolation, guided by a fitness function which, in our case, estimates how close a condition is to being satisfied. In traditional AVM, initially the variables are set to random values; in our scenario, the starting values are those provided by the global search algorithm. AVM considers each variable in turn and applies a local search algorithm on it.

1http://code.google.com/p/roops/
2Available at http://ranger.uta.edu/~csallner/dsc
Once the local search on a variable leads to no further improvement, AVM moves on to the next variable, and so on, until none of the variables lead to an improvement. If that is the case and no solution has been found, then the variables are reset to random values and the search starts over, until either a solution has been found or some other stopping condition applies.

When optimizing an individual variable, the type of search applied depends on the type of the value. In the basic case of an integer value, the search starts with exploratory moves consisting of adding and subtracting 1. If an exploratory move was successful (i.e., the fitness improved), then the search accelerates movement with pattern moves, e.g., +2, then +4, etc. Once the pattern search does not improve the fitness any further, the search on the variable goes back to exploratory moves. If a new exploratory move is successful, pattern search is again applied in the direction of the exploratory move. Once no further optimization of the variable is possible, the search moves on to the next variable.

A variant of this pattern search for floating point variables was defined by Harman and McMinn [13]. Exploratory moves for floating point values are performed for a range of precision values $p$, where the precision ranges from 0–7 for float variables, and from 0–15 for double values. Exploratory moves are applied using $\delta = 2^p \times dir \times 10^{-p}$. Here $dir$ denotes either +1 or −1, and $I$ is the number of the iteration, which is 0 during exploratory moves. If an exploratory move was successful, then pattern moves are made by increasing $I$ when calculating $\delta$.

Search on strings uses a set of simple search operators: First, we attempt to remove characters at the end of the string until no more fitness improvement is observed. Second, on the remaining characters we apply AVM in sequence. As a character (char) can be seen as a numerical value, we can apply the same pattern moves we described for integer values above. Finally, we attempt to add a random character at the back of the string and apply AVM on it. We repeat this step as long as fitness improves.

The search is guided by a fitness function, which is a numerical estimation of how far are we from satisfying a desired goal. The distance for conditions that represent numerical comparisons is calculated using standard branch distance calculation [16]. Distances for string operators are calculated using adhoc functions.

A constraint set is satisfied if the overall distance is 0. In the context of solving the collected constraints, the desired goal is to find a valuation that makes all constraints valid. It is worth mentioning that during the entire constraint solving process fitness is calculated on the desired path condition – there is no need to execute any tests. To calculate the distance for a given input valuation we simply evaluate all symbolic expressions using the concrete values, and calculate a condition-specific distance estimate for each condition. For a given path, the constraint set is interpreted as a conjunction of individual path conditions, thus the overall fitness of a constraint set is the sum of the normalized (in [0, 1]) individual distances.

5. CONCLUSIONS

Creating a tool that combines SBST and DSE requires addressing several integration questions. In this work we focused on an approach integrating these two very successful test generation techniques in a single tool. In order to cope with limitations in current constraint solving technology, we developed our own, search-based constraint solver. As our experiments on thousands of Java classes show [10], integrating SBST and DSE is beneficial in practice, and has no significant negative effect.

The benefits of such an integration lie not only in coverage and performance improvements: The combination retains the distinct advantages of SBST such as the ability to optimize test suites towards any coverage criterion as well as non-functional properties such as size or execution time. Furthermore, the search in the space of sequences of method calls guarantees that resulting test cases only represent valid object states.

EvoSuite as well as its Eclipse plugin are freely available on the EvoSuite website at http://www.evosuite.org. Acknowledgments. This project has been funded by a GoogleFocused Research Award on “Test Amplification”, the Norwegian Research Council, the ERC grant SPECMATE and and EU FP7 grant 295261 (MEALS).

6. REFERENCES


