Finite State Machine Induction using Genetic Algorithm Based on Testing and Model Checking

Fedor Tsarev
St. Petersburg State University of IT, Mechanics and Optics
Russia, St. Petersburg, Kronverksky pr., 49
+7 (812) 232-43-18
tsarev@rain.ifmo.ru

Kirill Egorov
St. Petersburg State University of IT, Mechanics and Optics
Russia, St. Petersburg, Kronverksky pr., 49
+7 (812) 232-43-18
egorovk@rain.ifmo.ru

ABSTRACT

In this paper, we describe the method of finite state machine (FSM) induction using genetic algorithm with fitness function, cross-over and mutation based on testing and model checking. Input data for the genetic algorithm is a set of tests and a set of properties described using linear time logic. Each test consists of an input sequence of events and the corresponding output action sequence. In previous works testing and model checking were used separately in genetic algorithms. Usage of such an approach is limited because the behavior of system usually cannot be described by tests only. So, additional validation or verification is needed. Calculation of fitness function based only on verification do not perform well because there are very few possible values of fitness function (verification gives only “yes” or “no” answer). The approach described is tested on the problem of finite state machine induction for elevator doors controlling. Using tests only the genetic algorithm constructs the finite machine working improperly in some cases. Usage of verification allows to induct the correct finite state machine.

Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Automatic Programming – p*rogram synthesis, program verification.*

General Terms

Algorithms, Experimentation, Verification.

Keywords

Genetic algorithm, model checking, finite state machine, automata-based programming, testing.

1. INTRODUCTION

In genetic programming, fitness function is usually evaluated by running the program on a number of test cases. Such an approach is connected with a number of difficulties. The main of them is that it is impossible to guarantee the behavior of the system on the test-cases not included into the training set. So, programs created using genetic programming or genetic algorithms based on tests cannot be used without additional testing or verification. If during this testing or verification some faults are found, program or test cases should be modified manually. Usually, it is not important, but in mission-critical systems this situation creates additional barrier to usage of such methods.

One of the solutions to the problem described is the usage of program verification techniques on the stage of fitness function evaluation. A practical method of verification is model checking [1, 2]. Usage of this method implies that the program should be transformed into a finite state model (so called Kripke model) and properties to be verified should be written in temporal logic. The result of verification is either the counterexample or the confirmation of the fact that properties hold.

For traditionally designed programs transformation of program to model and of counterexample from terms of model to terms of program can be rather difficult. These difficulties do not arise if the program is designed using automata-based programming [3, 11, 12]. Features of automata-based programs allow for automatically converting them into models which can be used for model checking. Counterexamples can be also converted automatically from the terms of model to the terms of program.

2. AUTOMATA-BASED PROGRAMMING

Automata-based programming is the programming paradigm in context of which it is proposed to design and implement the software system as a system of interacting automated controlled objects. Each automated controlled object consists of a finite state machine (FSM) and a controlled object (see Figure 1).

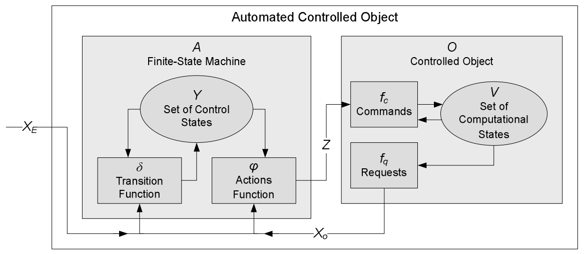

Figure 1. Automated controlled object.

FSM has a set of states, a transition function and an actions function. Controlled object has commands and requests (implemented by its methods) and a set of computational states.

FSM takes events and input variables as input. They can come from other parts of the system as well as from the controlled object. After receiving an event or an input variable the FSM makes transition on which some output action is sent to controlled object. Output action can also be performed on entering the state. Output actions can change the computational state of the controlled object.

The main idea of automata-based programming is to distinguish control states and computational states. The number of control states is not large so they can be drawn on transition graph, each of them differs qualitatively from others and they define actions. The number of possible computational states can be very large (and even infinite), they differ from each other quantitatively and define only results of actions but not actions themselves.

In this paper, we focus on automata-based programs with only one automated controlled object. We suppose that the controlled object, events and output actions are predefined and our task is to design the FSM.

3. FSM INDUCTION

Induction of FSMs with genetic algorithms has been studied by several researchers. In [8] two approaches to induction of FSMs from examples were compared. One of them is Evidence-Driven State Merging (EDSM) and another one is based on evolutionary algorithms. FSM is represented using the table of transitions. Lucas and Reynolds take into consideration only automata over binary alphabet, so, the total number of finite state machines with n states is equal to n^{2n}. The total size of search space is $2^n \cdot n^{2n}$ because each state can be either accepting or non-accepting.

To reduce the size of search space Lucas and Reynolds proposed a “smart state labeling” algorithm to determine which states of FSM are accepting and which are not. The size of reduced search space is equal to n^{2n}. Comparison of (1+1) evolutionary strategy with EDSM shows that it outperforms EDSM when the number of states is relatively small.

In [9] Lucas and Reynolds develop their approach further. They propose two new methods of mutation which they called sampled and quick-sampled mutation. They compared four algorithms – plain (which do not use state labeling), smart state labeling, sampled and quick-sampled. Results of experiments showed that sampled algorithm has the best performance.

In [10] Lucas applied (1+1) evolutionary strategy to induction of finite state transducers from the set of tests. Lucas and Reynolds used three types of fitness function: based on strict comparison of strings, based on Hamming distance [4] and based on edit distance (Levenshtein distance) [7]. Experiments showed that the edit distance function shows the best performance.

In [6] Johnson used verification (model checking) for fitness evaluation. He used (1+λ) evolutionary strategy as optimization method and computational tree logic to represent temporal properties. Input data for the evolutionary algorithm consisted of a set of temporal properties. To calculate the value of fitness function model was checked against each of the properties and the result is the number of properties which are true.

4. VERIFICATION OF AUTOMATA-BASED PROGRAMS

To describe requirements for automata-based program we use linear temporal logic (LTL) language. Time in this logic is considered to be discrete and linear. LTL syntax contains propositional variables *Prop*, Boolean operators (and, or, not) and temporal operators. Temporal operators (**X** – ne**X**t, **F** – in the **F**uture, **G** – **G**lobally in the future, **U** – **U**ntil, **R** – **R**elease) are used to compose statements about future.

Verifier which we use in this paper takes as input the model to be verified and an LTL-formula. The output of verification process is either the counterexample (a path in a Kripke model) or the confirmation of that the property holds for the model. This verifier is described in [2] and [13].

5. GENETIC ALGORITHM

Input data for the algorithm consists of the set of possible events; the set of possible output actions; the set of tests (is denoted by Tests, each test consists of the sequence of input events Input[i], and the corresponding sequence of output actions Answer[i]); the set of LTL-formulae describing the requirements for the finite state machine; the number N_s of states of FSM; the desired number N_t of FSM transitions.

In ideal case the output of the algorithm is the FSM with N_s states and N_t transitions that passes all tests and satisfies all LTL-formulae. In non-ideal case this FSM may have more transitions or pass not all tests or satisfy not all LTL-formulae.

The initial population consists of N randomly generated FSMs with N_s states. The main reproduction strategy is elitism – individuals are sorted in descending order according to their fitness function values and part of most fit goes directly to the next generation. After that the following process is repeated until next generation size becomes equal to N. Two individuals from the current generation are selected and then mutation or cross-over is applied to them. Both individuals-results of this operation are added to the next generation.

6. FITNESS FUNCTION EVALUATION

Fitness function evaluation is based on running the FSM on all tests and checking the finite state machine against all LTL-formulae. Each sequence Input[i] is given as input to the FSM and the obtained sequence of output actions Output[i] is recorded. After that the value FF_1 is calculated:

$$\mathrm{FF}_1 = \frac{\sum_{i=1}^{n} \left(1 - \frac{\mathrm{ED}(\mathrm{Output}[i], \mathrm{Answer}[i])}{\max(|\mathrm{Output}[i]|, |\mathrm{Answer}[i]|)}\right)}{n}$$

Here by ED(A, B) is denoted the edit distance. Note that the value of FF_1 is always between 0 and 1 and greater values correspond to better compliance of finite state machine with tests. To take the number of transitions into account the following value is calculated:

$$\mathrm{FF}_2 = T \cdot \mathrm{FF}_1 + \frac{1}{M} \cdot (\mathrm{M} - \mathrm{cnt})$$

Here by cnt the number of transitions in finite state machine is denoted, by T is denoted the “cost” of passing all tests (in experiments T is equal to 100), M is an arbitrary number greater that the number of possible transitions in FSM with N states and given set of events (in experiments M is equal to 100).

Such a structure of fitness function means that between two finite state machines behaving identically on tests the finite state machine with fewer transitions will have the greater value of FF_1. Also the finite state machine that passes ideally all tests will have the greater value of fitness function than the finite state machine that does not pass one of the tests. Fitness depends on the number of transitions because its minimization forces to delete the transitions not used in tests. It its minimization partially solves the over-fitting problem because with fewer transitions has more "general" behavior.The final value FF of fitness function is calculated using results of LTL-formulae checking:

$$\mathrm{FF} = \mathrm{FF}_2 + F \cdot \frac{n_1}{n_2}$$

Here by F is denoted the "cost" of "passing" all formulae (equal to 10 in experiments), n_1 is the number of LTL-formulae which are true for the FSM and n_2 is the total number of LTL-formulae.

7. INDIVIDUAL REPRESENTATION

Individual representation contains three parts: the number of states; the number of initial state; an array containing descriptions of states. Description of each state is the array of descriptions of outgoing transitions. Each transition has three fields: the event associated with this transition; the number of state which this transition goes to; the number of output actions on it. Output actions themselves are not encoded in the individual. They are determined using transitions labeling algorithm. For example, the individual given on Figure 2 has the following representation: {2, 0, {{A, 1, 0}, {T, 1, 1}}, {{T, 1, 1}, {M, 0, 2}}}.

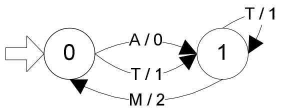

Figure 2. Example of the individual.

Transition labeling is done analogically to the state labeling algorithm from [8]. The transition is labeled with a sequence of actions which occurs most frequently on it in tests. Formally, for each transition T and each sequence of output actions zs we calculate $C[T][\mathrm{zs}]$ – the number of times sequence of output actions zs should be generated on transition T in tests. After that each transition is labeled with the sequence z for which the value $C[T][z]$ is maximal.

8. CROSS-OVER

Let us denote "parent" FSMs as P1 and P2 and "offspring" FSMs as S1 and S2. For initial states S1.is and S2.is one of the following will be true: S1.is = P1.is and S2.is = P2.is; or S1.is = P2.is and S2.is = P1.is. Since all FSMs have the same number of states cross-over is performed for each state number separately. Let us denote the list of transitions from state i of FSM A as $A.T[i]$. The "transitions cross-over" can be performed using one of the methods. First of them is "traditional cross-over":

1. A list containing all transitions from both P1.$T[i]$ and P2.$T[i]$ is constructed.
2. A random permutation is applied to this list.
3. One the following variants is chosen at random:
    a. S1.$T[i]$ will contain first |P1.$T[i]$| elements of the list and S2.$T[i]$ – remaining elements (by |L| is denoted length of list L).
    b. S1.$T[i]$ will contain first |P2.$T[i]$| elements of the list and S2.$T[i]$ – remaining elements (by |L| is denoted length of list L).

Second method is "test-based cross-over":

1. In FSMs P1 and P2 we mark transitions which are used in processing 10% of tests for which the difference $\frac{\mathrm{ED}(\mathrm{Output}[i], \mathrm{Answer}[i])}{\max(|\mathrm{Output}[i]|, |\mathrm{Answer}[i]|)}$ between the output and answer is minimal. This step is done during the calculation of fitness function.
2. Transitions *marked* during the previous step are copied to S1.$T[i]$ (from P1.$T[i]$) and to S2.$P[i]$ (from P2.$T[i]$).
3. A list L containing all *non-marked* transitions both from P1.$T[i]$ and P2.$T[i]$ is constructed.
4. A random permutation is applied to this list.
5. List S1.$T[i]$ is appended with first transitions from list L until reaches the size |P1.$T[i]$|. All remaining transitions are added to the list S2.$T[i]$.

Results of verification are also used on the stage of cross-over. If there is at least one LTL-formula that does not hold for the FSM then the longest counter-example is taken. If during the cross-over the transition from this counter-example is processed and this transition is not marked (in test-based crossover) then with probability of 10% is changed the number of output actions associated with this transition and the number of state to which this transition goes. After both traditional and test-based cross-over duplicated transitions deletion is applied to FSMs S1 and S2.

After the cross-over the finite state machine can have two transitions from one state with the same event (duplicated transitions). In order to delete them the following operations are done: the list of transitions is scanned and whether a transition with event which is already used encounters it is deleted.

9. MUTATION

To perform the mutation each of the following operations is applied to the finite-state machine with the probability equal to 0.05: change of initial state to randomly chosen one; change of each transition; addition or deletion of transition for each state. During the mutation results of verification are used in the similar way as during the cross-over. That means that if it is chosen to delete a transition from some state then if there is a transition from the counter-example it is deleted. Otherwise, the transition is chosen at random.

To change the transition one of following operations is chosen with equal probabilities: change of the state to which the transition leads by the randomly chosen; change of the event associated with the transition by the randomly chosen; change of the number of output actions associated with this transition – it is either increased or decreased by one, but cannot become negative or exceed MA (in experiments MA=3). Also if a transition belongs to the counter-example it is changed in the similar way as during cross-over. After the mutation duplicated transition deletion (described in the previous section) is applied to the finite state machine.

10. RESULTS

The genetic algorithm described in this paper was on problem of designing finite state machine controlling the doors of elevator. In this problem there are five events: e11 – button "Open the doors" pressed, e12 – button "Close the doors" pressed, e2 – doors are successfully opened, e3 – an obstacle prevents doors closing, e4 – doors jammed. There are also three output actions: z1 – start opening doors, z2 – start closing doors, z3 – call to emergency service. The test set in this problem contains nine tests (see Table 1 for example of tests).

Table 1. Tests for FSM controlling the elevator doors

Input sequence	Output sequence
e11, e2, e12, e2	z1, z2
e11, e2, e12, e2, e11, e2, e12, e2	z1, z2, z1, z2
e11, e2, e12, e3, e2, e12, e2	z1, z2, z1, z2

If we use only tests to design the finite state machine with genetic algorithm we can get as a result the finite state machine (see Figure 3) which passes all tests but works improperly in some cases – this finite state machine can generate an output action "start opening doors" when doors are open or it can generate an output action "start closing doors" when the doors are jammed.

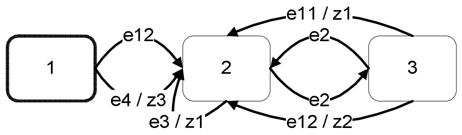

Figure 3. Finite state machine constructed using only tests.

Requirements for the FSM for elevator doors are expressed using five LTL-formulae (two of them are given in Table 2).

Table 2. LTL-formulae for FSM controlling the elevator doors

LTL-formula	Output sequence
G(wasEvent(ep.e11) => wasAction(co.z1))	If button "Open the doors pressed" then start opening doors
G(wasEvent(ep.e4) <=> wasAction(co.z3))	Call to emergency service is done if and only if doors are jammed

Using these LTL-formulae together with tests as a specification we can construct the finite state machine (see Figure 4) that works properly in all cases.

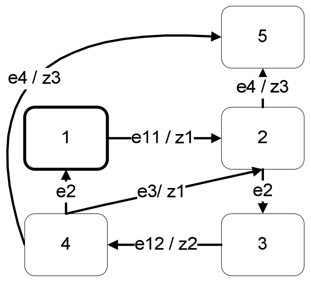

Figure 4. FSM constructed using tests and LTL-formulae.

To measure the performance of the algorithm 1000 runs were made for induction from tests only and from test and LTL-formulae. Following values of algorithm parameters were used in these experiments: population size – 2000; the number N_s of states of FSM – 6; the desired number N_t of FSM transitions – 7; elite size – 10%; mutation probability – 5%. For each run the number of fitness function calculations was recorded (see Table 3 for statistics). In test-based induction the correct FSM was constructed only in 9 cases of 1000, in verification and test-based in all 1000 cases.

Table 3. Number of fitness function calculations

	Test-based induction	Verification and test-based induction
Average	7.479×10^4	7.246×10^5
Minimal	2.184×10^4	7.054×10^4
Maximal	2.999×10^5	5.492×10^6
Standard deviation	2.54×10^4	7.729×10^5

11. CONCLUSION

In this paper, we described the method of FSM induction using genetic algorithm with fitness function, cross-over and mutation based on testing and model checking. The input data for the algorithm described is a tests set and LTL-formulae set. The main idea of this method is the reduction of search space with transition labeling algorithm, the test based cross-over and mutation which uses results of LTL-formula verification.

This method was applied the problem of elevator doors control FSM induction. In it usage of verification helped to induct the correct FSM that could be hardly constructed using tests only.

Future work includes development of cross-over methods that uses results of verification.

The research was supported by Ministry of Education and Science of Russian Federation in the context of Federal Program "Scientific and pedagogical personnel of innovative Russia".

12. REFERENCES

[1] Clarke, E., Grumberg, O., Peled, D. *Model Checking*. The MIT Press, 2001.

[2] Egorov, K., Shalyto, A. *Method for Verification of Automata-Based Programs* (in Russian). Information and Control Systems (Informatsionno-upravljajuschie sistemy). St. Petersburg, Politehnika. 2008, № 5, pp. 15–21.

[3] Gurov, V., Mazin, M., Narvsky, A., Shalyto, A. *Tools for Support of Automata-Based Programming*. Programming and Computer Software, 2007, Vol. 33, No. 6, pp. 343–355.

[4] Hamming, R. Error detecting and error correcting codes. *Bell System Technical Journal* 29 (2), pp. 147–160.

[5] Hoffman, L. Talking Model-Checking Technology. *Communications of the ACM*, 2008, Vol. 51. № 7, pp. 110–112.

[6] Johnson, C. Genetic Programming with Fitness based on Model Checking. *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007. Volume 4445/2007, pp. 114–124.

[7] Levenshtein, V. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10: 707–10. 1966.

[8] Lucas, S., Reynolds, J. *Learning DFA:* Evolution versus Evidence Driven State Merging. *The 2003 Congress on Evolutionary Computation* (CEC '03). Vol. 1, pp. 351–358.

[9] Lucas, S., Reynolds, J. Learning Deterministic Finite Automata with a Smart State Labeling Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 27, №7, 2005, pp. 1063–1074.

[10] Lucas, S. Evolving Finite-State Transducers: Some Initial Explorations. *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. Volume 2610/2003, pp. 241–257.

[11] Polikarpova, N., Shalyto, A. *Automata-based programming*. Piter, 2009 (in Russian).

[12] Shalyto, A. *Logic Control and Reactive Systems: Algorithmization and Programming*. Automation and Remote Control, Vol. 62, No. 1, 2001, pp. 1–29.

[13] Verification Technology of Automata-Based Control Programs with Complex Behavior. Report (in Russian). SPbSU ITMO. 2007. http://is.ifmo.ru/verification/_2007_02_report-verification.pd