Automated Test Generation for Multi-state Systems

Pedro Reales Mateo
University of Castilla-La Mancha
Ciudad Real, Spain
(+34)926295354 ext.96607
pedro.reales@uclm.es

Macario Polo Usaola
University of Castilla-La Mancha
Ciudad Real, Spain
(+34)926295354 ext.3730
macario.polo@uclm.es

ABSTRACT
This paper describes a genetic algorithm based on mutation testing to generate test cases for classes with multiple states. The fitness function is based on the *coverability* and the *killability* of the individuals. The paper includes a small empirical section that shows evidences of the ability of the algorithm to generate good test cases.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features, Software testing.

General Terms
Algorithms, Desig, Reliability, Experimentation and Verification.

Keywords
Mutation testing, Genetic algorithms., Tests generation

1. INTRODUCTION
Executable test cases are pieces of code that exercise some functionality of the system under test (the SUT) to find errors. One of the main problems to test classes is "the state" problem. Usually, an instance of a class may have different states and, to reach a determinate state, it can be required that the object has been previously in other states. For example, think in a "traffic-lights" object: to reach the state "red", it should be in the "yellow state" previously. Typically, test generation algorithms do not manage this issue specifically: they just generate test data and sequences of calls in order to exercise the methods of the SUT.

This article presents a novel algorithm called MACO (Mutation-based generation of Automated tests for Classes with multiple states Ordered or restricted) that efficiently manages the states in order to improve the test generation tasks. The main contributions of our algorithm are: 1) Test cases are bidimensionally codified with (a) calls to setup the SUT in a concrete state, and (b) calls to exercise the SUT services from such concrete state; and 2) The fitness function, which is based on mutation testing [1] and is calculated in terms of the *coverability* and the *killability* of the test case. The presented algorithm is implemented in Bacterio [2], a tool to automate the whole testing process of Java applications.

This paper is organized as follows: after a revision of some related works, the algorithm is described Section 3. Section 4 presents a first empirical validation. Finally, we draw our conclusions and future line of work.

2. RELATED WORK
Optimization techniques have been widely used to generate test data, such as tabu search [3] and, also, genetic algorithms, since Shiba et al. [4]. Some of the most recent work related with this paper is Baudry et al. [5], who describe a "bacteriologic" algorithm ("inspired in GA") whose fitness is calculated in terms of the mutation score and Fraser et al [6], who describes an mutation based GA algorithm to generate test cases, which include oracles. In general, authors are not too explicit about their algorithms' details.

Copyright is held by the author/owner(s).
GECCO'13 Companion, July 6–10, 2013, Amsterdam, the Netherlands.
ACM 978-1-4503-1964-5/13/07.

3. DESCRIPTION OF THE ALGORITHM
This section explains in detail the proposed algorithm. The MACO algorithm is specially designed to manage the states of the objects. This is done through the genetic representation of the test cases and the designed crossover and mutation operations. In the MACO algorithm, a test case is represented as a bidimensional structure: on the one side, there is a sequence of calls to: (1) the constructor, (2) maybe calls to operations to initialize required objects (class fields, for example) and (3) calls to services of the SUT; on the other side, the test case also has a set of test data (which are the *Param values* elements on the right side of the figure) and a *return value* element.

Due to the bidimensional genetic representation, the crossing is not so direct as usual, and some prevention must be considered. There are three possibilities: 1) "cross in depth": this combines the calls to constructors and the list of inits, what ensures diversity; 2) "swap cross": this crosses the list of inits and exchange the constructor call when the resulting test proceeding from a crossing in depth is not valid; 3) "methods cross": this exchanges the method calls of two individuals: the elements in the same position of the two sequences are crossed in depth and the remaining elements are randomly distributed between the descendants.

Besides the crossover function, genetic mutations are important to create descendants that cannot appear only with the crossover function. We have designed three kinds of mutation operators: 1) to mutate a single element, which implies to generate randomly a new instance of the element or mutate the components elements are mutated; 2) to mutate a list of elements, which implies to generate randomly the whole list or mutate each element of the list changing its position in the list; and 3) to mutate basic values, where mutations for numeric (increment, decrement and negation), boolean (negation) and string (random generation) values are applied.

3.1 Fitness function
The fitness function of the MACO algorithm is based on mutation testing. However, unlike other also mutation-based genetic algorithms, the fitness function is based in two properties of the test cases: *Coverability* and *Killability*.

The *coverability* of a test determines the ability of the test case to reach mutations, taking into account how difficult is to reach each mutation. The formula to calculate the coverability appears in Equation 1, where m_i represents the *i-th* mutant, and c is the number of mutants covered by the test case t. On the other side, the *killability* of a test case determines the ability of a test case to kill mutants, taking into account how difficult is to kill the killed

mutant. The expression to calculate the killability appears in Equation 1, where k is the number of mutants killed by the test case t, and m_i represents the *i-th* mutant.

$$\boldsymbol{Coverability}(t) = \frac{\sum_{i=1}^{c}[1 - MutantReachability(m_i)]}{c}$$

$$MutantReachability(m) = \frac{Number\ of\ tests\ that\ cover\ m}{Total\ number\ of\ tests}$$

$$\boldsymbol{Killability}(t) = \frac{\sum_{i=1}^{k}[1 - MutantFragility(m_i)]}{k}$$

$$MutantFragility(m) = \frac{Number\ of\ tests\ that\ kill\ m}{Total\ number\ of\ tests}$$

Equation 1. Calculus of the *coverability* and *killability*

Coverability rewards a test case as it visits more unvisited mutations, whereas killability rewards those test cases that kill more hard-to-kill mutants. In order to combine the two fitness values (killability and coverability), the algorithm has the same structure that a NSGA-II algorithm [7], which is designed to combine several fitness values in the same genetic algorithm.

4. EMPIRICAL ANALYSIS

In order to analyze the proposed algorithm empirically, we have generated tests for three classes that have the "state problem" commented in the introduction of this paper (*Board*, *Player* and *Street* classes of the *Monopoly* application, which have been used in previous studies [8]), and to the classic Triangle-type determination problem, which has not the "state problem".

TABLE I shows the execution results of the algorithm: number of iterations, number of generated tests, mutation score achieved by the final set of tests (note that equivalent mutants were not identified) and statement coverage of the final set.

TABLE I. Experimental results (I=iterations, ST=selected tests, MS= mutation score and SC = statement coverage)

Classes	# I	# ST	MS	SC
Street (175 LOC)	24	17	93,96%	90,03
Board (1239 LOC)	77	60	79,32%	99,9%
Player (962 LOC)	84	103	79,06%	70,01%
Triangle (170 LOC)	53	24	83,92%	98,2%

We can see that in 238 iterations, the algorithm was able to generate 204 tests for the four classes. In mean, the mutation achieved by the test cases is 84% and the statement coverage is 90%. An interesting data was obtained with the *Player* class. The mutation score obtained against this class was 79,06% but the statement coverage was 70,01%. From the result of the experimentation, 83 mutants from the 535 mutants of the *Player* class (15%), were not covered by the test cases, which explains why the statement coverage has a low value.

Figure 1 shows how the killability and the coverability evolve for each class. The graphs show that these values can get better or worse in each iteration, but they show a trend positive, being better across successive generations. Again, a special case is the *Player* class: the mean coverability obtained with this class is very high from the beginning of the execution. This data explains the coverage results obtained with this class, since the algorithm was focused on increasing the killability and did not search to cover new mutants. This result suggests a possible improvement in the adjustment of the calculus of the fitness function.

The empirical results obtained from the execution of the MACO algorithm against the *Street*, *Board*, *Player* and *Triangle* classes induce to think that the proposed algorithm is able to overcome the consequences derived from the "state problem", which make difficult the automatic generation of tests. MACO also works properly to evolve basic values.

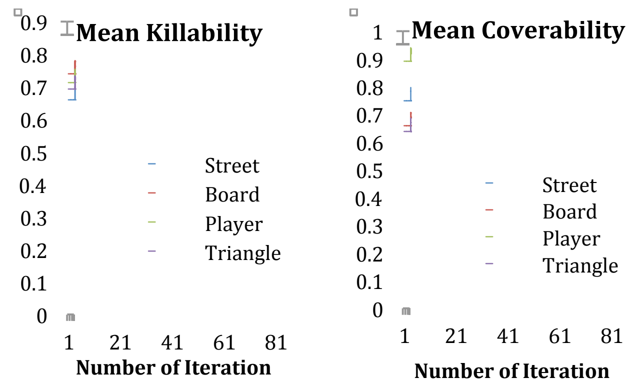

Figure 1. Mean of the killability and coverability

5. CONCLUSIONS AND FUTURE WORK

In this paper a novel genetic algorithm is proposed to improve the task of generating tests automatically for classes with multiple states with different restrictions, specially execution preconditions (in particular, existence of certain objects in concrete states to execute SUT services). As a future work, we will improve the algorithm with the development of new techniques to generate the initial population and in the mutation through diminishing randomness and providing some extra information to the algorithm that could guide the search.

6. ACKNOWLEDGMENTS

This paper has been partially supported by the GEODAS-BC project (TIN2012-37493-C03-01). Pedro Reales has a FPU from Ministerio de Educación (AP2009-3058).

7. REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, Cambridge, UK, 2008.

[2] P. R. Mateo and M. P. Usaola, "Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases," 2012, pp. 646–649.

[3] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado, "A tabu search algorithm for structural software testing," *Computers & Operations Research*, 35(10), pp. 3052–3072, Oct. 2008.

[4] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2004, pp. 72 –77 vol.1.

[5] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon, "Automatic test case optimization: a bacteriologic algorithm," *IEEE Software*, 22(2), pp. 76 – 82, Apr. 2005.

[6] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," presented at the ISSTA, 2010, p. 147.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182 –197, Apr. 2002.

[8] P. Reales Mateo, M. Polo Usaola, and J. L. Fernández Alemán, "Validating 2nd-Order Mutation at System Level," *IEEE Transactions on Software Engineering*, pp. 1–1, 2012.