Third International Conference on Software Testing, Verification, and Validation Workshops

Generating Feasible Test Paths from an Executable Model Using a Multi-Objective Approach

Thaise Yano, Eliane Martins
Institute of Computing
State University of Campinas, UNICAMP
Campinas, SP, Brazil
tyano,eliane@ic.unicamp.br

Fabiano L. de Sousa
Space Mechanics and Control Division
National Institute for Space Research, INPE
São José dos Campos, SP, Brazil
fabiano@dem.inpe.br

***Abstract*—Search-based testing techniques using meta-heuristics, like evolutionary algorithms, has been largely used for test data generation, but most approaches were proposed for white-box testing. In this paper we present an evolutionary approach for test sequence generation from a behavior model, in particular, Extended Finite State Machine. An open problem is the production of infeasible paths, as these should be detected and discarded manually. To circumvent this problem, we use an executable model to obtain feasible paths dynamically. An evolutionary algorithm is used to search for solutions that cover a given test purpose, which is a transition of interest. The target transition is used as a criterion to get slicing information, in this way, helping to identify the parts of the model that affect the test purpose. We also present a multi-objective search: the test purpose coverage and the sequence size minimization, as longer sequences require more effort to be executed.**

***Keywords*-model-based testing; feasible path; executable model; multi-objective optimization**

I. Introduction

Most of the search-based testing approaches consider the generation of test data to cover structural aspects of a program. However applying search-based techniques, like evolutionary algorithm (EA), to generate test sequence from specification models has been growing [1], [2], [3].

In this study we consider the system behavior is represented by an Extended Finite State Machine (EFSM). Test generation from EFSM using EA presents some challenges. Firstly, a sequence need to be generated to put the machine in a desired state, in order to cover a given target. In other terms, it is necessary a transition path from the initial state to the target. A second point is that input events, in general, are not single data. Like programming language functions, events in a transition have a name and various parameters, which can have different types. The loss of information is also another problem by the fact that the context variables in an EFSM are not controlled by an EA [4]. Finally, an undecidable problem for most model-based testing approaches is the generation of feasible paths. Most of the existing methods are based on a reachability analysis of the graph representing the structure of the model. Since data constraints are not taken into account, many paths are not semantically feasible, although structurally valid.

In this paper, we present an approach to derive test sequences from an EFSM using an EA. We extend the previous work [5] in order to generate sequences that take into consideration the control and the data flow of an EFSM. The test generation aims to satisfy a given test purpose [6], which in particular specifies a transition to be covered. The main objective of the approach is to address the aforementioned challenges. To reduce the loss of information problem, the objective function uses the dependency analysis to guide the search. To avoid the generation of infeasible paths, we use an executable version of the model. Executable modeling is useful for validating requirements, as the model can be tested as a prototype. By instrumenting the executable model, it is possible to monitor its execution with an input sequence to determine whether a suitable path has been taken. Suitable means a path that covers the test purpose. To generate the input sequence, we use a multi-objective optimization based on the Pareto optimality named as M-GEO$_{vsl}$ (Multi-Objective Generalized Extremal Optimization with variable string length). The main features of M-GEO$_{vsl}$ are: $i.$ the population has variable lengths to determine the minimal sequence size automatically and $ii.$ the search space contains the input events as well as their parameters.

The paper is organized as follows. Section II describes the steps of the test sequence generation approach. Section III shows an example of the approach application. Section IV remarks some conclusions and future works.

II. The Test Sequence Generation Approach

Figure 1 shows the steps of the test sequence generation approach from a model.

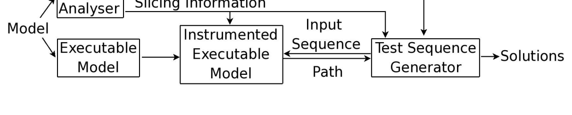

Figure 1. Test sequence generation process

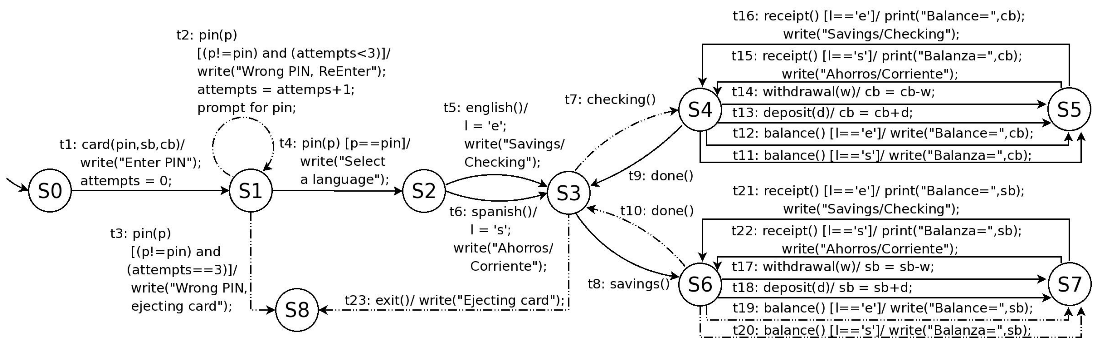
t16: receipt() [l=='e']/ print("Balance=",cb); write("Savings/Checking");
t15: receipt() [l=='s']/ print("Balanza=",cb); write("Ahorros/Corriente");
t14: withdrawal(w)/ cb = cb-w;
t13: deposit(d)/ cb = cb+d;
t12: balance() [l=='e']/ write("Balance=",cb);
t11: balance() [l=='s']/ write("Balanza=",cb);
t2: pin(p) [(p!=pin) and (attempts<3)]/ write("Wrong PIN, ReEnter"); attempts = attemps+1; prompt for pin;
t7: checking()
t5: english()/ l = 'e'; write("Savings/ Checking");
t1: card(pin,sb,cb)/ write("Enter PIN"); attempts = 0;
t4: pin(p) [p==pin]/ write("Select a language");
S0
S1
S2
S3
S4
S5
t9: done()
t10: done()
t6: spanish()/ l = 's'; write("Ahorros/ Corriente");
t3: pin(p) [(p!=pin) and (attempts==3)]/ write("Wrong PIN, ejecting card");
t21: receipt() [l=='e']/ print("Balance=",sb); write("Savings/Checking");
t22: receipt() [l=='s']/ print("Balanza=",sb); write("Ahorros/Corriente");
t17: withdrawal(w)/ sb = sb-w;
t18: deposit(d)/ sb = sb+d;
t19: balance() [l=='e']/ write("Balance=",sb);
t20: balance() [l=='s']/ write("Balanza=",sb);
t8: savings()
S6
S7
S8
t23: exit()/ write("Ejecting card");

Figure 2. EFSM for an ATM system [7]. Dashed arrows represents the critical transitions with respect to t_{18}

The *model analyser* produces slicing information, that is, the parts of the model that affect the transition of interest. The target transition is given by the test purpose. A *test purpose* defines specific parts of a system to be tested. We use the notation proposed by Hwang *et al.* [6] to describe a test purpose as a set of conditions. Each condition, in fact, indicates one target transition. A condition is a conjunction of constraints related to: state (source or target of a transition), input event, action and variable (conditions on variable values). The test purpose TP which corresponds to the transition t_{18} of the ATM (Automated Teller Machine) EFSM defined by Korel *et al.* [7] (Figure 2) is shown as follows:

$$
\begin{aligned}
TP &= \{cond_1\}\\
cond_1 &= constraint_1 \wedge constraint_2 \wedge \cdots \wedge constraint_\mu\\
constraint_1 &= state : source = S_6\\
constraint_2 &= state : target = S_7\\
constraint_3 &= input : deposit(d)\\
constraint_1 &= action : addSb(d)
\end{aligned}
$$

We use a backward slicing to obtain the transitions that influence the target transition given by the TP. The slicing criterion for EFSM can be defined as a pair $SC = < t, V_{sc} >$ where t is the transition corresponding to TP and V_{sc} the variable set in TP. The model slice M' contains all transitions and states that could potentially affect the values of V_{sc} at t. Model slicing is still an open research problem, but a discussion about this subject is out of the scope of this paper. For the example, we consider an algorithm proposed in the literature [8]. Figure 3 shows the model slice M' of the ATM example with the transition t_{18} as slicing criterion.

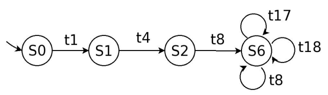

Figure 3. Model slice of the ATM EFSM

Using the model slice, we identify the *critical transitions*, which are the transitions that lead to a miss of the test purpose, including the transitions within a cycle. A transition is critical when it diverges away from the transitions of the slice M'. In the ATM example, the critical transitions are indicated with dashed arrows in Figure 2.

The *executable model* implements the behavior of the model M in a programming language, like Java, that takes the generated test sequence as input and produces a transition path triggered by the sequence. The executable model only triggers a transition when the associated guard is satisfied, considering the involved input parameters and/or variables in the condition. As a result, only feasible paths are produced. The executable model is instrumented in order to verify the transitions triggered during execution, especially whether the critical transitions is traversed.

The *test sequence generator* takes into account the slicing information and the triggered transitions to evaluate the solution. To generate the test sequence, we propose the evolutionary algorithm M-GEO$_{vsl}$, which consists of a multi-objective implementation of GEO [9] based on the Pareto optimality. Each solution is non-dominating which means that it cannot be improved in any objective without causing a degradation in at least one other objective. Differently from M-GEO [10], M-GEO$_{vsl}$ allows the number of design variables to change during the evolution process, in order to handle test sequences with different sizes. Another difference with M-GEO is the use of discrete encoding of the design variables, instead of binary values. Each design variable of the population corresponds to a species and for each of them is associated a fitness value. The population is compounded of three parts (Figure 4): i. test sequence size, ii. sequence of input events, and iii. parameters of all input events. The sequence size as a design variable allows to generate sequences with different numbers of input events. The parameters are also generated during the evolution process, and not afterwards.

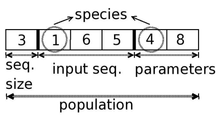

Figure 4. Population in M-GEO$_{vsl}$

The main steps of M-GEO$_{vsl}$ are presented in Figure 5. In first step of M-GEO$_{vsl}$, the population of the design variables X is initialized randomly with uniform distribution according to their domain. We assume that the parameters with identical name are the same, in this way the number of parameters is constant. In the second step, each design variable i is mutated to other value mut_i of the variable domain, one at a time, generating X'_i. All objective functions evaluate X'_i, calculating $[F_1(X'_i), F_2(X'_i), \ldots, F_{nof}(X'_i)]$, where nof is the number of objective functions. After this evaluation, the value of the variable i returns to its original one. This process is repeated to all variables. As the sequence size is one design variable, the algorithm should deal with two situations when this variable is mutated: the new value of sequence length is bigger or lower than the current value. When the new value is bigger than the current one, the sequence is completed with random values (Figure 6). In the other case, the extra input events in the sequence are ignored. Also in this step, the Pareto front and Pareto optimal set are updated and saved. In the third step, one objective function F_c is randomly chosen from all objective functions. Once F_c is chosen, for each variable i is associated a fitness value, given by $\Delta_i = F_c(X'_i) - ref$, where ref is a given value of reference (*e.g.*, zero). In the next step, all variables are ranked by their fitness value. The first position ($k = 1$) of the ranking belongs to the least adapted variable. For a minimization problem, the lowest value of Δ_i means $k = 1$. Since GEO has only the mutation operator, a variable j is selected to be mutated according to the probability distribution $P \sim k^{-\tau}$, where k is the rank of the variable j and τ the only adjustable parameter in the algorithm. The stop condition is the maximum number of evaluations of all objective functions. If this condition is not achieved, the algorithm verifies whether a new independent execution should be started. In the affirmative case, the algorithm returns to the first step and a new point in the search space is found, but the Pareto front remains the same. Otherwise, the algorithm returns to second step and begins a new iteration.

```
$initializePopulation()$
**while** **not** $stopCondition$ **do**
    $mutatePopulation()$
    $updateParetoOptimal()$
    $selectObjectiveFunction()$
    $calculateFitness()$
    $rankFitness()$
    $mutateSpecies()$
    **if** (**not** $stopCondition$)**and**$independentExecution$ **then**
        $initializePopulation()$
    **end if**
**end while**
**return** $paretoOptimal$
```

Figure 5. M-GEO$_{vsl}$ algorithm

For test sequence generation, M-GEO$_{vsl}$ uses two objective functions: to cover the test purpose (F_1) and to minimize the test sequence size (F_2). The objective function F_1 is

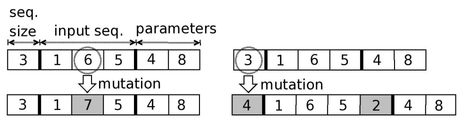

Figure 6. Mutations in M-GEO$_{vsl}$

inspired by the evolutionary structural testing [11] in order to direct the search towards the test purpose. To reduce the cost of testing, the objective function F_2 intends to minimize the test sequence size. A value into the range [0,1] is added to sequence size in the sense of penalizing unexpected inputs of the sequence. Unexpected inputs are the inputs not specified in a given state.

$$\begin{aligned} F_1 &= AL + ND \\ F_2 &= \mathit{sequence\_size} + (1 - 1.001^{unexpected\_inputs}) \end{aligned}$$

where

$$\begin{aligned} AL &= t\_slice - triggered\_t\_slice \\ ND &= 1 - 1.001^{-d} \end{aligned}$$

The objective function F_1 is calculated in terms of the approach level AL and normalized distance ND. The approach level AL measures how close an input sequence is to reach a path that traverses the test purpose. AL is calculated based on the slice information, and is given by the number of transitions in the slice M' that were triggered during model execution. The normalizing distance ND [12] is calculated at the point where the control flow takes a critical transition t_c, that diverges away from a transition t_i. In an EFSM, a state can be the source of many transitions; therefore two situations need to be considered for the calculation of the term d in ND: i. the input event of t_c and t_i are the same but the guards are different and ii. the input event of t_c and t_i are different. In the first case, the distance d is computed using the functions defined by Tracey *et al.* [13]. For example, if a guard transition $(x == y)$ needs to be evaluated as true to reach the test purpose, the distance function is defined as $abs(x - y) + K$ for transitions with the same input event. The value K is a non-zero positive constant which is always added if the term is not true. In the second case, taking t_c receives a high penalty in order to distinguish solutions with different input events.

III. Example of the Approach Application

In order to illustrate the approach, we take the ATM system already presented and consider the transition t_{18} as the test purpose TP. The executable model of the ATM EFSM is generated by the SMC[1] (State Machine Compiler) tool. Given a description of an EFSM, the SMC generates a source code of the model in Java.

[1]Available in http://smc.sourceforge.net.

Table I
GENERATED DATA FOR THE ATM EFSM

F_1	F_2	Sequence size	Input sequence	Parameters						Path
4.0	1.0	1	0	243	421	217	233	20	979	t_1
3.0	4.002	4	10 0 1 0	704	947	113	704	232	31	t_1t_4
2.0	5.001	5	0 1 6 2 6	5	867	624	5	876	142	$t_1t_4t_5t_8$
1.0	9.003	9	0 1 10 3 6 0 1 10 11	442	873	12	442	942	694	$t_1t_4t_6t_8t_{17}t_{22}$
1.632	9.001	9	0 1 10 3 6 7 6 9 11	442	686	12	442	924	654	$t_1t_4t_6t_8t_{10}t_8t_{18}t_{22}$
0.0	22.015	22	0 9 9 7 1 1 4 1 2 3 6 11 0 0 1 9 4 10 11 4 6 10	292	970	430	292	477	74	$t_1t_4t_5t_8t_{18}t_{21}t_{17}$

M-GEO$_{vsl}$ was performed with: tau value of 3.75, 1.000.000 function evaluations and 100 independent executions. The Table I shows the generated points of the Pareto front and the respective sequence size, input sequence, parameters (pin, sb, cb, p, w and d) and transition path. Note that the last two lines of the table are solutions that reaches the test purpose TP. The fifth line represents the sequence $seq = \{card(442, 686, 12)$, $pin(442)$, $withdrawal(924)$, $spanish()$, $savings()$, $done()$, $savings()$, $deposit(654)$, $receipt()\}$. The transition path triggered by seq is $path = \{t_1t_4t_6t_8t_{10}t_8t_{18}t_{22}\}$. When the machine receives an unexpected input event, the input event is ignored and the machine remains in the current state.

IV. CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented an evolutionary approach for test sequence generation from an EFSM. A solution for the path feasibility problem has been proposed by using an executable model. We use a multi-objective evolutionary algorithm, M-GEO$_{vsl}$, that can consider two objectives: to search for a test sequence that covers a target transition, as well as to minimize the length of this test sequence. To guide the search for a solution, we used model slicing. A limitation of the approach, that we have already considered as a next step, is to cope with the situation in which no slice can be found for the model. Also, further experiments have being carried out to validate the proposed approach.

ACKNOWLEDGMENT

This research is supported by CAPES, CNPq and Serasa Experian.

REFERENCES

[1] A. Baresel, H. Pohlheim, and S. Sadeghipour, “Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms,” in *Proc. GECCO’03*, 2003, pp. 2428–2441.

[2] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo, “Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms,” in *Proc. GECCO’05*. Washington DC, USA: ACM, Jun. 2005, pp. 1081–1082.

[3] A. S. Kalaji, R. M. Hierons, and S. Swift, “Generating feasible transition paths for testing from an extended finite state machine (efsm),” in *Proc. ICST’09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 230–239.

[4] P. McMinn and M. Holcombe, “Evolutionary testing of state-based programs,” in *Proc. GECCO’05*. New York, NY, USA: ACM, 2005, pp. 1013–1020.

[5] T. Yano, E. Martins, and F. L. De Sousa, “An evolutionary approach for robustness testing,” in *Proc. Int. Conf. on Evolutionary Computation (ICEC’09)*, 2009, pp. 277–280.

[6] I. Hwang, M. Lallali, A. R. Cavalli, and D. Verchère, “Modeling, validation, and verification of pcep using the if language,” in *Proc. Formal Techniques for Distributed Systems (FMOODS/FORTE)*, ser. Lecture Notes in Computer Science, vol. 5522. Springer, 2009, pp. 122–136.

[7] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, “Slicing of state-based models,” in *Proc. Int. Conf. on Software Maintenance (ICSM’03)*, Washington, DC, USA, 2003, p. 34.

[8] K. Androutsopoulos, D. Binkley, D. Clark, and M. Harman, “Slicing extended finite state machines,” Department of Computer Science, King’s College London, Tech. Rep. TR-08-05, July 2008.

[9] F. L. De Sousa, F. M. Ramos, P. Paglione, and R. M. Girardi, “New stochastic algorithm for design optimization,” *AIAA Journal*, vol. 41, no. 9, pp. 1808–1818, Sep. 2003.

[10] R. L. Galski, “Development of improved, hybrid, parallel and multiobjective versions of the generalized extremal optimization method and its application to the design of spatial systems,” Ph.D. dissertation, INPE, São José dos Campos, SP, Brazil, 2006, (in Portuguese).

[11] P. McMinn, “Search-based software test data generation: a survey,” *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[12] A. Baresel, H. Sthamer, and M. Schmidt, “Fitness function design to improve evolutionary structural testing,” in *Proc. GECCO’02*, 2002, pp. 1329–1336.

[13] N. Tracey, J. Clark, J. McDermid, and K. Mander, “A search-based automated test-data generation framework for safety-critical systems,” *Systems engineering for business process change: new directions*, pp. 174–213, 2002.