Generation of Text Suites with Verification and Memetic Algorithms

Pulagam Manikanta
M.Tech Student
GIET
Rajahmundry, A.P, India.

R.R.S.Ravi kumar
Assistant Professor-CSE
GIET
Rajahmundry, A.P, India.

S.Maruthuperumal, Ph.D.
Prof. and HOD CSE and IT
GIET
Rajahmundry, A.P, India.

ABSTRACT
Genetic Algorithms have been successfully applied to the generation of unit tests for classes, and are well suited to create complex objects through sequences of method calls. However, because the neighborhood in the search space for method sequences is huge, even supposedly simple optimizations on primitive variables (e.g., numbers and strings) can be ineffective or unsuccessful. To overcome this problem, we extend the global search applied in the EvoSuite test generation tool with local search on the individual statements of method sequences. In contrast to previous work on local search, we also consider complex data types including strings and arrays. Arigorous experimental methodology has been applied to properly evaluate these new local search operators. In our experiments on a set of open source classes of different kinds (e.g., numerical applications and text processing), the resulting test data generation technique increased branch coverage by up to 32% on average over the normal Genetic Algorithm. A common scenario in software testing is therefore that test data are generated, and a tester manually adds test oracles. As this is a difficult task, it is important to produce small yet representative 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 percent smaller test suites.

Keywords
Search-based software engineering, length, branch coverage, genetic algorithm, infeasible goal, collateral coverage

1. INTRODUCTION
Software testing is one of the most important techniques applied to improve software quality. When there is no automated test oracle available as for example is often the case in white-box testing, test generation techniques aim to produce small test suites with high code coverage, such that these test suites can be analyzed by the developer in feasible time. In the case of object-oriented classes, test cases amount to sequences of method calls, and search-based testing has been demonstrated to be an effective solution as it can handle not only optimizations on primitive datatypes, but also on complex data structures and sequences of method calls. The use of search-based techniques for optimizing entire test suites towards high branch coverage for classes has been shown to be a successful technique for this purpose. Empirical experiments have shown that it is practically usable on a wide range of programs A common approach in the literature is to generate a test case for each coverage goal (e.g., branches in branch coverage), and then to combine them in a single test suite. However, the size of a 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 or serendipitous coverage. When an individual of the search is a test suite consisting of a variable number of test cases, each of which is a sequence of method calls of variable length, then the neighborhood in the search space is simply huge: Here, a neighbor is not only defined by the next successive value for primitive types, but also as all the possible calls that can be inserted on existing objects and all the possible changes that can be performed on the sequences of method calls. If somewhere in such a test suite there is an individual primitive value that needs to be optimized, then the probability of it being mutated during the search with a Genetic Algorithm is low, and so the optimization towards the targets dependent on this value can take long. The urgency of this problem becomes even more apparent when one also considers string variables as primitives, where again the neighborhood is huge. Consider the example test case in Figure 1: Even under very strong simplifications, with a "basic" Genetic Algorithm we would need an average of at least 768,000 costly fitness evaluations (i.e., test executions) to cover the target branch. If the budget is limited, then indeed the approach might fail to cover such goals. To overcome this problem, we extend the Genetic Algorithm used in the whole test suite generation approach to a Memetic Algorithm: At regular intervals, the search inspects the primitive variables and tries to apply local search to improve them. Although these extensions are intuitively

useful and tempting, they add additional parameters to the already large parameter space. In fact, misusing these techniques can even lead to worse results, and so we conducted a detailed study to find the best parameter settings .In detail, the contributions of this paper are: Memetic algorithm for test suite optimization: We present a novel approach to integrate local search on primitive values in a global search for test suites. Local search for complex values: We extend the notion of local searches commonly performed on numerical inputs to string inputs, arrays, and objects. Sensitivity analysis: We have implemented the approaches an extension to the EvoSuite tool and analyze the effects of the different parameters involved in the local search, and determine the best configuration. Evaluation: We evaluate our approach on a set of 16 open source classes, and compare the results to the standard search-based approach that does not include local search.

```
c l a s s Foo
 {
boolean bar ( St r ing s )
 {
i f ( s . equa l s ( ”bar ”) ) // t a r g e t
}
}
Foo foo = new Foo ( ) ; St r ing s = ” t e s t ”;
foo . bar ( s ) ;
```

Example class and test case: In theory, four edits of s can lead to the target branch being covered. However, with a Genetic Algorithm where each statement of the test is mutated with a certain probability (e.g., 1/3 when there are three statements) one would have to be really lucky: If the test is part of a test suite (size 10) of a Genetic Algorithm (population 50) and we only assume a character range of 128, then even if we ignore all the complexities of Genetic Algorithms, we would still need on average at least 50×4×1/(110×13× 1128) = 768,000 fitness evaluations before covering the target branch.

2. VERIFICATION AND TEST

Verification consists in checking that a specification satisfies a property which may be given by a temporal logic formula or another more abstract specification. Although the verification problem is different from the test problem the models used in both activities are very similar. The operational semantics of specifications can be defined in terms of transition systems.

3. BACKGROUND

3.1 Local Search Algorithms

With local search algorithms one only considers the neighborhood of a candidate solution. For example, a hill climbing search is usually started with a random solution, of which all neighbors are evaluated with respect to their fitness for the search objective. The search then continues on either the first neighbor that has improved the fitness, or the best neighbor, and again considers its neighborhood. The search can easily get stuck in local optima, which is typically overcome by restarting the search with new random values. A popular version of hill climbing often used in test data generation is Korel's Alternating Variable Method. The Alternative Variable Method (AVM) is a local search technique similar to hill climbing, and was introduced by Korel. The AVM considers each input variable of an optimization function in isolation, and tries to optimize it locally. Initially, variables are set to random values. Then, AVM starts with exploratory moves on the first variable. For example, in the case of an integer an exploratory move consists of adding a delta of +1 or −1. If the exploratory, move was successful (i.e., the fitness improved), then the search accelerates movement with pattern moves. For example, in the case of an integer, the search would next try +2, then +4, etc. Once the next step of the pattern search does not improve the fitness any further, the search goes back to exploratory moves on this variable. If 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. If no variable can be improved the search restarts at another randomly chosen location to overcome local optima.

3.2 Global Search Algorithms

In contrast, global search algorithms try to overcome local optima in order to find more globally optimal solutions. Harman and McMinn recently determined that global search is more effective than local search, but less efficient, as it is more costly. With evolutionary testing, one of the most commonly applied global search algorithms is a Genetic Algorithm (GA). A GA tries to imitate the natural processes of evolution: An initial population of usually randomly produce candidate solutions is evolved using search operators that resemble natural processes. Selection of parents for reproduction is based on their fitness (survival of the fittest). Reproduction is performed using crossover and mutation with certain probabilities. With each iteration of the GA, the fitness of the population improves, until either an optimal solution has been found, or some other stopping condition has been With local search algorithms one only considers the neighborhood of a candidate solution. For example, a hill climbing search is usually started with a random solution, of which all neighbors are evaluated with respect to their fitness for the search objective. The search then continues on either the first neighbor that has improved the fitness, or the best neighbor, and again considers its neighborhood. The search can easily get stuck in local optima, which is typically overcome by restarting the search with new random values. A popular version of hill climbing often used in test data generation is Korel's Alternating Variable Method. The Alternative Variable Method (AVM) is a local search technique similar to hill climbing, and was introduced by Korel. The AVM considers each input variable of an optimization function in isolation, and tries to optimize it locally. Initially, variables are set to random values. Then, AVM starts with exploratory moves on the first variable. For example, in the case of an integer an exploratory move consists of adding a delta of +1 or −1. If the exploratory. move was successful (i.e., the fitness improved), then the searc accelerates movement with pattern moves. For example, in the case of an integer, the search would next try +2, then +4, etc. Once the next step of the pattern search does not improve the fitness any further, the search goes back to exploratory moves on this variable. If 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. If no variable can be improved the search restarts at another randomly chosen location to overcome local optima.

3.3 Global Search Algorithms

In contrast, global search algorithms try to overcome local optima in order to find more globally optimal solutions. Harman and McMinn recently determined that global search is more effective than local search, but less efficient, as it is more costly. With evolutionary testing, one of the most commonly applied global search algorithms is a Genetic

Algorithm (GA). A GA tries to imitate the natural processes of evolution: An initial population of usually randomly produce candidate solutions is evolved using search operators that resemble natural processes. Selection of parents for reproduction is based on their fitness (survival of the fittest). Reproduction is performed using crossover and mutation with certain probabilities. With each iteration of the GA, the fitness of the population improves, until either an optimal solution has been found, or some other stopping condition has been met (e.g. maximum time or number of fitness evaluations). In evolutionary testing, the population would for example consist of test cases, and the fitness estimates how close a candidate solution is to satisfying a coverage goal. The initial population is usually generated randomly, i.e., a fixed number of random input values is generated. The operators used in the evolution of this initial population depend on the chosen representation. A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found. For example, to generate tests for branch coverage, a common fitness function integrates the approach level (number of unsatisfied control dependencies) and the branch distance (estimation of how close the deviating condition is to evaluating as desired). Such search techniques have not only been applied in the context of primitive data types, but also to test object-oriented software using method sequences.

3.4 Memetic Algorithms

A Memetic Algorithm (MA) hybridizes global and local search. The use of MAs for test generation was originally proposed by Wang and Jeng in the context of test generation for procedural code. Arcuri combined a GA with hill climbing to form a MA when generating unit tests for container classes. Harman and McMinn analyzed the effects of global and local search, and concluded that MAs achieve better performance than global search and local search. Baresi et al. also use a hybrid evolutionary search in their TestFul test generation tool, where at the global search level a single test case aims to maximize coverage, while at the local search level the optimization targets individual branch conditions.

Algorithm 1. The genetic algorithm applied in EVOSUITE

1 current population generate random population
2 repeat
3 Z elite of current population
4 while jZj 6¼ jcurrent populationj 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 fÞ ¼ minðfitnessðP1Þ; fitnessðP2ÞÞ 12 fO ¼ minðfitnessðO1Þ; fitnessðO2ÞÞ 13 lÞ ¼ lengthðP1Þ þ lengthðP2Þ
14 lO ¼ lengthðO1Þ þ lengthðO2Þ
15 TB ¼ best individual of current population 16 if fO < fÞ _ ðfO ¼ fÞ ^ lO lÞ Þ then
17 for O in fO1;O2g do
18 if lengthðOÞ 2 lengthðTBÞ then 19 Z Z [fOg
20 else
21 Z Z [fP1 or P2g 22 else
23 Z Z [fP1; P2g
24 current population Z
25 until solution found or maximum resources spent

4. WHOLE TEST SUITE GENERATION

In whole test suite generation, the optimization target is not to produce a test that reaches one particular coverage goal, but it is to produce a complete test suite that maximizes coverage, while minimizing the size at the same time.

Representation: An individual of the search is a test suite, which is represented as a set T of test cases ti. Given |T| = n, we have T = {t1,t2, . . . ,tn}. A test case is a sequence of statements t = hs1,s2, . . . ,sli 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) = Pt T lt. There are several different types of statements in a test case: Primitive statements define primitive values, such as Booleans, integers, or Strings Constructor statements invoke constructors to produce new values; Method statements invoke methods on existing objects, using existing objects as parameters; Field statements retrieve values from public members of existing objects; Array statements define arrays; Assignment statements assign values to array indexes or public member fields of existing objects. Each of these statements defines a new variable, with the exception of void method calls and assignment statements. Variables used as parameters of constructor and method calls and as source objects for field assignments or retrievals need to be already defined by the point at which they are used in the sequence. Crossover of test suites means that offspring recombine subsets from parent test suites. For example, for two selected parents P1 and P2, a random value is chosen from [0,1], and on one hand, the first offspring O1 will contain the first |P1| test cases from the first parent, followed by the last (1 − alpha)|P2| test cases from the second parent. On the other hand, the second Off spring O2 will contain the first |P2| test cases from the second parent, followed by the last (1 − alpha)|P1|| test cases from the first parent. Mutation of test suites means that test cases are inserted, deleted, or changed. With probability sigma, a test case is added. If it is added, then a second test case is added with probability Sigma square, and so on until the ith test case is not added (which happens with probability 1 – sigma square). Each test case is changed with probability 1/|T|. There are many different options to change a test case: One can delete or alter existing statements, or insert new statements. We perform each of these three operations with probability 1/3; on average, only one of them is applied, although with probability (1/3)3 all of them are applied. When removing statements from a test it is important that this operation must ensure that all dependencies are satisfied. Inserting statements into a test case means inserting method calls on existing calls, or adding new calls on the class under test.

Fitness Function: In this paper, we consider branch coverage as the optimization target, although the approach can be applied to any coverage criterion that can be expressed with a fitness function. Typically, fitness functions for other coverage criteria are based on the branch coverage fitness function. Branch coverage requires that for every conditional statement in the code there is at least one test that makes it evaluate to true, and one that makes it evaluate to false. For this, we can use a standard metric used in search based testing, the branch distance. For every branch, the branch distance estimates how close that branch was to evaluating to true or to false. For example, if we have the branch x == 17, and concrete test case where x has the value 10, then the branch distance to make this branch true would be 17 − 10 = 7, while the branch distance to making this branch false is 0.

To achieve branch coverage in whole test suite generation, the fitness function tries to optimize the sum of all normalized, minimal branch distances to 0 – if for each branch there exists a test such that the execution leads to a branch distance of 0, then all branches have been covered.

5. SEARCH GUIDANCE ON STRINGS

The fitness function in whole test suite generation is based on branch distances. EvoSuite works directly on Java bytecode, where except or reference comparisons, the branching instructions are all based on numerical values. Comparisons on strings first map to Boolean values, which are then used in further computations; e.g., a source code branch like if (string1.equals(string2)) consists of a method call on String. Equals followed by a comparison of the Boolean return value with true. To offer guidance on string based branches we replace calls to the String. Equals method with a custom method that returns a distance measurement. The branching conditions comparing the Boolean with true thus have to be changed to check whether this distance measurement is greater than 0 or not (i.e., == true is changed to == 0, and == false is changed to > 0). The distance measurement itself depends on the search operators used; for example, if the search operators support inserts, changes, and deletions, then the Levenshtein distance measurement can be used. This transformation is an instance of testability transformation which is commonly applied to improve the guidance offered by the search landscape of programs. Search operators for string values have initially been proposed by Alshraideh and Bottaci. Based on our distance measurement, when a primitive statement defining a string value is mutated, each of the following is applied with probability 1/3 (i.e., with probability (1/3)3 all are applied):

Deletion: Every character in the string is deleted with probability 1/n, where n is the length of the string. Thus, on average, one character is deleted.

Change: Every character in the string is changed with probability 1/n; if it is changed, then it is replaced with a random character.

Insertion: With probability= 0.5, a random character is inserted at a random position p within the string. If a character was inserted, then another character is inserted with probability 2, and so on, until no more characters are inserted.

6. EVALUATION

The presented techniques depend on a number of parameters, and so evaluation needs to be carefully done with respect to these. We therefore aim to empirically answer the following three research questions:

RQ1: Does local search improve the performance of whole test suite generation?

RQ2: Which parameter combination gives the best results? RQ3: How do the results vary based on the available search budget?

Experimental Setup:

To answer the research questions, we first need to decide for how long and how often to run local search in the MA; how often we apply local search depends on the number X of generations, how much local search is actually done is dependent on the population size. Consequently, we also had to consider the population size when designing the experiments. We also considered seeding from byte code as a further parameter to experiment with, as we expected it to have a large impact on the performance in the cases in which local search is successful (and this is confirmed in the experiments). In total, we had four different parameters to experiment with. For our evaluation, we used the classes already used in previous experiments, but had to exclude those on which EvoSuite trivially achieves 100% coverage. In the choice of a variegated set of classes to experiment with, we tried to strike a balance among the different kinds of classes. To this end, beside classes coming from the case study in, we also included four benchmark classes on integer and floating point calculations from the Roops2 benchmark suite for object-oriented testing, This results in a total of 16 classes, The use of only 16 classes was necessitated by the complex evaluation setup. In this paper, we study the effects of four different parameters. For population size, local search budget and rate we considered five different values, i.e., {5, 25, 50, 75, 100}, and for seeding two (on/off). We also included further configurations without local search (i.e., the default GA in EvoSuite), but still considering the different combinations of population size and seeding. On each class, for each parameter, we ran EvoSuite 30 times with different random seeds to take into account their random nature. In each run, the stopping criterion was a 10 minute timeout. Thus, in total the experiments took$((2 \times 53) + (2 \times 5)) \times (30 \times 10 \times 16)/(60 \times 24) = 866$ days of computational time, which required the use of a cluster of computers. During these runs, EvoSuite was configured using the optimal configuration determined in our previous experiments on tuning. Results For both the cases in which seeding were used and not, we analyzed the 125 configurations using MA, and chose the one that resulted with highest average coverage over the 16 classes in the case study. The same is done for the basic GA, i.e., we evaluated which configuration of the population size gave best results. We call these four configurations (two for MA, and two for GA)"tuned".To evaluate the statistical and practical differences among the different settings, we followed the guidelines in. Statistical difference is evaluated with a two-tailedMann-Whitney U-test, whereas the magnitude of improvement is quantified with the Vargha-Delaney standardized effect size Aˆ12.with high statistical confidence, that the MA outperforms the standard GA in many, but not all, cases. For classes such as Cookie, improvements are as high as a $87 - 55 = 32\%$ average coverage difference (when seeding is not used). RQ1: The MA achieved up to a 32% higher branch coverage than the standard GA. one thing that is clearly visible in Table 2 is that seeding, as expected leads to higher results. On one hand, when seeding is not used, the difference in average coverage between the MA and the GA is $86-79 = 7\%$. On the other hand, when seeding is used, the difference is $89 - 88 = 1\%$. At a first look, such an improvement might be considered low. But the statistics in Table 2 points out a relatively high average 0.61 effect size, with four classes having a strong statistical difference. This is not in contrast with the 1% difference in the raw values of the achieved coverage.

7. THREATS TO VALIDITY

S paper compares the whole test suite generation approach based on a Genetic Algorithm to a hybrid version that uses a Memetic Algorithm with local search. Threats to construct validity are on how the performance of a testing technique is defined. We measured the performance in terms of branch coverage. However, in practice we might not want a much larger test suite if the achieved coverage is only slightly higher.Furthermore,this performance measure does not take into account how difficult it will be to manually evaluate the test cases and to add assert statements (i.e., to check 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. There is also the threat to external validity regarding the generalization to other types of software, which is common for any empirical analysis. Because of the large number of experiments required, we only used 16 classes for our evaluation. To make claims on generalization we will need to conduct further studies on representative sets of classes.

8. CONCLUSIONS

Whole test suite generation has been shown to be effective at producing test suites with high coverage for object-oriented particular statement in a test suite of sequences of method calls have a low probability of occurring, it is not necessarily efficient. To overcome this problem, we have defined a set of local search operators, and extended the Genetic Algorithm used in the EvoSuite test generation tool to a Memetic Algorithm. Experiments on a set of case study classes have demonstrated that this approach results in higher coverage given a fixed search budget. We have observed that the effect is very dependent on the class on which test generation is applied, which makes it difficult to find an optimal parameter configuration. It will therefore be important for future work to make the local search adaptive, such that local search operators that lead to success on a particular problem instance are applied more frequently than those that are not. 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 assertions and, to ease understanding; we try to make the produced test cases more readable.

9. REFERENCES

[1] M. Alshraideh and L. Bottaci. Search- based software test data generation for string datausingprogram-specificsearch operators. Software Testing, Verification, and Reliability, 16(3):175–203, 2006.

[2] A. Arcuri. Theoretical analysis of local search in software testing. In Symposium on Stochastic Algorithms, Foundations and Applications (SAGA), pages 156–168, 2009.

[3] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR), 2012. (to appear).

[4] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In International Symposium on Search Based Software Engineering (SSBSE), pages 33–47, 2011.

[5] A. Arcuri and X. Yao. A memetic algorithm for test data generation of object- oriented software. In IEEE Congress on Evolutionary Computation (CEC), pages 2048–2055, 2007.

[6] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for java. In IEEE Int. Conference on Software Testing, Verification and Validation (ICST), pages 185–194, 2010.

[7] R. Ferguson and B. Korel. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology, 5(1):63–86, 1996.

[8] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In IEEE Int. Conference on SoftwareTesting,Verificationand Validation (ICST), pages 121–130, 2012.

[9] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In ACM/IEEE International Conference on Software Engineering (ICSE), pages 178–188, 2012.

[10] G. Fraser and A. Arcuri. Whole test suite generation. IEEE Transactions on Software Engineering, 39(2):276–291, 2013.

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

[12] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P.Tonella, and T. Vos, "Symbolic Search-Based Testing," Proc. IEEE/ACM 26th Int'l Conf. Automated Software Eng., 2011.

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

[14] B. 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.

[15] C. Csallner and Y. Smaragdakis, "JCrasher: An Automatic Robustness Tester for Java," Software Practice and Experience, vol. 34, pp. 1025-1050, 2004.

[16] W. Feller, An Introduction to Probability Theory and Its Applications, vol. 1, third ed. Wiley, 1968.

[17] G. Fraser and A. Arcuri, "Evolutionary Generation of Whole Test Suites," Proc. 11th Int'l Conf. Quality Software, pp. 31-40, 2011.

[18] 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.

[19] G. Fraser and A. Arcuri, "It Is Not the Length That Matters, It Is How You Control it," Proc. Fourth IEEE Int'l Conf. Software Testing, Verification and Validation, pp. 150-159, 2011.

[20] G. Fraser and A. Zeller, "Exploiting Common Object Usage in Test Case Generation," Proc. Fourth IEEE Int'l Conf. Software Testing, Verification and Validation, pp. 80-89, 2011.