Test-Based Extended Finite-State Machines Induction with Evolutionary Algorithms and Ant Colony Optimization

Daniil Chivilikhin
St. Petersburg National Research University of IT, Mechanics and Optics
Russia, St. Petersburg, Kronverksky pr., 49
+7 (812) 232-43-18
chivilikhin.daniil@gmail.com

Vladimir Ulyantsev
St. Petersburg National Research University of IT, Mechanics and Optics
Russia, St. Petersburg, Kronverksky pr., 49
+7 (812) 232-43-18
ulyantsev@rain.ifmo.ru

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

ABSTRACT
In this paper we consider the problem of extended finite-state machines induction. The input data for this problem is a set of tests. Each test consists of two sequences – an input sequence and a corresponding output sequence.

We present a new method of Extended Finite-State Machines (EFSM) induction based on an Ant Colony Optimization algorithm (ACO) and a new meaningful test-based crossover operator for EFSMs. New algorithms are compared with a genetic algorithm (GA) using a traditional crossover, a (1+1) evolutionary strategy and a random mutation hill climber. This comparison shows that the use of test-based crossover greatly improves performance of GA. GA on average also significantly outperforms the hill climber and evolutionary strategy. ACO outperforms GA, and the difference between average performance of ACO and GA hybridized with hill climber is insignificant.

Categories and Subject Descriptors
I.2.2 [**Artificial Intelligence**]: Automatic Programming – *Program synthesis.*

General Terms
Algorithms, Experimentation.

Keywords
Extended Finite-State Machine Induction, Ant Colony Optimization, Genetic Algorithm, Crossover.

1. INTRODUCTION
In search-based software engineering (SBSE) [1]–[3] search-based optimization algorithms are used to solve software engineering problems. As survey [2] shows, the methods most widely used in this area are evolutionary algorithms.

Automata-based programming [4] is the programming paradigm in the context of which it is proposed to design and implement a software system as a system of interacting automated controlled objects. Each automated controlled object consists of a finite-state machine and a controlled object. The finite-state machine has a set of states, a transition function and an action function. The controlled object has commands and requests (implemented by its methods) and a set of computational states.

A finite state machine 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 finite-state machine makes a transition on which some output action is sent to the controlled object. The output action can also be performed on entering the state. The output actions can change the computational state of the controlled object. The main idea of automata-based programming is to distinguish between control states and computational states. The number of control states is not large so they can be drawn on a transition graph, each one of them differs qualitatively from the 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 the 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 finite-state machine.

As survey [2] shows, only evolutionary algorithms, ant colony optimization [14], simulated annealing and hill climbing methods are used for software design in SBSE. The same survey states there is not enough works comparing different methods. FSM and EFSM induction is a subcase of software design.

Similar problems have been studied by several researchers. In [4] two approaches to induction of FSMs from examples were compared. This comparison showed that (1+1) evolutionary strategy outperforms Evidence-Driven State Merging when the number of states is relatively small.

In [7] Lucas applied (1+1) evolutionary strategy to induction of finite state transducers from a set of tests. Lucas and Reynolds used three types of fitness functions: based on strict comparison of strings, based on Hamming distance [8] and based on edit distance [9]. Experiments showed that the edit distance function shows the best performance.

In [10] Johnson used verification (model checking) for fitness evaluation. He used a (1+λ) evolutionary strategy as an optimization method and computational tree logic to represent temporal properties.

Use of evolutionary strategies and random mutation hill climber in these papers is inspired by the fact that it is hard to develop a meaningful crossover operation for EFSMs since there is no way to automatically identify functionally coherent modules in automata [11], [10].

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

Previous works such as [12] show that simulated annealing does not improve performance compared to genetic algorithms.

In this paper we propose a new meaningful crossover operator for EFSM induction problem and a new approach to EFSM induction based on ant colony optimization. We compare these approaches with other evolutionary algorithms.

The paper is structured as follows. Section 2 contains the problem definition. In Section 3 we describe the EFSM representation in algorithms. In Section 4 we describe the fitness function. Section 5 describes the new crossover operator for genetic algorithms and in Section 6 we describe the new ant colony optimization (ACO) algorithm for EFSM induction. Section 7 presents the results of experimental evaluation. The paper ends with conclusion.

2. PROBLEM DEFINITION

In this section we give the definition of the EFSM and describe the input data for the problem of test-based EFSM induction.

2.1 Definition of an EFSM

An EFSM is a seven-tuple $<E, X, Z, \Sigma, s_0, \varphi, \delta>$, where E is a set of events, X is a set of Boolean input variables, Z is a set of output actions, Σ is a set of states, $s_0 \in \Sigma$ is the initial state, $\varphi: \Sigma \times E \times 2^X \to S$ is the transition function, $\delta: \Sigma \times E \times 2^X \to Z^*$ is the action function.

In practice EFSMs are represented as transitions graphs on which each transition is marked with an event e, a Boolean formula f over input variables (it defines the guard condition) and a sequence of output actions o. Upon receiving an event e the transition t is executed if the Boolean formula f is evaluated to "true" with the current values of input variables. When t is executed a sequence of output actions o is generated and the EFSM comes to the state to which t leads. Fig. 1 shows an example of transition graph.

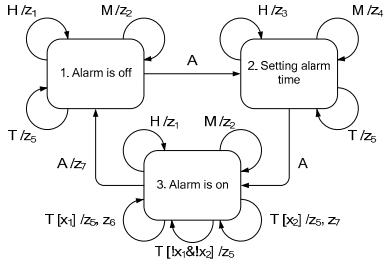

Figure 1. Transition graph example.

This transition graph represents the EFSM for the alarm clock control system. This alarm clock has three buttons for setting the current time, setting the alarm to a specific time and turning it on or off. These buttons are marked "A", "M" and "H". The alarm clock has two operation modes – one mode for setting the current time and another one for setting the alarm time.

When the alarm is off, the user can push the buttons "H" and "M" to increment the hours and minutes of the current time respectively. If the user presses the "A" button, the clock is switched to the alarm setting mode. In this mode the same buttons "H" and "M" are used to adjust the time when the alarm should sound. When the user presses the "A" button in this mode, the alarm is set to go off when the current time will be equal to the alarm time. When the alarm is ringing, the user can press the "A" button to switch it off, however it will automatically turn off after one minute. The alarm clock also contains a timer that increments the current time each minute. This system has four input events, seven output actions and two Boolean input variables.

2.2 Input Data

The input data for inducting an EFSM consists of:

- a set of events $E = \{e_1, e_2, \ldots, e_n\}$;
- a set of input variables $X = \{x_1, x_2, \ldots, x_m\}$;
- a set of output actions $Z = \{z_1, z_2, \ldots, z_k\}$;
- a set of tests denoted by T;
- the maximum number of states in the sought finite-state machine C.

The tests have the following structure: the j-th test consists of two sequences – the input sequence I_j and the reference output sequence A_j. The elements of the input sequence are pairs (e, f), where e is some event from E and f is a Boolean formula over input variables defining the guard condition.

2.3 Output Data

The output data in the problem of test-based EFSM induction is a finite-state machine with no more than C states which complies with each test from T or a notification that such a finite-state machine does not exist. A finite-state machine M complies with the test $T_j = (I_j, A_j)$ if the result of passing the sequence I_j to M equals A_j.

3. INDIVIDUAL REPRESENTATION

The individual representation contains three parts: the number of states; the number of the initial state; an array containing thes descriptions of states. The description of each state is an array of descriptions of outgoing transitions. Each transition has three fields: the event associated with this transition; the number of the state which this transition leads to; the number of output actions on it.

Note that we do not evolve the output sequences on the EFSM transitions and do not store actions in the individual representation. Instead, we use a technique introduced in [13] and used in [14] called *smart transition labeling*, which is similar to the smart state labeling used in [4]. The idea is to evolve only the numbers of output actions for each transition. The output sequences are selected optimally using a simple algorithm based on tests. For example, the EFSM given on Fig. 2 has the following representation: {2, 0, {{A, 1, 0}, {T, 1, 1}}, {{T, 1, 1}, {M, 0, 2}}}.

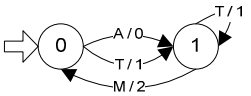

Figure 2. Example of an individual.

4. FITNESS FUNCTION

To evaluate the fitness function, first, we pass each of the input sequences I_j to the EFSM and record the corresponding output sequences O_j. After that, the following value is calculated:

$$FF_1 = \frac{1}{|T|}\sum_{j=1}^{|T|}\left(1 - \frac{ED(O_j, A_j)}{\max\left(len(O_j), len(A_j)\right)}\right)$$

Here, $|T|$ denotes the cardinality of set T, which is the number of test examples, $len(s)$ denotes the length of sequence s and $ED(s_1, s_2)$ denotes the edit distance between sequences s_1 and s_2.

The final expression for the fitness function has the form:

$$FF_2 = \begin{cases} 10 \cdot FF_1 + \frac{1}{M} \cdot (M - cnt), FF_1 < 1 \\ 20 + \frac{1}{M} \cdot (M - cnt), FF_1 = 1 \end{cases}$$

Here M is some number greater than the maximum possible number of transitions in EFSM (in experiments M is equal to 100), and cnt is the number of transitions in EFSM. This fitness function favors EFSMs fully compliant with tests and having less transitions over EFSMs not compliant with tests and having more transitions.

5. CROSSOVER OPERATOR

The crossover operator takes two EFSMs P1 and P2 as input and outputs two EFSMs S1 and S2. The initial states of S1 and S2 have number "0" since it holds for all EFSMs generated by the genetic algorithm. Let us denote the transitions lists for state i of P1 and P2 as P1.$T[i]$ and P2.$T[i]$ respectively. The transitions lists S1.$T[i]$ and S2.$T[i]$ are constructed in the following way:

1. The transitions used in processing 10% of tests for which the normalized edit distance between sequences A_j and O_j $\frac{\mathrm{ED}(O_j, A_j)}{\max(len(O_j), len(A_j))}$ is minimal are marked in EFSMs P1 and P2.
2. The transitions are inserted into list S1.$T[i]$ in the following order: firstly, marked transitions from P1.$T[i]$, secondly, marked transitions from P2.$T[i]$, thirdly, non-marked transitions from P1.$T[i]$.
3. The transitions are inserted into list S2.$T[i]$ in the following order: firstly, marked transitions from P2.$T[i]$, secondly, marked transitions from P1.$T[i]$, thirdly, non-marked transitions from P2.$T[i]$.
4. Duplicated and contradictory transitions removal is applied to both lists. Let us denote the input list for this operation as L_{in}, and the output list – as L_{out}. Following actions are performed to construct L_{out}:
   - transitions from L_{in} are considered one by one;
   - the transition t is inserted into L_{out} if it does not contain a transition marked with the same event as t and with a guard condition having a common satisfying assignment with guard condition of t.

6. ACO FOR EFSM INDUCTION

In ACO, solutions are built by a set of artificial ants which use a stochastic strategy. The solutions can be represented either as paths in the graph called the *construction graph*, or simply by graph vertices. Each edge (u, v) of the construction graph (u and v are vertices of the graph) has an assigned pheromone value τ_{uv} and can also have an associated heuristic distance η_{uv}. The pheromone values are modified by the ants in the process of solution construction, while the heuristic distances are assigned initially and are not changed.

In our ACO algorithm for FSM induction the search space, which is the set of all FSMs with given parameters (number of states, the set of events and the set of actions), is represented in the form of a directed graph G. The nodes of G are associated with FSMs while the edges of G are associated with FSM mutations.

Informally speaking, a mutation of a FSM is a small change in its structure. For a particular transition, a mutation can change its output action or destination state. Mutations can also change the number of FSM accessible states. In this work we consider two FSM mutation types:

- **Change transition end state**. For a random transition in the EFSM the transition's end state is set to another state selected uniformly randomly from the set of all states.
- **Change the number of actions on transition**. For a random transition in the EFSM the number of actions performed on it is set to a random number from zero to some upper bound.

For each pair of FSMs and the corresponding pair of nodes there exists a path in G from one node to the other and vice versa, i.e. by applying certain mutations to some FSM we can get any other FSM from the search space.

We use N artificial ants in our algorithm. On each iteration, start nodes are selected for each ant. In the next step called `ConstructSolutions`, the ants explore the graph, selecting the next node to visit according to the pheromone values on the edges. Each ant is given at most n_{stag} iterations to run without an increase in its best fitness value. Let the artificial ant be located in a node u holding a FSM A and let the fitness value of A be $f(A) = f_A$. If node u has successors, then the ant selects the next node v to visit according to the following rules:

a) With a probability of p_{new} the ant attempts to construct new edges of the graph by making a fixed number of mutations of A. In this case, the ant selects the best node from the newly constructed successors of u.

b) With a probability of $(1 - p_{new})$ the ant stochastically selects the next node from the existing successors of node u according to the classical ACO formula:

$$p_{uv} = \frac{\tau_{uv}^{\alpha} \cdot \eta_{uv}^{\beta}}{\sum_{w \in N_u} \tau_{uw}^{\alpha} \cdot \eta_{uw}^{\beta}}$$

Here, N_u is the successors set of node u, $v \in N_u$ and $\alpha, \beta \in [0,1]$. We have not been able to come up with a reasonable meaning for the heuristic distances η_{uv}, so all the heuristic distances are considered to be equal and do not influence path selection.

If u does not have any successors, then the next node is selected according to rule a) with a probability of 1.0.

In the next step, called `UpdatePheromones`, the pheromone values of all graph edges are changed. Here we use a variation of the Elitist Ant model, where the best so far solution deposits pheromone along with the other solutions on each step. For each graph edge (u, v) we store $\Delta\tau_{uv}^{best}$ – the best pheromone value that any ant has ever deposited on edge (*u, v*). For each ant path, a sub-path is selected that spans from the start of the path to the best solution node in the path. The values of $\Delta\tau_{uv}^{best}$ are updated for all edges along this sub-path. Next, for each graph edge (*u, v*), the pheromone value is updated according to the classical formula:

$$\tau_{uv} = \rho\tau_{uv} + \Delta\tau_{uv}^{best},$$

where $\rho \in [0,1]$ is the evaporation rate. The whole population of N artificial ants is given at most N_{stag} iterations to run without an increase in the best fitness function value. After this number is exceeded, the algorithm is restarted.

When we apply ACO to FSM generation we have to deal with huge graphs, sometimes consisting of several millions vertices. In our work we apply a variation of the *expansion technique* introduced in ACOhg [15] – we limit the ant path lengths to reduce the size of the graph we store in memory.

7. EXPERIMENTAL EVALUATION

The following algorithms were compared in experimental evaluation: a genetic algorithm with traditional crossover (GA-1), a random mutation hill climber (RMHC), (1+1) evolutionary strategy (ES), a genetic algorithm with test-based crossover (GA-2), GA-2 hybridized with RMHC (GA-2+HC) and ant colony optimization (ACO). Experiments were performed using tests for the alarm clock problem [13]. The input data contained 38 tests with a total length of input sequences 242 and a total length of answer sequences 195. We searched for the solution among EFSMs with 4 states, in each experiment the goal was to construct an EFSM with 14 transitions which corresponds to the value of fitness function equal to 20.86.

Genetic algorithms had the following values of parameters: population size – 2000; selection method – elitism; elite size – 10%; mutation probability – 5%. Mutation in GA was performed in the same way as in [14]. RMHC also used the same mutation. In ES the number of mutations was selected from an exponential distribution with λ=0.5 and each mutation was the same as in RMCH. Note that mutations used in evolutionary algorithms differ from mutations used in ACO. In GA-2+HC GA was used until 95% of target fitness value was reached and after that the EFSM was optimized with RMHC. ACO had the following values of parameters: number of ants N – 5, ant stagnation parameter n_{stag} – 20, population stagnation parameter N_{stag} – 4, new mutation probability p_{new} – 0.6.

We have performed 1000 runs of each algorithm; the number of fitness function evaluations has been recorded for each run. All constructed EFSMs use only three out of four states and are isomorphic to the EFSM from Fig. 1. Table 1 shows the statistical parameters of these runs.

Table 1. Experiment results.

Algorithm	Min	Max	Avg	Median
GA-1	855390	38882588	5805943	4588736
RMHC	1150	9592213	1423983	957746
ES	1506	9161811	3447390	856730
GA-2	32830	599022	117977	83787
GA-2+HC	26740	188509	53706	48106
ACO	2440	210971	53944	46293

8. CONCLUSION

We have presented a new method of Finite-State Machines (FSM) induction based on an Ant Colony Optimization algorithm (ACO) and new crossover operator for genetic algorithms. New algorithms are compared with a (1+1) evolutionary strategy, a random mutation hill climber and different variants of genetic algorithm. This comparison shows that use of test-based crossover greatly improves the performance of a genetic algorithm. The genetic algorithm on average also significantly outperforms hill climber and evolutionary strategy. ACO outperforms GA-2, and the difference between average performance of ACO and GA-2+HC is insignificant.

9. ACKNOWLEDGMENT

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".

10. REFERENCES

[1] Clark J., et al. *Reformulating Software Engineering as a Search Problem.* IEEE Proceedings—Software, vol. 150, № 3, 2003, pp. 161–175.

[2] Harman M., Mansouri A. and Zhang Y. *Search-Based Software Engineering: A Comprehensive Analysis and Review of Trends, Techniques, and Applications*, tech. report TR- 09-03, Dept. of Computer Science, King's College London, 2009.

[3] Harman M. *Software Engineering Meets Evolutionary Computation* // Computer. 2011. Vol. 44, № 11, pp. 31 – 39.

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

[5] Dorigo, M. 1992. *Optimization, Learning and Natural Algorithms.* PhD thesis. Polytechnico di Milano, Italy.

[6] Lucas, S. and Reynolds, J. Learning DFA: Evolution versus Evidence Driven State Merging. The 2003 Congress on Evolutionary Computation (CEC'03). Vol. 1, 351-348.

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

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

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

[10] 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.

[11] Lucas, S. and 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.

[12] Zaikin, A. *Development of finite-state machines induction methods with simulated annealing for the "Competition for resources" game.* Nauchno-technicheskij vestnik SPbGU ITMO (in Russian). 2011. № 2, pp. 49 – 54.

[13] Tsarev, F. 2010. *Method of finite-state machine induction from tests with genetic programming.* Information and Control Systems (Informatsionno-upravljayushiye sistemy, in Russian), no. 5, pp. 31-36. http://is.ifmo.ru/works/_zarev.pdf

[14] Tsarev, F. and Egorov, K. *Finite-state machine induction using genetic algorithm based on testing and model checking*. 2011. In *Proceedings of the 2011 GECCO Conference Companion on Genetic and Evolutionary Computation* (GECCO'11). NY. ACM. 2011. pp. 759-762.

[15] Alba, E. and Chicano, F. 2007. *ACOhg: dealing with huge graphs.* In Proceedings of the 9th annual conference on Genetic and evolutionary computing (GECCO'07). ACM, NY, USA, pp. 10-17.