

Functional Search-based Testing from State Machines

Raluca Lefticaru, Florentin Ipaté
Department of Computer Science and Mathematics
University of Pitesti

Str. Targu din Vale 1, 110040 Pitesti, Romania
Email: raluca.lefticaru@gmail.com, florentin.ipate@ifsoft.ro

Abstract

The application of metaheuristic search techniques in test data generation has been extensively investigated in recent years. Most studies, however, have concentrated on the application of such techniques in structural testing. The use of search-based techniques in functional testing is less frequent, the main cause being the implicit nature of the specification. This paper investigates the use of search-based techniques for functional testing, having the specification in form of a state machine. Its purpose is to generate input data for chosen paths in a state machine, so that the parameter values provided to the methods satisfy the corresponding guards and trigger the desired transitions. A general form of a fitness function for an individual path is presented and this approach is empirically evaluated using three search techniques: simulated annealing, genetic algorithms and particle swarm optimization.

Keywords: search-based testing, genetic algorithms, simulated annealing, particle swarm optimization, finite state machine.

1 Introduction

Many papers in the field of software engineering present different aspects of state-based testing, like: generation of test cases for finite state machine (FSMs) [6, 12], coverage criteria for state machines [6, 19, 5, 28], generation of test cases from UML state diagrams and evaluation of their fault detection capability [19, 1, 20, 3]. Given a state machine diagram, it is possible to automatically derive a transition tree and obtain test sequences that accomplish certain criteria [20]. Briand et al. used test cases manually derived from UML state diagrams to evaluate their fault detection capability [1, 3], other authors obtained test cases using statistical functional testing [5] or with the aid of artificial intelligence methods [7]. Some tools, capable of generating actual test values automatically, also exist [19, 5, 20]. How-

ever, the process of developing such kind of tools is very complex and some limitations can appear. Although for FSMs the generation of test cases is automated, for UML state machine diagrams the transformation of function sequences into methods call sequences is hindered by the difficulty of choosing the input values to be provided to the methods.

The approach we present is to transform the data generation task for state machines into an optimization problem, which can be solved using metaheuristic search techniques. These have been successfully applied, mainly in the context of structural testing, for automatic generation of test data [25, 21, 23], for grey-box and non-functional property testing. In structural testing, the program is represented as a directed graph and the search-based techniques are used to generate test data to cover the desired graph elements (nodes, branches or paths). A comprehensive survey on the search-based data generation is presented in [14].

Functional search-based testing has been less investigated and the studies have concentrated on Z specifications [10, 24, 26] and conformance testing [26]: an objective function of the form pre-condition $\wedge \neg$ post-condition, that measures the "closeness" of the test data to uncovering a fault in the implementation was employed. In [27, 4] the application of evolutionary functional testing to some industrial software (for example an automatic parking system which could automate the parking procedure in future cars) is detailed.

Evolutionary methods for testing state-based programs have been studied, but only from a structural testing point of view, regarding the flags (loop-assigned flags), enumerations and counters used to manage the internal state of the objects [15, 16].

The paper is structured as follows. Section 2 provides a brief overview of metaheuristic search techniques. Section 3 presents the strategy used for functional search-based testing from state machines and some experimental results are resumed in section 4. Finally, conclusions and further work are drawn in section 5.

2 Metaheuristic Search Techniques

2.1 Simulated Annealing

Simulated annealing (SA) is a local search technique, proposed by Kirkpatrick et al. in 1983 [11]. The optimization starts with an initial candidate solution; a random generated "neighbour" replaces the current solution if it has a better objective value. The probability of acceptance of a worse candidate, usually $p = e^{-\frac{\delta}{t}}$ (where δ is the objective value difference between the current solution and the neighbouring inferior solution) is higher at the beginning of the algorithm, when the temperature t is also high and is gradually decreased according to a cooling schedule that controls the parameter t . This way, the dependency on the start position is lost and more search space is explored.

2.2 Genetic Algorithms

Genetic algorithms (GAs) [17, 18] are a class of *evolutionary algorithms*, that use techniques inspired from biology, such as selection, recombination (crossover) and mutation, applied on a population of potential solutions, called *chromosomes* (or *individuals*).

The *fitness (objective) function* assigns a score (fitness) to each chromosome in the current population, which depends on how close that chromosome to the solution of the problem is. Throughout this paper, the fitness is considered to be positive and finding a solution corresponds to minimizing the fitness function, i.e. a solution will be a chromosome with fitness 0.

Various mechanisms have been devised for selecting the individuals to be used to create offspring, based on their fitness, such as fitness-proportionate selection, sigma scaling, elitism, Boltzmann selection, tournament, rank and steady-state selection [18].

After the selection step, recombination takes place to form the next generation from parents and offspring. Single-point crossover, probably the best known form of recombination, randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two new offspring. Crossover is applied to individuals selected at random, with a probability (rate) p_c . Depending on this rate, the next generation will contain the parents or the offspring.

In the case of binary encoding, the mutation operator randomly flips some bits in a chromosome. In real encoding a random number is added to each vector entry of an individual. Mutation can occur at each bit position in a string with some probability p_m , usually very small [18]. This operator is responsible for introducing variation in the population.

Predicate	Objective function obj
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
Boolean	if <i>TRUE</i> then 0 else K
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$

Table 1. Tracey's objective functions

2.3 Particle Swarm Optimization

Particle swarm optimization (PSO) is a population based stochastic search technique developed by Eberhart and Kennedy in 1995, inspired by social metaphors of behavior and swarm theory. The system is initialized with a population of random solutions, called *particles*, which maintain their current position, velocity and best position explored so far. However, unlike in GAs, in PSO the selection and crossover operators are not employed: all the particles are kept as members of the population. The potential solutions fly through the problem space by following the current optimum particles. The velocity of a particle is updated according to its own previous best position and also to the best previous position of the entire population [22]. The particles are manipulated according the equations: $v_{id} = w * v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * Rand() * (p_{gd} - x_{id})$ and $x_{id} = x_{id} + v_{id}$, where: $X_i = (x_{i1}, x_{i2}, \dots, x_{iD})$ represents particle i , $V_i = (v_{i1}, v_{i2}, \dots, v_{iD})$ its velocity, $P_i = (p_{i1}, p_{i2}, \dots, p_{iD})$ its best previous position, g is the index of the best particle, $c_1, c_2 > 0$ are the acceleration coefficients, $rand()$, $Rand()$ random functions in the range $[0, 1]$ and w the initial inertia weight.

3 Fitness Function Design for Search-based Testing from State Machines

Given a particular path in a state machine, $p = S_1 \xrightarrow{f_1[g_1]} S_2 \xrightarrow{f_2[g_2]} S_3 \xrightarrow{f_3[g_3]} \dots S_m \xrightarrow{f_m[g_m]} S_{m+1}$, where f_i represent methods, S_i states and g_i conditions (for example guards in the state machine diagram), an *individual* (possible solution) is a list of input values, $x = (x_1, x_2, \dots, x_n)$, corresponding to all parameters of the methods f_1, f_2, \dots, f_m (in the order they appear). If the sequence of method calls, having the parameter values x_1, x_2, \dots, x_n , determines the transitions between the

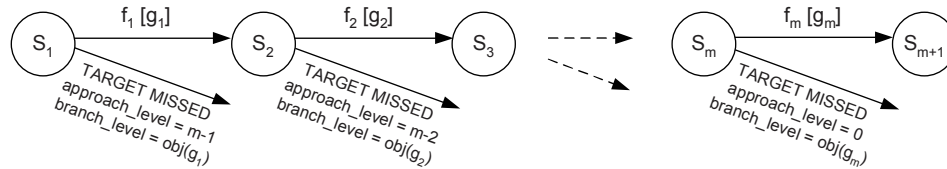


Figure 1. Calculating the fitness function

states specified by the path p and validates the predicate g_i of each transition, then x is a *solution* for p .

The transition constraints g_1, g_2, \dots, g_m are not necessarily the guards from the state machine diagram; they can be predicates obtained from stronger variants of state-based criteria, such as full predicate [20], in which all individual clauses in a decision are exercised, or disjunct coverage [3], in which guard conditions are transformed into disjunctive normal form and a separate transition is defined for each disjunct.

In order to employ a search technique to find a solution for a given path, a general fitness function was tailored in [13] and the preliminary experiments showed that its usage in the context of GAs is promising. The first component of this fitness function has a similar metric in evolutionary structural test data generation, namely the *approach level*. This is calculated by subtracting one from the number of critical branches lying between the node from which the individual diverged away from the target node, and the target itself. In this approach, a critical branch is a program branch which leads to a miss of the current structural target for which test data is sought [16]. For state-based testing we consider all the transitions to be critical and for the transitions without a guard, the *true* predicate as pre-condition.

An individual has missed the target path if at some point a pre-condition is not satisfied (this could cause a transition to another state). The *approach_level* is 0 for the individuals which follow the target path; for those which diverge from the path, it is calculated as described in Fig. 1.

A fitness function using only the approach level has plateaux (for each value $0, 1, \dots, m - 1$) and consequently it does not offer enough guidance to the search. Therefore a second component of the fitness function will compute, for the place where the actual path diverges from the required one, how close the pre-condition predicate to being true was. This second metric is called *branch level* and can be derived from the guard predicates using the transformations from Table 1 [24, 26, 14], where K is a positive failure constant. The branch level is then mapped onto $[0, 1)$ - it is said to be *normalized*. Thus the fitness value is: $fitness = approach_level + normalized_branch_level$. Fig. 2 presents two fitness landscapes for n -dimensional individuals, having $n - 2$ dimensions fixed.

4 Empirical Evaluation

The fitness function presented was employed to generate test data for Java classes, having state machine specifications with transition guards described in terms of algebraic predicates. The results obtained when GAs were used in conjunction with this kind of fitness function are reported in [13] for two cases: (1) generating numerical input values for given paths; (2) conformance testing. The results, correlated in the first case with the number of solutions from the search space, showed that for the classes considered GAs had a good convergence rate (note that the recombination operator used was heuristic crossover [17], which guided the generation of the offspring, by comparing the parents fitness values, in the direction of the fitter parent). The second experiment used mutation testing to evaluate the capacity of a fitness function of the form "pre-condition $\wedge \neg$ post-condition" to detect faults in the implementation and the results were also promising. An open issue in this case is an increased number of generations needed to discover some mutants, due to the fact that the fitness function landscape contained plateaux.

Additional experiments were performed for the state-based approach from section 3, using 30 path-corresponding landscapes, with different difficulties, in order to evaluate the metaheuristic search techniques effectiveness and efficiency. The Matlab toolbox for genetic algorithms and direct search [9] was used for SA and GAs; for PSO, the experiments were performed with the particle swarm optimization toolbox (PSOt) [8], presented in [2]. The same maximum number of function evaluations was set for the three metaheuristic search techniques (when solving the same problem) and several trials were performed to investigate their tuning choices in order to detect the combinations of parameter settings which optimized each search process.

The preliminary results, averaged after 100 runs for each test object, showed that the best results, measuring the effectiveness (finding a global minimum) and efficiency (a smaller number of function evaluations) were obtained by GAs and PSO. A particular observation is that for more complex landscapes, having more local minima (like the second landscape from Fig.2), PSO outperformed GAs and

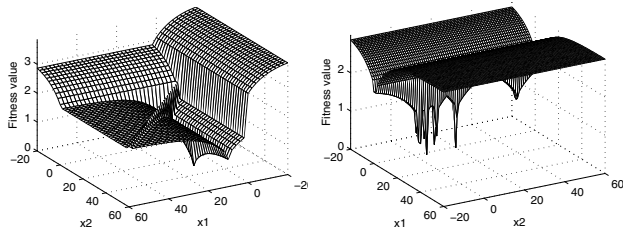


Figure 2. Fitness function landscapes

the difference was statistical significant for 17 out of 21 test objects (confidence 95%), while for simpler function, having only one minimum (the first landscape from Fig.2) GAs achieved better results (9 out of 9 cases). Also, hybridizing SA or GAs with local search techniques improved their effectiveness. However the evaluation is still in progress and a larger benchmark of test objects should be used for comparison.

5 Conclusions and Future Work

This paper presents an approach for automatic generation of test data, using state diagrams and metaheuristic search techniques. Furthermore, experimental evidence show that the test data obtained can cover difficult paths in the machine and a slightly different design of the fitness function can be used for specification conformance testing.

Future work concerns analyzing, on a larger benchmark of classes, the effectiveness and the efficiency of several search techniques, finding categories of problems for which certain search algorithms achieve better results and studying the influence of the fitness landscape.

Further directions are the possibility of extending the strategy presented for multi-class testing and the derivation of the fitness function from hierarchical and concurrent state machine diagrams.

References

- [1] G. Antonioli, L. C. Briand, M. D. Penta, and Y. Labiche. A case study using the round-trip strategy for state-based class testing. In *ISSRE '02*. IEEE Computer Society, 2002.
- [2] B. Birge. PSOT - a particle swarm optimization toolbox for use with Matlab. In *IEEE Swarm Intelligence Symposium*, pages 182 – 186, 2003.
- [3] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Softw. Eng.*, 30(11):770–793, 2004.
- [4] O. Bühler and J. Wegener. Evolutionary functional testing. *Computers and Operations Research*, 2007. doi: 10.1016/j.cor.2007.01.015.
- [5] P. Chevalley and P. Thévenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *COMP-SAC'01*, pages 205–214. IEEE Computer Society, 2001.
- [6] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.
- [7] P. Fröhlich and J. Link. Automated test case generation from dynamic models. In *ECOOP'00*, pages 472–492. Springer-Verlag, 2000.
- [8] <http://psotoolbox.sourceforge.net/>. PSOToolbox.
- [9] <http://www.mathworks.com/products/gads/>. Genetic Algorithm and Direct Search Toolbox 2.2.
- [10] B. Jones, H. Sthamer, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of 3rd International Conference on Software Quality Management*, pages 435–444, 1995.
- [11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
- [13] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *SYNASC 2007*, pages 188–195, 2007.
- [14] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
- [15] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *GECCO*, pages 2488–2498, 2003.
- [16] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *GECCO*, pages 1013–1020, 2005.
- [17] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, 1996.
- [18] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.
- [19] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML*, pages 416–429, 1999.
- [20] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.
- [21] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Softw. Test., Verif. Reliab.*, 9(4):263–282, 1999.
- [22] Y. Shi and R. Eberhart. Empirical study of particle swarm optimization. In *CEC*, pages 1945–1950, 1999.
- [23] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.
- [24] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98*, pages 73–81. ACM Press, 1998.
- [25] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *ASE*, pages 285–288, 1998.
- [26] N. J. Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York, 2000.
- [27] J. Wegener and O. Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *GECCO (2)*, pages 1400–1412, 2004.
- [28] Y. Wu, M.-H. Chen, and J. Offutt. UML-based integration testing for component-based software. In *ICCBSS*, pages 251–260, 2003.