7th IFAC Conference on Manufacturing Modelling, Management, and Control
International Federation of Automatic Control
June 19-21, 2013. Saint Petersburg, Russia

Test-Based Induction of Finite-State Machines with Continuous Output Actions

Igor Buzhinsky * **Vladimir Ulyantsev** ** **Anatoly Shalyto** ***

* *Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, St. Petersburg, Russia (e-mail: buzhinsky@rain.ifmo.ru)*
** *Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, St. Petersburg, Russia (e-mail: ulyantsev@rain.ifmo.ru)*
*** *Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, St. Petersburg, Russia (e-mail: shalyto@mail.ifmo.ru)*

Abstract: In this paper we improve an earlier developed method of finite-state machine (FSM) induction for controlling objects with complex behavior. This method allows to construct FSMs with continuous (real-valued) output actions. A set of human-created training samples serves as input data for it. We apply an ant colony optimization algorithm and a (μ, λ)-evolution strategy for solving the problem as more effective than a genetic algorithm used in the initial method. The modification of the method is evaluated on the problem of unmanned aircraft control.

Keywords: automatic control, finite-state machines, ant colony optimization, evolution strategies, genetic algorithms, aircraft control

1. INTRODUCTION

Automata-based programming (see Polikarpova and Shalyto (2009), Shalyto (2001), Gurov et al. (2007)), a relatively new programming style, suggests to code computer programs in a way similar to the automatization of technological processes. A key component of automata-based programs is a finite state machine (FSM), or an automaton. One of the problems suitable for automata-based programming is the problem of controlling objects which are able to behave differently in response to the same input data, or objects with *complex behavior*. An example of such problem is the Artificial Ant problem (see Koza (1992)).

For a variety of problems, including the Artificial Ant problem, manual construction of FSMs is difficult. However, it is possible to automate the process of FSM construction. For that, a performance criteria for FSMs is defined. One way to do this is to introduce a *fitness function* which maps FSMs to real numbers, such that for better FSMs the value of the fitness function is greater. There are plenty of search optimization algorithms such as genetic algorithms or evolution strategies which can be used for finding FSMs which maximize the fitness function. Another approach, which does not use fitness functions, is to define constraints an FSM must satisfy. This approach was used in Heule and Verwer (2010) and later in Ulyantsev and Tsarev (2012).

In Polikarpova et al. (2010), a genetic algorithm was applied to construct a Mealy FSM capable of controlling a model of unmanned aircraft. In this approach fitness function computation was based on modelling FSM's behavior in a flight simulator, and a month was required to build target FSMs with proper behavior. Later, in Alexandrov et al. (2011) a fitness function based on training samples, or tests, was used. This reduced the time required to build a target FSM to several hours. The approach suggested in Alexandrov et al. (2011) was also based on genetic algorithms. However, it allowed to construct Mealy FSMs with real-valued, or continuous, output actions as well as with discrete ones. The input values, which are continuous as well, are transformed to the predicate values used as input events.

In the present work the approach suggested in Alexandrov et al. (2011) is further developed. An ant colony optimization algorithm and an evolution strategy are used to construct FSMs controlling a model of an unmanned aircraft. This approach shows better performance than approaches described above. *FlightGear* (http://www.flightgear.org) open-source flight simulator (see Fig. 1) is used for test recording and FSM testing.

2. FINITE-STATE MACHINES

A finite-state machine is a sextuple $(S, \Sigma, \Delta, \delta, \lambda, s_0)$, where S is a finite set of states, Σ is a set of input events, Δ is a set of output actions, $\delta : S \times \Sigma \to S$ is a transition function, $\lambda : S \times \Sigma \to \Delta$ is an output function and $s_0 \in S$ is a start state.

We define l predicates $x_1, ..., x_l$ which map continuous input flight data to Boolean values. There are $2l$ events in Σ: two events x_i and $\neg x_i$ for each predicate x_i. Output actions are real-valued, thus $\Delta = \mathbb{R}^C$ where C is a number of flight controls managed by an FSM.

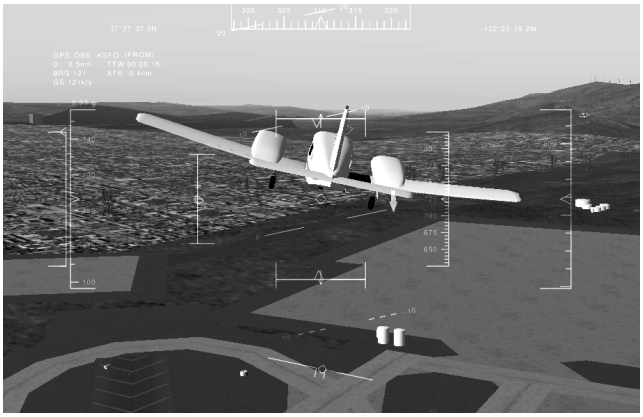

Fig. 1. *FlightGear* flight simulator (screenshot)

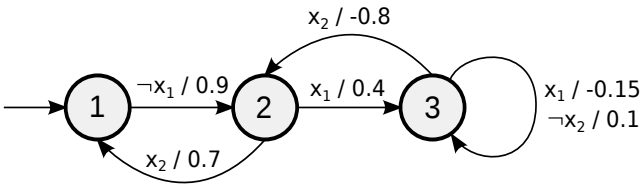

Fig. 2. An example of a Mealy FSM with continuous output actions ($l = 2$, $C = 1$)

When the FSM receives a new tuple of input data, the predicate values are calculated. Then, for each predicate x_i, if there is a transition for x_i or $\neg x_i$ (depending on the predicate value) from the current state of FSM, this transition is executed. An example of an FSM is shown in Fig. 2.

3. PROBLEM DESCRIPTION

We consider a finite set of training samples, or tests, which describe the behavior of the controlled object. Having the tests, we want to construct an FSM capable of controlling the object. We assume that all tests are human-prepared and therefore may not show a perfect behavior of the controlled object. In this paper we describe an FSM induction method improvement: an ant colony optimization algorithm and an evolution strategy are applied to the problem of controlling an unmanned aircraft. To model the behavior of the unmanned aircraft, we use *FlightGear* flight simulator, which allows to record input values corresponding to the flight parameters (airspeed, altitude, etc.) of the aircraft, as well as positions of flight controls (ailerons, rudder, etc.), or output values.

Flight control positions can be characterized with values: integral (for controls such as starter) or real (for controls such as elevator or ailerons). To simplify the description of the problem, from this point on we shall assume that all these values are real. The control of the unmanned aircraft is performed by FSM output actions which change the flight control positions.

The FSM induction method presented in this paper is test-based. Each test is a sequence of flight parameters and corresponding flight control positions. In our work all tests in a test set describe an aerobatic figure to be performed by the FSM controlling the aircraft.

Now let us formalize the term *test*. Each test $T[i]$ ($i = 1..N$, where N is the size of the test set), has its own length $\text{len}[i]$ as the number of time moments recorded. We define an *input tuple* $\text{in}[i][t]$ of the i-th test ($i = 1..N$) at time t ($t = 1..\text{len}[i]$) as a sequence of P real numbers corresponding to the flight parameters of the aircraft recorded at time t:

$$\text{in}[i][t] = (\text{in}[i][t][1], ..., \text{in}[i][t][P]) .$$

Similarly, an *output tuple* $\text{out}[i][t]$ is a sequence of C real numbers defining the flight control positions of the aircraft at a specific moment:

$$\text{out}[i][t] = (\text{out}[i][t][1], ..., \text{out}[i][t][C]) .$$

For each control m we assume that all output tuple values are bounded with a pair of numbers $c_m^{\min}$ and $c_m^{\max}$:

$$c_m^{\min} \le \text{out}[i][t][m] \le c_m^{\max}, i = 1..N, t = 1..\text{len}[i], m = 1..C.$$

For instance, we expect the values of elevator to belong to the segment $[-1, 1]$.

During test creation, input tuples and corresponding output tuples are recorded with a rate of 10 Hz. When an FSM interacts with the flight simulator, it receives an input tuple and generates an output tuple with the same rate.

To sum up, test $T[i]$ is formed of two tuple sequences, $\text{in}[i]$ and $\text{out}[i]$, each of length $\text{len}[i]$. An example of a test is shown in table 1. The data for the test example is taken from a real test.

Table 1. Test example for $P = 4$, $C = 3$

Sequence	Description	$t = 1$	$t = 10$	$t = 20$
$\text{in}[i][t][1]$	Pitch angle (°)	3.078	3.544	4.112
$\text{in}[i][t][2]$	Roll angle (°)	−0.076	0.351	3.413
$\text{in}[i][t][3]$	Heading (°)	198.03	198.11	198.41
$\text{in}[i][t][4]$	Airspeed (knots)	251.42	252.29	253.20
$\text{out}[i][t][1]$	Aileron position	0.000	0.032	0.073
$\text{out}[i][t][2]$	Rudder position	0.000	0.016	0.037
$\text{out}[i][t][3]$	Elevator position	−0.035	−0.039	−0.037

Each FSM transition is marked with events, represented by Boolean formulae of the form x_i or $\neg x_i$, where x_i is one of the predicates. Each predicate value at time t may depend not only on $\text{in}[i][t]$, but also on $\text{in}[i][t']$, where $1 \le t' \le t-1$. For instance, we can compute the value of "acceleration is positive" predicate at time t by calculating the difference between the aircraft's velocities at times t and $t-1$.

Consider some FSM receiving predicate values computed on the input tuples of the i-th test. Each transition j of an FSM is marked with its own output tuple $u_j = (u_j[1], ..., u_j[C])$, which corresponds not to the flight control values, but to the changes of them. The resulting output tuples $\text{ans}[i][t] = (\text{ans}[i][t][1], ..., \text{ans}[i][t][C])$ produced by the FSM are defined in the following way. The resulting output tuple at time $t = 1$ is equal to the first output tuple in the test:

$$\text{ans}[i][1][m] = \text{out}[i][1][m], m = 1..C. \quad (1)$$

If transitions $j_1, ..., j_l$ were executed at time $t > 1$, then

$$\text{ans}[i][t][m] = \text{ans}[i][t-1][m] + \sum_{s=1}^{l} u_{j_s}[m], m = 1..C. \quad (2)$$

Thus, the resulting output tuple produced by the FSM at time t is the sum of out$[i][1]$ and output tuples of transitions executed up to the moment t.

4. FSM INDUCTION METHOD

To generate an FSM which demonstrates behavior close to the one given by tests, we use search optimization algorithms. We start the description of the FSM induction method with the fitness function and the individual representation. After that, a detailed description of the algorithm implementations is given.

4.1 Fitness Function

The fitness function f, which is defined below, is a measure of similarity of the FSM's behavior and the behavior shown in the tests. The less the difference between sequences out$[i]$ and ans$[i]$ is, the more the value of f is. We define f in the following way:

$$f(\text{FSM}) = 1 - \sqrt{\frac{1}{N}\sum_{i=1}^{N} d_i^2},$$

where d_i is the distance between two output tuple sequences:

$$d_i = \sqrt{\frac{1}{\text{len}[i]}\sum_{t=1}^{\text{len}[i]}\frac{1}{C}\sum_{m=1}^{C}\left(\frac{\text{out}[i][t][m] - \text{ans}[i][t][m]}{c_m^{\max} - c_m^{\min}}\right)^2}.$$

4.2 Individual Representation

Consider all possible FSMs with M states and T transitions and assume transition tables are used. For each transition the output tuple, which is formed by C real numbers, should be defined. If we used FSMs as individuals for search optimization algorithms, the search space would become continuous and rather huge.

However, in Alexandrov et al. (2011) a *transition labeling* algorithm was suggested that, given the transition function, finds the output function which maximizes the value of the fitness function f. Using equations 1 and 2 to expand the values ans$[i][t][m]$ for different i, t and m, and taking the partial derivatives of f with respect to u_j $(j = 1..T)$, it is possible to get a linear equation system

$$\frac{\partial f}{\partial u_j} = 0 \ \ (j = 1..T),$$

which is solved by the algorithm. The running time of the transition labeling algorithm is $O\left(T^3 + T^2\sum_{i=1}^{N}\text{len}[i]\right)$.

Further we will refer to an FSM with only the transition function defined as to the FSM *skeleton*. Thus, the way to reduce the search space size is the following: FSM skeletons are used as individuals and the transition labeling procedure is executed for each generated skeleton.

Later, if A is an FSM skeleton, $f(A)$ will denote the fitness function of the FSM formed from A by defining the output tuples.

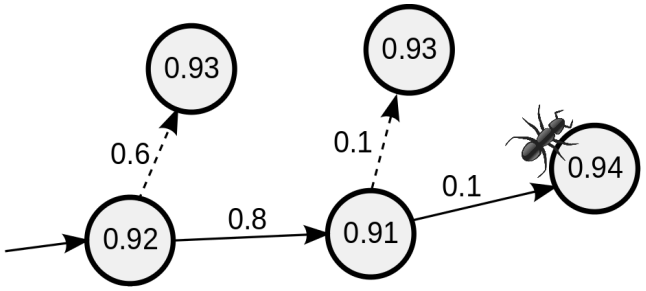

Fig. 3. A part of the construction graph. Arcs are marked with pheromone values and fitness function values are shown inside the circles. A possible ant path is shown with solid arrows

4.3 Optimization Algorithms

In this paper we apply an ant colony optimization algorithm and a (μ, λ)-evolution strategy to construct FSMs with appropriate fitness values. The performance of these algorithms is compared to the performance of the genetic algorithm earlier used in Alexandrov et al. (2011).

We will further call some FSM skeleton A *better* than some other FSM skeleton B if $f(A) > f(B)$. Recall that output tuples are assigned to FSM skeletons automatically once they are generated.

Ant colony optimization algorithm. Ant colony optimization (ACO) was originally proposed in Dorigo (1992). In the present work, we apply an ACO-based algorithm suggested in Chivilikhin and Ulyantsev (2012).

In this algorithm, the search space, which is the set of all FSM skeletons with given number of states, is represented in the form of a directed graph G called the *construction graph*. The nodes of G are associated with FSM skeletons while the arcs of G are associated with small changes in skeletons – mutations. For our problem, we have chosen the mutation to be a change of one of the skeleton's transitions. Thus, there is an arc between each pair of FSM skeletons which differ in a single transition. The graph is initialized with a single vertex and is enlarged during the algorithm. An example of a part of the construction graph is shown in Fig. 3.

At each iteration of the algorithm N_{ants} ants are placed at some vertices of the graph, and then they move searching for individuals with high fitness values. Each ant remembers the best (in terms of fitness value) vertex it visited. The path produced by the ant which found the best vertex of all vertices visited by ants is used to place new ants at the next iteration of the algorithm (ants are randomly placed along this path). Each ant's path is formed according to the following rules (assume the ant is in the vertex v).

(1) With a probability of p_{new} the FSM corresponding to v is mutated to get N_{mut} new neighbors of v which are added to G with the arcs from v to them. The ant selects the best vertex among the new neighbors and moves to it.
(2) Otherwise, the ant moves to one of the existing neighbors of v. The probability of moving to some

neighbor u is proportional to the *pheromone* value of the arc vu.

Each arc uv has its own pheromone value τ_{uv} which is updated at the end of each iteration of the algorithm after all ants made their paths (an ant stops making its path after it visits N_{stag} vertices without the fitness value increase) according to the formula:

$$\tau'_{uv} = \rho\tau_{uv} + \tau_{uv}^{\text{best}},$$

where ρ is the evaporation rate and τ_{uv}^{best} is the best pheromone value deposited on the arc. If uv belongs to the prefix of an ant's path ending with the best vertex w on it, τ_{uv}^{best} is updated by the value increasing with the w's fitness.

In our implementation we used the following parameter values: $N_{\text{ants}} = 4$, $N_{\text{stag}} = 40$, $N_{\text{mut}} = 35$, $p_{\text{new}} = 0.25$, $\rho = 0.35$.

Evolution strategy. Evolution strategies (ES) (see Back et al. (1991)) are algorithms operating with a set of individuals called a *generation*. There are two types of ES: so-called (μ, λ) and $(\mu + \lambda)$-ES, where $\mu, \lambda \in \mathbb{N}$. For each type of ES, the generation size is μ and λ new individuals are generated at each iteration. The only operation used to acquire new individuals is the mutation of individuals from the current generation. At the end of each iteration of a (μ, λ)-ES, a new generation is formed from μ best mutated individuals. In contrast, in a $(\mu + \lambda)$-ES μ new individuals are chosen from both the current generation and mutated individuals.

We have found that a $(50, 500)$-ES shows the best performance on the problem stated in comparison with other evolution strategies inspected.

Genetic algorithm. Genetic algorithms (GA) (see Koza (1992)) are similar to evolution strategies in the way that both algorithms operate with generations. In GA, new individuals are generated not only using mutations, but also using a *crossover* operator which forms a new individual from two existing ones. The *selection* operator forms a new generation from the current one.

In this work, the implementation of GA is similar to the implementation used in Alexandrov et al. (2011). We use 300 as the generation size, 30 as the number of elite individuals, the uniform crossover and the tournament selection operators.

5. EXPERIMENTAL EVALUATION

In this section the experimental evaluation of the described method is presented and the performance of used search optimization algorithms is analysed.

5.1 Aerobatic Figures

The presented method was tested on two aerobatic figures: the Nesterov loop and the barrel roll (360° clockwise rotation around the roll axis). Computer simulation was used to record tests and to model the behavior of generated FSMs.

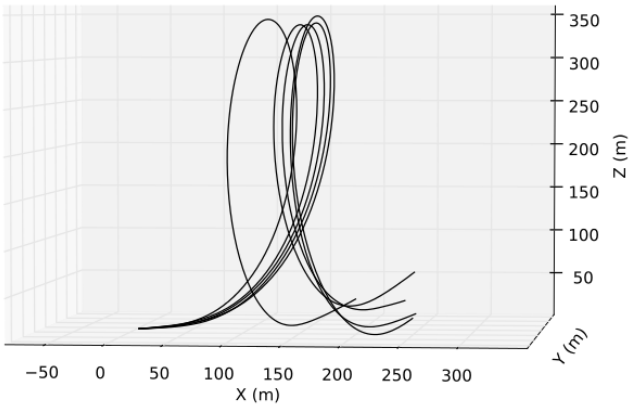

Fig. 4. Trajectories of the aircraft in several tests of the Nesterov loop test set

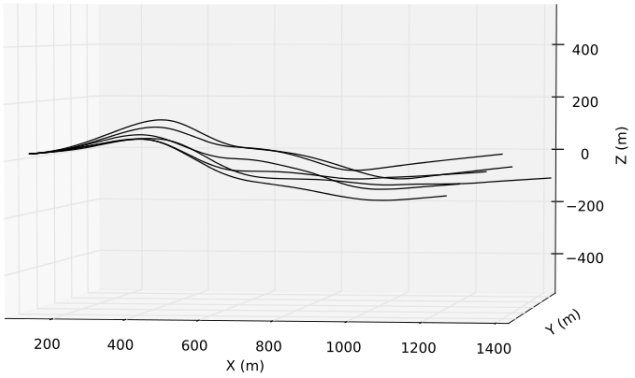

Fig. 5. Trajectories of the aircraft in several tests of the rotation test set

For the Nesterov loop, a model of the *Piper PA34-200T Seneca II* was used. For the barrel roll, we used a model of the *Gloster Meteor*, a jet fighter, as this aerobatic figure was impossible to perform by a civil aircraft. Two sets of 23 and 28 tests were recorded for the Nesterov loop and the barrel roll respectively. We note that the tests for the barrel roll did not show perfect figure execution: the aircraft did not hold the initial altitude and heading well. Trajectories of the aircraft in several tests from the training sets are shown in Fig. 4 and 5.

We used eight predicates for the Nesterov loop and six predicates for the barrel roll to map the continuous input values recorded in the tests to Boolean ones. In Fig. 6 the aircraft's pitch at different times is shown for several tests of the barrel roll test set. In it you can see a graphical interpretation of two predicates:

(1) $x_1(t) = (|\text{pitch}(t) - \text{pitch}(1)| < 2)$;
(2) $x_2(t) = (\text{pitch}(t) > \text{pitch}(1))$,

where pitch(t) is the aircraft's pitch angle at time t. In the barrel roll three flight controls were used: elevator, ailerons and rudder. Fig. 7 shows the time dependence of the aircraft's elevator position for the same tests.

5.2 Results

As mentioned before, a genetic algorithm was earlier applied to solve the problem. However, the time required to build an FSM showing good performance was about

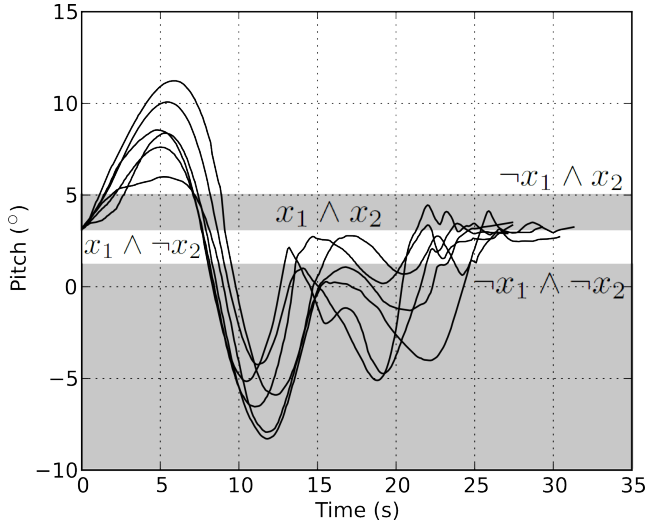

Fig. 6. Pitch time dependence for several tests. Four areas of the pitch axis are annotated with predicate values

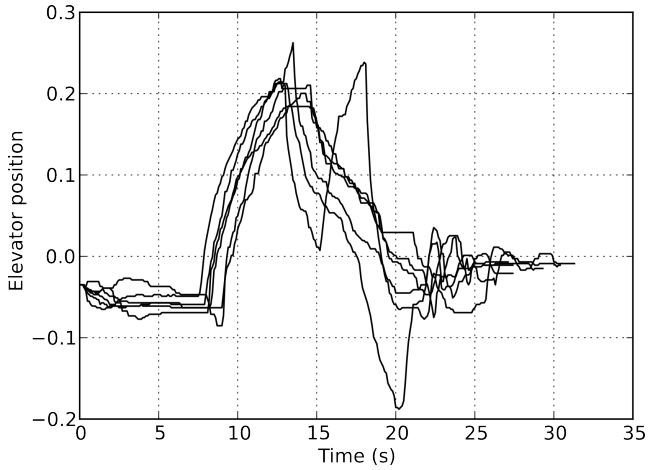

Fig. 7. Time dependence of the elevator position for several tests

several hours. To reduce the number of fitness function evaluations performed by a search algorithm, the ACO algorithm and the ES were implemented.

For both aerobatic figures and corresponding test sets, 25 runs of all three search optimization algorithms (ACO, (μ, λ)-ES and GA) were executed. Every run was stopped after 10^5 fitness function evaluations, which took about 17 or 21 minutes for different test sets. The algorithms were searching for FSMs with $M = 4$ states, which was enough to get an appropriate FSM behavior. Tables 2 and 3 show the run statistics of the search optimization algorithms. There are several fitness values in the left columns of the tables. The numbers of runs in which these fitness values were reached are presented in the three right columns of the tables. On average, the quality of FSMs with lowest and highest fitness values from the left columns of the tables was different.

From the presented statistics we conclude that both ACO and ES outperform the GA. Moreover, the performance of the ACO-based algorithm is better than the performance of the ES.

Table 2. Number of runs in which the fitness values were reached (out of 25) on the Nesterov loop test set

Fitness value	ACO	(μ, λ)-ES	GA
0.9890	11	8	0
0.9887	21	18	2
0.9884	24	24	8
0.9881	24	24	17
0.9878	24	24	21

Table 3. Number of runs in which the fitness values were reached (out of 25) on the barrel roll test set

Fitness value	ACO	(μ, λ)-ES	GA
0.9884	8	3	0
0.9882	23	18	5
0.9880	25	24	15
0.9878	25	24	19
0.9876	25	24	24

An *Intel Core* 2 *Quad Q*9400 processor was used in the experiments. With all four cores of the processor involved in the computation (multiple fitness function evaluations were performed in parallel), the average run time of ACO and ES was about 21 minutes on the barrel roll test set and about 17 minutes on the Nesterov loop test set. The difference of the run times is mainly due to the different sum lengths of the test sets. In most cases, one or two runs of one of these algorithms were enough to build an FSM with an appropriate fitness value, i.e. the fitness value was enough to control the aircraft.

5.3 Results Analysis

FSMs induced by the search optimization algorithms were tested in simulation. For both aerobatic figures, the Nesterov loop and the barrel roll, more than 90% FSMs with the highest fitness function values from different runs of ACO and ES were capable of controlling the aircrafts (it was concluded that the FSM was capable of controlling the aircraft if five out of five figure executions performed in simulation were successful).

FSMs made minor errors in the end of figure executions, i.e. the aircraft's roll or pitch angles were not close enough to zero. However, we conclude that it is possible to fix such faults with one of the following approaches:

(1) the endings of the tests can be recorded more accurately;
(2) it is possible to find predicates better than the ones that were used;
(3) the figure endings can be detected automatically and the control can be transmitted to another preconstructed FSM that stabilizes the aircraft's flight.

A screenshot of the Gloster Meteor performing the barrel roll, as well as the FSM controlling it, is shown in Fig. 8.

6. CONCLUSION

The problem of test-based construction of finite-state machines with continuous output actions is considered in the paper. The method introduced in Alexandrov et al. (2011) is improved by the use of an ant colony optimization algorithm and a (μ, λ)-evolution strategy. These algorithms

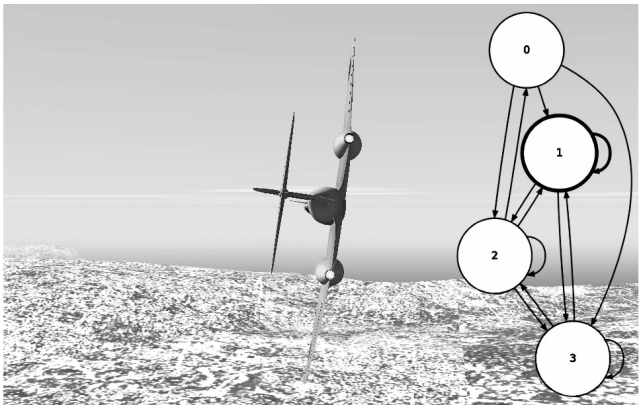

Fig. 8. Screenshot of the Gloster Meteor performing the barrel roll under control of an FSM shown on the right. The current state of the FSM is marked with a bold circle

were found to show better performance on the problem of unmanned aircraft control than the genetic algorithm used in Alexandrov et al. (2011).

REFERENCES

Alexandrov, A., Sergushichev, A., Kazakov, S., and Tsarev, F. (2011). Genetic algorithm for induction of finite automata with continuous and discrete output actions. In N. Krasnogor (ed.), *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation (GECCO '11)*, 775–778. ACM, New York, USA.

Back, T., Hoffmeister, F., and Schwefel, H.P. (1991). A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 2–9. Morgan Kaufmann, La Jolla, CA.

Chivilikhin, D. and Ulyantsev, V. (2012). Learning finite-state machines with ant colony optimization. In *Proceedings of the Eighth International Conference on Swarm Intelligence (ANTS 2012). Brussels, Belgium. 12.09.2012 — 14.09.2012. Lecture Notes in Computer Science*, volume 7461, 268–275. Springer.

Dorigo, M. (1992). *Optimization, learning and natural algorithms*. Ph.D. thesis, Politecnico di Milano, Italy.

Gurov, V., Mazin, M., Narvsky, A., and Shalyto, A. (2007). Tools for support of automata-based programming. *Programming and Computer Software*, 3, no. 6, 343–355.

Heule, M. and Verwer, S. (2010). Exact dfa identication using sat solvers. In J. Sempere and P. Garca (eds.), *Grammatical Inference: Theoretical Results and Applications 10th International Colloquium, ICGI 2010. Lecture Notes in Computer Science*, volume 6339, 66–79. Springer.

Koza, J. (1992). *Genetic programming: on the programming of computers by natural selection*. MIT Press, Cambridge, MA, USA.

Polikarpova, N. and Shalyto, A. (2009). *Automata-based programming (in Russian)*. Piter, St. Petersburg.

Polikarpova, N., Tochilin, V., and Shalyto, A. (2010). Method of reduced tables for generation of automata with a large number of input variables based on genetic programming. *Journal of Computer and Systems Sciences International*, 49, no. 2, 265–282.

Shalyto, A. (2001). Logic control and reactive systems: Algorithmization and programming. *Automation and Remote Control*, 62, no. 1, 1–29.

Ulyantsev, V. and Tsarev, F. (2012). Extended finite-state machine induction using sat-solver. In *Proceedings of the 14th IFAC Symposium "Information Control Problems in Manufacturing — INCOM'12"*, 512–517. IFAC.