ISSN 0005-1179, Automation and Remote Control, 2016, Vol. 77, No. 3, pp. 473–484. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © D.S. Chivilikhin, V.I. Ulyantsev, A.A. Shalyto, 2016, published in Avtomatika i Telemekhanika, 2016, No. 3, pp. 137–151.

LOGICAL CONTROL

Modified Ant Colony Algorithm for Constructing Finite State Machines from Execution Scenarios and Temporal Formulas

D. S. Chivilikhin, V. I. Ulyantsev, and A. A. Shalyto

ITMO University, St. Petersburg, Russia
e-mail: chivdan@rain.ifmo.ru, ulyantsev@rain.ifmo.ru, shalyto@mail.ifmo.ru

Received December 10, 2014

Abstract—We solve the problem of constructing extended finite state machines with execution scenarios and temporal formulas. We propose a new algorithm *pstMuACO* that combines a scenario filtering procedure, an exact algorithm *efsmSAT* for constructing finite state machines from execution scenarios based on a reduction to the Boolean satisfiability problem, and a parallel ant colony algorithm *pMuACO*. Experiments show that constructing several initial solutions for the ant colony algorithm with reduced sets of scenarios significantly reduces the total time needed to find optimal solutions. The proposed algorithm can be used for automated construction of reliable control systems.

DOI: 10.1134/S0005117916030097

1. INTRODUCTION

In certain cases, logical control systems are presented with increased reliability requirements. This relates, for instance, to systems used in aviation, space industry, and power generation. In these fields, errors in the control program may be unacceptable. One can reach a high level of reliability for control programs with verification. One approach to program verification is *model checking* [1], where one begins by constructing a model of the program being tested. Note that in the general case the model is not equivalent to the original program since it is constructed either by hand or heuristically. Properties that a model of the tested program should satisfy are written using a temporal logic language. To test whether the model in question satisfies these properties one uses special software, namely verifiers such as *NuSMV* (http://nusmv.fbk.eu/) or *SPIN* (http://spinroot.com).

One approach to constructing logical control systems is programming with explicit state identification, or *automata programming* [2, 3]. The essence of this approach is to represent the logic of a program's operation with one or several interacting extended finite state machines. One advantage of automata programming over other approaches is the higher level of automation available for verification of automata program with model checking [4]. This advantage is due to the fact that an exact model of an automata program can be constructed automatically.

In some cases, automata programs can be constructed by hand, but this is a tedious process, and programs constructed by humans are often suboptimal. This is not surprising since in some cases problems of constructing finite state machines are NP-hard [5, 6]. Therefore, currently developed methods for their automated construction use metaheuristic optimization algorithms [7–9], e.g., genetic [10] and ant colony [11] algorithms, and methods based on reducing these problems to other NP-hard problems, e.g., the Boolean satisfiability problem SAT [12, 13] and the constraint satisfaction problem CSP [14].

Let us give an example where an automata program can be constructed automatically. Suppose that we know certain examples of the necessary automata program's behavior and some of its more general properties written in a temporal logic language. The approach proposed in [10] lets one use a genetic algorithm to automatically construct an extended finite state machine that satisfies these behavior samples and temporal properties.

An important problem in all methods of automatic finite state machine construction is the fact that in order to construct even small finite state machines (up to ten states) one may need up to several hours of computation on a desktop computer. The work [15] proposes a parallel algorithm based on an ant colony algorithm [11, 16]. This algorithm lets one use parallel computations to reduce the time needed for constructing finite state machines by several times.

This work continues the work [15]. We solve the problem of constructing extended finite state machines (EFSMs) from execution scenarios and temporal formulas. We propose a new algorithm *pstMuACO* that combines parallel ant colony algorithm *pMuACO* [15], exact algorithm for constructing extended finite state machines from execution scenarios *efsmSAT* [13], which is based on solving SAT, and a scenario filtering procedure. It follows from [6] that the problem at hand is at least NP-hard, which justifies our choice of algorithms to solve it. Our experiments have shown that the new algorithm lets one construct FSMs significantly faster than other known algorithms.

2. PROBLEM SETTING FOR CONSTRUCTING AN EXTENDED FINITE STATE MACHINE

An extended finite state machine [2] is a 7-tuple $\langle X, E, Y, Z, y_0, \phi, \delta\rangle$, where X is the set of Boolean input variables, E is the set of input events, Y is the set of states, $y_0 \in Y$ is the initial state, Z is the set of output actions, $\phi\colon Y \times E \times 2^X \to Y$ is the transition function, and $\delta\colon Y \times E \times 2^X \to Z^*$ is the output function. Thus, in this work we consider FSMs each of whose transitions is labeled with an input event, a Boolean formula over input variables, and a sequence of output actions. The semantics for the operation of finite state machines is as follows. Suppose that the FSM is at state y. When the next input event $e \in E$ arrives, the FSM checks whether there exists a transition from state y labeled by event e. If such a transition exists, and the Boolean formula that labels it is satisfiable under current values of input variables, then the FSM transitions into the new state y' and sends the sequence of output actions that labels this transition as output.

A sample extended finite state machine with three states is shown on Fig. 1. Each transition is labeled with an input event from $E = \{e_0, e_1, e_2\}$, a Boolean formula of a unique input variable x, and a sequence of output actions from $Z = \{z_0, z_1, z_2\}$. The initial state is marked with a bold frame.

One of the possible types of input data in the problem of constructing finite state machines from their specifications are sample behaviors that the user wants to observe in the program. Such sample behaviors may include test examples, work scenarios, or negative scenarios [17]. The second type of input data are temporal formulas that the program must satisfy. In this work we consider execution scenarios as sample behaviors, and temporal formulas are specified in the language of Linear Time Logic (*LTL*).

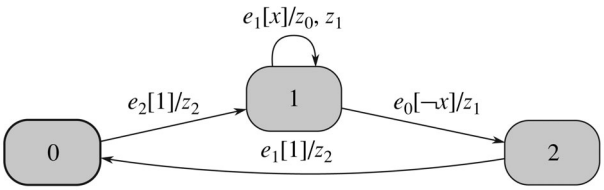

Fig. 1. A sample controlling finite state machine.

An *execution scenario* s_i is a sequence of triples $\{\langle e_i^j, \varphi_i^j, O_i^j\rangle\}_{j=0}^{l_i-1}$, called *scenario elements*, where l_i is the number of elements in scenario s_i, $e_i^j \in E$ is the input event, $\varphi_i^j \colon 2^X \to \{0,1\}$ is a Boolean formula over the input variables, and $O_i^j \in Z^*$ is a sequence of output actions. We say that an FSM satisfies a work scenario $\langle e_i^j, \varphi_i^j, O_i^j\rangle$ in state y if in this state there exists a transition labeled by event e_i^j, a sequence of output actions O_i^j, and a formula equal to φ_i^j as a Boolean formula.

Let us describe how the FSM processes an execution scenario. Scenario elements are processed one by one. In processing a scenario element, we check whether the FSM has a transition from the current state that satisfies this element. If such a transition exists, the FSM transitions to the new state and sends the sequence of output actions written on the transition as output. Thus, processing a scenario generates a path in the FSM which represents the sequence of visited states.

An FSM satisfies an execution scenario if it satisfies all scenario elements in the corresponding states of this path. For instance, the FSM depicted on Fig. 1 satisfies scenario $\langle e_2, 1, (z_2)\rangle\langle e_1, x, (z_0, z_1)\rangle\langle e_0, \neg x, (z_1)\rangle$, but does not satisfy scenario $\langle e_2, 1, (z_2)\rangle\langle e_2, \neg x, (z_1)\rangle$.

An *LTL*-formula includes propositional variables characteristic for the specific problem, logical operators $(\wedge, \vee, \neg)$, and temporal operators such as $\boldsymbol{G}$*lobally* (at any moment of time), *ne*$\boldsymbol{X}$*t* (at the next moment of time), $\boldsymbol{F}$*uture* (some time in the future), $\boldsymbol{U}$*ntil*, and $\boldsymbol{R}$*elease*. Formulas that we consider in this work contain the following propositional variables:

—$\forall e \in E$: wasEvent(e) means that there has been a transition labeled with input event e;

—$\forall z \in Z$: wasAction(z) means that there has been a transition labeled with output action z.

The FSM depicted on Fig. 1, satisfies *LTL*-formula $\boldsymbol{G}(\neg\text{wasEvent}(e_0) \vee \boldsymbol{F}(\text{wasEvent}(e_2) \wedge \text{wasAction}(z_2)))$ that states that if there has been a transition labeled by event e_0 then some time in the future there will be a transition labeled by event e_2 and output action z_2. Formula $\boldsymbol{G}\,(\neg\text{wasEvent}(e_2) \vee \boldsymbol{X}\,(\text{wasEvent}(e_1)))$ does not hold for this FSM since after a transition with event e_2 there may be either a transition with event e_1 or with event e_0.

In this work, we solve the problem of constructing an EFSM with a given number of states that satisfies a given set of execitopm scenarios S and a set of *LTL*-formulas. The considered algorithms for constructing finite state machines perform a guided search of candidate solutions. To evaluate how well a solution candidate satisfies given set of scenarios and *LTL*-formulas, we use the corresponding *fitness function*.

3. THE FITNESS FUNCTION

To evaluate how well finite state machines satisfy a given set of scenarios S and *LTL*-formulas, we use a fitness function (FF) proposed in [17]. This FF is based on the edit distance [18], an automata program verifier [17], and has the form

$$F = F_{tests} + F_{LTL} + \frac{M - n_{transitions}}{100M},$$

where $n_{transitions}$ is the number of all transitions of the FSM, and M is a number that is guaranteed to exceed $n_{transitions}$. In the experiments, we used M =100.

The first component of the FF F_{tests} estimates how well the FSM satisfies a given set of execution scenarios S. In computing the FF value on the FSM, each scenario s_i is processed as follows. The FSM receives as input pairs of input events and Boolean formulas of the scenario s_i: $\langle e_i^0, \varphi_i^0\rangle, \langle e_i^1, \varphi_i^1\rangle, \ldots, \langle e_i^{l_i-1}, \varphi_i^{l_i-1}\rangle$. After processing each such pair, the FSM outputs some sequence of output actions. As a result, for the ith scenario we get a sequence of sequences $A_i = A_i^0, \ldots, A_i^{l_i-1}$. With the resulting output sequence A_i and reference sequence $O_i = O_i^0, \ldots, O_i^{l_i-1}$ written in the scenario, we compute Levenstein's edit distance ED(O_i, A_i), whose original version for binary strings

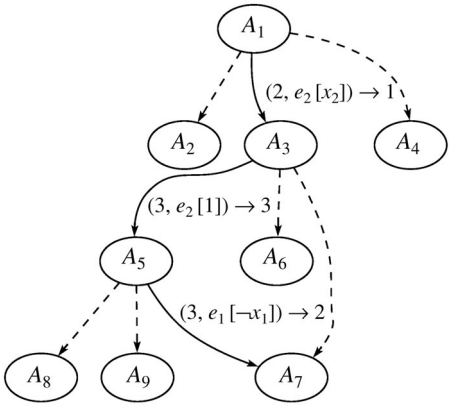

Fig. 2. A sample mutation graph.

was proposed in [18]. The expression for F_{tests} looks like

$$F_{tests} = \frac{1}{|S|} \sum_{i=0}^{|S|-1} \left(1 - \frac{\mathrm{ED}\,(O_i, A_i)}{\max\left(\mathrm{len}\,(O_i), \mathrm{len}\,(A_i)\right)}\right),$$

where $|S|$ is the number of work scenarios, $\mathrm{len}(p)$ is the length of sequence p, and $\mathrm{ED}\,(p_1, p_2)$ is the Levenstein's edit distance between sequences p_1 and p_2.

The second component of FF F_{LTL} evaluates how well the FSM corresponds to given temporal formulas. Verifier developed in [17] lets us single out FSM transitions that definitely do not occur in the counterexample. We call such transitions *verified*. The contribution of the ith *LTL*-formula is computed as the number of verified transitions $t^i_{checked}$ divided by the number of transitions $t^i_{reachable}$ reachable from the initial state. F_{LTL} is computed as the average of this value over all *LTL*-formulas:

$$F_{LTL} = \frac{1}{k} \sum_{i=0}^{k-1} \frac{t^i_{checked}}{t_{reachable}},$$

where k is the number of *LTL*-formulas. Note that if an FSM satisfies a formula then the number of verified transitions equals the number of reachable transitions, and consequently, the maximal value of F_{LTL} equals one.

The presence of the third FF component is determined by the fact that FSMs with few transitions are considered preferable to FSMs with more transitions.

4. ANT COLONY ALGORITHM BASED ON THE MUTATION GRAPH

In this section, we show a brief description of the algorithm *MuACO* whose complete version can be found in [11]. *MuACO* (*Mutation-based Ant Colony Optimization*) is a metaheuristic search optimization algorithm designed to construct finite state machines. The operation of *MuACO* is based on the so-called *mutation graph* whose vertices correspond to FSMs and edges correspond to *mutations* of finite state machines. A mutation is a small change in the FSM structure caused by an application of the *mutation operator*. When solving the problem of constructing extended finite state machines from execution scenarios and temporal formulas, *MuACO* employs two mutation operators. The first chooses a random transition in the FSM and changes the state y where this transition is leading to. The new state is chosen uniformly at random among all states except y. The second operator sequentially considers all FSM states and with a certain probability adds a new transition or deletes an existing transition in each state.

A sample mutation graph is shown on Fig. 2. The label $(3, e_1[\neg x_1]) \to 2$ on an edge means that the corresponding mutation has changed the state where transition from state 3 along event e_1 and formula $\neg x_1$ leads to state 2.

Each edge uv of the mutation graph (u and v are graph vertices corresponding to solution candidates) is assigned with heuristic information values η_{uv} and pheromone values τ_{uv}. Heuristic information on an edge uv is computed as $\eta_{uv} = \max(\eta_{\min}, F(v) - F(u))$, where $\eta_{\min} = \text{const} = 10^{-3}$. The pheromone values τ_{uv} initially equal $\tau_{\min} = \text{const} = 10^{-3}$ and change during the algorithm's operation.

We will consider the maximization problem for the FF. The algorithm begins from a unique initial solution that can be either randomly generated or served as input. This solution becomes the first vertex of the mutation graph. The algorithm looks for solutions with a colony of N_{ants} ants. On each iteration of the colony, ants move along the mutation graph. At the beginning of every step, an ant is located at a certain vertex u and chooses which vertex v to move to. For this purpose, the ant does one of the following:

- with the roulette method [19], the ant chooses the next vertex v among the set of vertices N_u incident to u. The probability p_{uv} to choose vertex v is computed with the following formula, classical in ant colony algorithms [20]:

$$p_{uv} = \frac{\tau_{uv}^{\alpha} \times \eta_{uv}^{\beta}}{\sum\limits_{w \in N_u} \tau_{uw}^{\alpha} \times \eta_{uw}^{\beta}},$$

  where $v \in N_u$, and $\alpha, \beta \in [0, 1]$ reflect the importance of pheromone values and heuristic information respectively;

- by applying mutation operators, create N_{mut} new finite state machines. All new FSMs are added to the graph as children of vertex v. The ant moves to the vertex corresponding to an FSM with the largest value of FF.

Each ant stores a counter and the FF value $f_{ant}^{\max}$ of the best FSM it has found. If at the present step the ant has found a solution whose FF value is larger than $f_{ant}^{\max}$, the value $f_{ant}^{\max}$ is updated and the counter remains unchanged; otherwise the counter is incremented by one. When the counter value becomes equal to n_{stag}, the ant is forced to stop.

A similar procedure is performed for the ant colony: one stores a counter and the maximal FF value of an FSM found by the ants. If at the present iteration of the colony this FF value does not increase, the counter is incremented by one. If the counter's value reaches N_{stag}, we assume that the algorithm has come to stagnation, and the algorithm is restarted.

At the end of each ant colony iteration, we update the pheromone values on all edges of the mutation graph. Apart from heuristic information and pheromone values, for each edge we store the value $F_{uv}^{\max}$ corresponding to the largest value FF of FSM found by an ant that visited this edge.

Consider the paths of all ants on a given iteration. The FF value for an ant is the FF value of the best FSM found by this ant. For each ant's path, we find a segment from the beginning to the vertex corresponding to the best FSM along this path. For all edges in this segment, we update the values $F_{uv}^{\max}$, then update the pheromone values:

$$\tau_{uv} = \max(\tau_{\min}, (1 - \rho)\tau_{uv} + F_{uv}^{\max}),$$

where $\rho \in [0, 1]$ is the pheromone's evaporation rate.

5. PARALLEL ANT COLONY ALGORITHM *PMUACO*

In this section, we show a parallel ant colony algorithm proposed in [15]. This algorithm is intended to be used on multicore computers with shared memory but can also be extended to run on a cluster. We call this algorithm *pMuACO* (*parallel MuACO*).

Suppose that the parallel algorithm has access to m execution threads. In each thread, we run the *MuACO* algorithm, generating a separate initial solution randomly for each thread. It has been shown in [15] that interaction between individual threads in an algorithm significantly improves its efficiency. We have implemented two interaction mechanisms based on an archive of K best solutions for all threads. At every time moment, the K_i cell stores the best solution found by the ith algorithm.

The first interaction mechanism is activated when the ith *MuACO* algorithm reaches stagnation and is restarted. The starting solution is no longer randomly generated but chosen from the archive of best solutions $K \setminus \{K_i\}$.

The second mechanism works on each ant colony iteration of every *MuACO* algorithm; it uses the genetic *crossover* operator for extended finite state machines. The crossover operator receives two FSMs (parents) as input and returns two FSMs (children). Each child contains transition chosen from both parents. The work [15] uses a crossover operator for finite state machines proposed in [10] that takes into account which transitions in the parents have been used to compute the FF value.

Before the beginning of an iteration of the ith algorithm, we choose a solution K_j, $j \neq i$ randomly from the archive of best solutions. The crossover operator is applied to K_i and K_j; it returns two FSMs A_1 and A_2. Then, we choose the finite state machine whose FF value is larger, say A_{best}. The A_{best} FSM is added to the mutation graph of the ith *MuACO* algorithm as a child of the vertex associated with K_i. In the subsequent iteration of the ith algorithm, one ant will start from the vertex associated with A_{best}, and the other ants, as before, from the vertex associated with the currently best solution found by the ith algorithm K_i.

6. PROPOSED MODIFICATION OF THE PARALLEL ANT COLONY ALGORITHM

In this work, we propose to generate initial solutions for the parallel algorithm *pMuACO* from reduced sets of execution scenarios. This modification is an extension of the approach proposed in [14], where to get an initial approximation we use the exact method of constructing extended finite state machines from execution scenarios. Exact methods are based on reducing the problem of constructing extended finite state machines with work scenarios to the Boolean satisfiability problem (SAT) and the constraint satisfaction problem (CSP). To solve the SAT and CSP problems, we use third party software such as *cryptominisat* (https://github.com/msoos/cryptominisat) and *Choco* (http://choco-solver.org/). It has been shown in [14] that joint application of the exact algorithm and ant colony algorithm [11] significantly reduces the time needed to construct finite state machines.

An obvious development of this approach is to use it to get an initial approximation in parallel algorithms. We call this algorithm *psMuACO* (*parallel SAT MuACO*). First, with an exact algorithm we construct a finite state machine satisfying all scenarios. Then we use this FSM as an initial solution for the *pMuACO* algorithm. However, we note that parallel combinatorial optimization algorithms reach maximal efficiency if the search in different threads starts from different initial solutions. This is caused by the fact that for different initial solutions the search space covered by the algorithm per unit of time increases significantly. If we use exact software solutions for SAT and CSP, the algorithms are usually deterministic: the same set of scenarios always leads to the same FSM. To use this approach together with a parallel ant colony algorithm, we have to use algorithms based on SAT or CSP to get several different finite state machines.

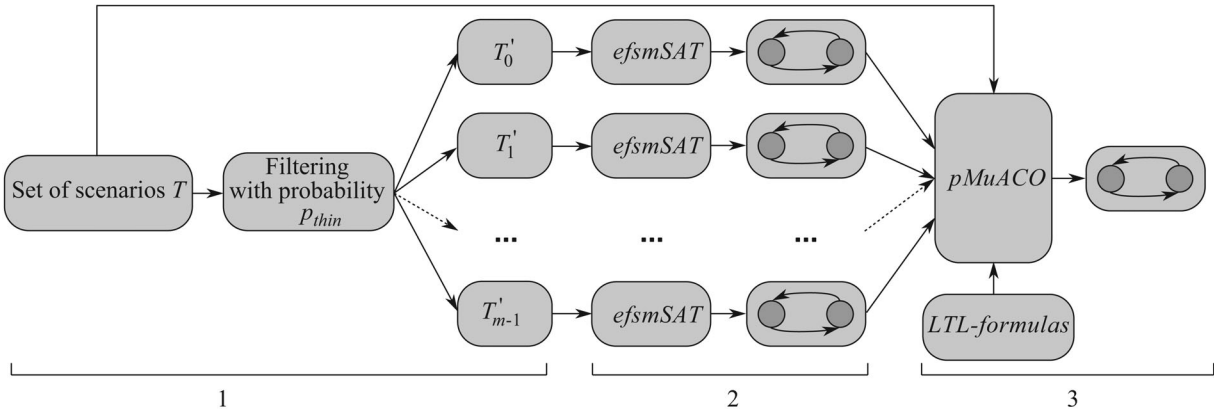

Fig. 3. Flowchart of the proposed *pstMuACO* algorithm.

We propose to solve this problem by constructing initial solutions with reduced sets of scenarios obtained from the original set with a filtering procedure. The algorithm consists of three stages.

1. **Scenario filtering procedure**. With a set of scenarios T, generate n_{start} different finite state machines. For this purpose, we first construct n_{start} reduced sets of scenarios $T'_0, \ldots, T'_{n_{start}-1}$. Each such set is obtained from the original set T by filtering: removing each scenario with probability p_{thin}.
2. **Constructing initial solutions with the** *efsmSAT* **method**. Each ith FSM is constructed from a reduced set of scenarios T'_i with the *efsmSAT* method based on a reduction to the SAT problem [13]. The FSMs are constructed independently by running *cryptominisat* in parallel to solve SAT problems. This approach does not guarantee different finite state machines, but in experiments the share of identical finite state machines has proven to be near zero.
3. **Constructing the FSM with the** *pMuACO* **algorithm**. In case when $n_{start} = m$, the ith FSM constructed with the reduced set of scenarios T'_i is used as an initial solution for the ith thread of the algorithm. If $n_{start} < m$, the initial solution in each thread is chosen randomly out of the set of generated initial solutions.

The number of finite state machines n_{start} generated with SAT and the probability of deleting a scenario p_{thin} are parameters of the algorithm. The flowchart of the proposed algorithm *pstMuACO* (*parallel SAT thinned out MuACO*) is shown on Fig. 3.

7. EXPERIMENTAL STUDY

In this section, we present computational experiments, the process of data preparation, and the choice of values for algorithm parameters. To tune the values of parameters, we have used a server with a 24-core AMD Opteron(TM) processor 6234 @ 1.4 GHz, and all computational experiments have been performed on a server with 64-core AMD Opteron(TM) processor 6378 @ 2.4 GHz.

7.1. Data Preparation

For our computational experiments, we have prepared two input datasets: a training set and a test set. Each of them is a set of examples, and each example consists of a number of execution scenarios and a set of *LTL*-formulas. To generate such an example, we first randomly generate an FSM with the following parameters: number of states N, number of input events $|E| = 2$, number of input variables $|X| = 1$, number of output actions $|Z| = 2$, and sequences of output actions on the transitions contain zero to two actions.

Using this FSM, we construct a set of work scenarios. Each scenario corresponds to a random path in the FSM. The beginning of each scenario coincides with the FSM's initial state. The length

of the scenario equals the number of transitions occurring in the path. In total, we generate $5N$ scenarios of total length $100N$.

Construction of the *LTL*-formulas that FSM A satisfies was done as follows. The formula construction algorithm receives as input the FSM A for which the formulas are to be constructed and the desired number of formulas n_{LTL} (here $n_{LTL} = 2$). First, with the *randltl* software [21] we generate $100n_{LTL}$ random LTL-formulas, where the number of vertices in each formula tree does not exceed 15. Next, with a verifier [17] we find only formulas that A satisfies. For further testing of each such formula, we generate 50 random finite state machines and test for each machine whether the formula holds for it. The formula is added to the final set of formulas if it holds for at most half of those finite state machines. The formula generation process halts when we get the necessary number of formulas.

7.2. Tuning Algorithm Parameters

Keeping in mind that computational experiments should produce fair results, we chose the parameters of the algorithms under comparison not by hand but with an automated procedure called *parameter tuning*. For this purpose, we used the *irace* software [22]. The parameter tuning procedure consists of the following. Suppose we have to find good values of parameters for an algorithm intended to solve problem P. The tuning program (*irace*) receives as input:

- a description of the parameters for the tuned algorithm: name, type, range of admissible values;
- a set of examples I for problem P;
- a time constraint on the operation of the tuning program.

Sets of algorithm parameter values are called *configurations*. At first, *irace* generates a set of random configurations. On each iteration, inefficient configurations are discarded with the following mechanism. The tuned algorithm is parameterized with a configuration and sequentially run on several examples from I, computing the problem's objective function value P on every run. A configuration is considered inefficient and is discarded if the set of objective function values for this configuration is statistically worse than for some other configuration.

7.3. Studying the Properties of the Proposed Modified Algorithm

Here we describe the first set of experiments; its purpose was to study the properties of the proposed *pstMuACO* algorithm. We did not tune the algorithm's parameters separately, using the same values as in [15]: $N_{ants} = 4$, $N_{stag} = 28$, $n_{stag} = 45$, $N_{mut} = 44$, $\rho = 0.52$. We have used

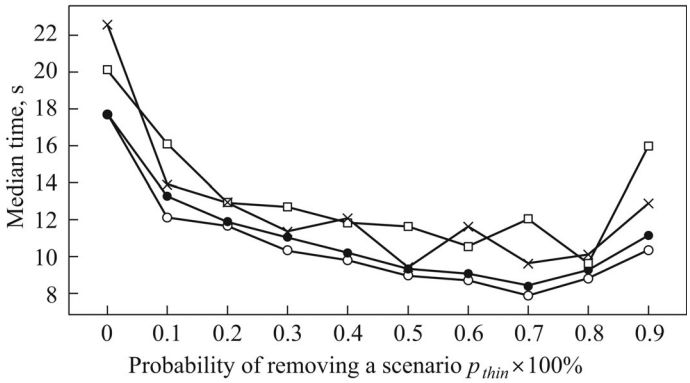

Fig. 4. Graphs showing how running time depends on the probability of removing a scenario: $n_{start} = 1$ (□), 2 (×), 6 (•), 16 (○).

a set of 100 examples each of which was generated from an FSM with ten states. The algorithm had 16 execution thread at its disposal. In the first experiment, the probability of filtering out a scenario p_{thin} varied from zero to 0.9 with step 0.1, the number of finite state machines generated with SAT n_{start} varied from zero to 16. Graphs that show how the median algorithm execution time depends on p_{thin} for different values of n_{start} are shown on Fig. 4.

All values of n_{start} led to similar behavior: the median execution time first decreases as p_{thin} increases, and then, as we come closer to the maximal value of p_{thin}, begins to increase again. Experimental results indicate that the values of $p_{thin} = 0.7$ and $n_{start} = 16$ are optimal for our values of algorithm parameters.

7.4. Comparison with Previous Solutions

Likewise, the purpose of the second set of experiments was to compare the algorithm *pstMuACO* proposed in this work with algorithm *psMuACO* and algorithm *pMuACO* that have been proposed earlier in [15]. Parameter values for the algorithms were tuned with *irace*, with 24 hours of machine time allocated for tuning each algorithm. The training set for the tuning consisted of 200 examples constructed with FSMs with 10–20 states. The resulting values of algorithm parameters are shown in Table 1. In all algorithms, values of parameters α and β were equal to one.

Table 1. Values of algorithm parameters obtained with *irace*

Parameter	*pMuACO*	*psMuACO*	*pstMuACO*
Maximal number of ant steps before increasing the FF value n_{stag}	34	41	22
Maximal number of ant colony iterations before increasing the FF value N_{stag}	33	35	17
Number of mutations N_{mut}	44	27	44
Number of ants N_{ants}	4	4	17
Pheromone evaporation rate ρ	0.2	0.6	0.21
Number of solutions generated with SAT n_{start}	–	–	7
Probability of filtering out a scenario p_{thin}	–	–	0.48

Note that the resulting values of new parameters $p_{thin} = 0.48$ and $n_{start} = 7$ differ from the values considered optimal in the previous experiment. This is because in this experiment we tuned the parallel algorithm *pstMuACO*, while in the previous one we took some values of parameters for the *MuACO* algorithm and found optimal values of new parameters for it.

Testing was performed with three sets of 50 examples each constructed with FSMs with 10, 15, and 20 states respectively. Figures 5a and 5b show how the median execution time for the algorithms depends on the number of states. The graphs show that execution time of algorithm *psMuACO* is

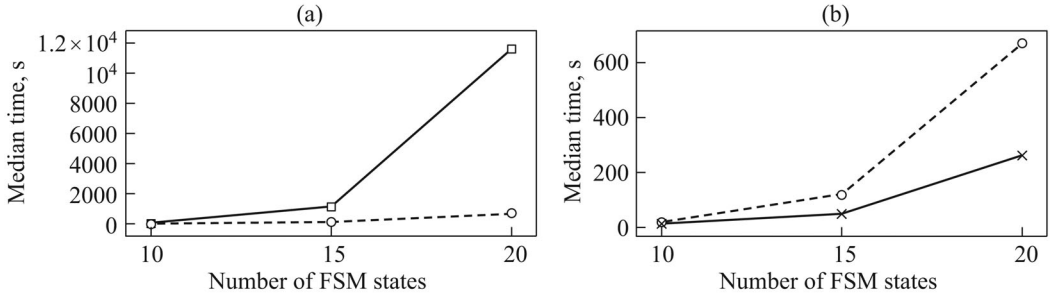

Fig. 5. Execution time of the algorithms as a function of the number of states in the FSM: *pMuACO* (□), *psMuACO* (○), *pstMuACO* (×).

Table 2. Median execution time of the *pstMuACO* algorithm on data with increased dimension, seconds; TL indicates that the run did not finish in time allotted.

Number of states	5	6	7	8	9	10
$\|X\| = 1, \|Z\| = \|E\| = 2$	3.2	4.2	5.5	5.4	6.5	13.3
$\|X\| = \|Z\| = \|E\| = 3$	47.0	38.6	74.9	176.1 (19 TL)	TL	TL

significantly less than the median execution time for algorithm *pMuACO*. Algorithm *pstMuACO*, in turn, runs faster than *psMuACO*. For instance, for finite state machines with 20 states the median execution time of algorithm *pstMuACO* was 2.5 times less than the median execution time of algorithm *psMuACO* and 17 times less than the median execution time of algorithm *pMuACO*. We also note that for finite state machines with 20 states in six out of 50 cases algorithms *pMuACO* and *psMuACO* did not finish in 24 hours and were interrupted. However, even if they did finish in some longer time, it would not change the median execution time value, and the graphs on Figs. 5a and 5b would remain the same.

On the other hand, we also computed the average execution time for examples on which all algorithms finished successfully within 24 hours. For instance, for finite state machines with 20 states this value for the *pMuACO* algorithm was 20 588 seconds; for algorithm *psMuACO*, 5421 s; for algorithm *pstMuACO*, 491 s. Thus, the proposed algorithm *pstMuACO* is, on average, more than 40 times faster than algorithm *pMuACO* and more than 11 times faster than algorithm *psMuACO*.

To test statistical significance of the differences in algorithms, we have used the Wilcoxon test [23]. The test was run separately for each number of states and each pair of algorithms. The null hypothesis was that the median of the samples' difference was zero. The alternative hypotheses was that it was less than zero. The confidence level was set to 0.05. For the pair of algorithms *pMuACO* and *psMuACO* we obtained the following p-values: 2.07×10^{-6} (10 states), 7.14×10^{-5} (15 states) and 1.25×10^{-3} (20 states). For the pair of algorithms *psMuACO* and *pstMuACO* we obtained the following p-values: 0.03 (10 states), 5.44×10^{-7} (15 states), and 1.95×10^{-5} (20 states). These results indicate that for the considered data the *psMuACO* algorithm is in all cases statistically faster than algorithm *pMuACO*, and the *pstMuACO* algorithm is statistically faster than the *psMuACO* algorithm.

Finally, we performed an experimental study on data of increased dimension. We considered FSMs with five to ten states, three input events, three output variables, and three output actions. Sequences of output actions on the transitions contained zero to three actions. We set a time limit on each experiment of 300 seconds. Results of these experiments are shown in Table 2. Judging by the data, we can conclude that increased input data dimension leads to a significant increase in the time needed to construct the optimal solution. For instance, for N equal to five, six, and seven the median execution time of algorithm *pstMuACO* for data with increased dimension is approximately 10 times larger than the execution time on small dimension data. For large values of N, most runs did not finish in time allotted.

8. CONCLUSION

In this work, we propose a new parallel algorithm for constructing extended finite state machines *pstMuACO*. Our experiments have shown that the new algorithm is faster than all previously used approaches. For instance, for the considered finite state machines with 20 states the new algorithm is, on average, 40 times faster than a simple parallel ant colony algorithm. The proposed algorithm can be used for automated construction of reliable control systems given that the finite state machines in question have relatively small dimension.

ACKNOWLEDGMENTS

This work was supported by the Government of Russian Federation, grant no. 074-U01 and the Russian Foundation for Basic Research, project no. 14-01-0055114 a.

REFERENCES

1. Clarke, E., Grumberg, O., and Peled, D., *Model Checking*, Cambridge: MIT Press, 1999.
2. Polikarpova, N.I. and Shalyto, A.A., *Avtomatnoe programmirovanie* (Automata Programming), St. Petersburg: Piter, 2009.
3. Shalyto, A.A., Algorithmic Graph Schemes and Transition Graphs: Their Use in Software Realization of Logical Control Algorithms. II, *Autom. Remote Control*, 1996, vol. 57, no. 7, part 2, pp. 1027–1045.
4. Vel'der, S.E., Lukin, M.A., Shalyto, A.A., et al., *Verifikatsiya avtomatnykh programm* (Verification of Automata Programs), St. Petersburg: Nauka, 2011.
5. Gold, M., Complexity of Automaton Identification from Given Data, *Inf. Control*, 1978, vol. 37, no. 3, pp. 302–320.
6. Ul'yantsev, V.I., Applying Method for Solving the Boolean Satisfiability Problem for Constructing Controlling Finite State Machines with Work Scenarios, Bachelor's Thesis, SPSU IFMO, 2011, available at http://is.ifmo.ru/diploma-theses/2011/bachelor/ulyantsev/thesis.pdf.
7. Luke, S., *Essentials of Metaheuristics*, Raleigh: Lulu, 2009.
8. Karpenko, A.P., *Sovremennye algoritmy poiskovoi optimizatsii. Algoritmy, vdokhnovlennye prirodoi*, (Modern Search Optimization Algorithms: Algorithms Inspired by Nature), Moscow: Mosk. Gos. Tekhn. Univ. im. N.E. Baumana, 2014.,
9. Kureichik, V.V., Kureichik, V.M., and Rodzin, S.I., *Teoriya evolyutsionnykh vychislenii* (Theory of Evolutionary Computation), Moscow: Fizmatlit, 2012.
10. Tsarev, F. and Egorov, K., Finite State Machine Induction Using Genetic Algorithm Based on Testing and Model Checking, *Proc. 13th Ann. Conf. on Genetic and Evolutionary Computation Companion*, New York, 2011, pp. 759–762.
11. Chivilikhin, D. and Ulyantsev, V., MuACO*sm*: A New Mutation-Based Ant Colony Optimization Algorithm for Learning Finite-State Machines, *Proc. 15th Ann. Conf. on Genetic and Evolutionary Computation*, New York, 2013, pp. 511–518.
12. Heule, M. and Verwer, S., Exact DFA Identification Using SAT Solvers, *Proc. 10th Int. Colloquium Conf. on Grammatical Inference: Theoretical Results and Applications*, Berlin, 2010, pp. 66–79.
13. Ulyantsev, V. and Tsarev, F., Extended Finite-State Machine Induction Using SAT Solver, *Proc. 10th Int. Conf. on Machine Learning and Applications*, Los Alamitos, 2011, vol. 2, pp. 346–349.
14. Chivilikhin, D., Ulyantsev, V., and Shalyto, A., Combining Exact and Metaheuristic Techniques for Learning Extended Finite-State Machines from Test Scenarios and Temporal Properties, *Proc. 13th Int. Conf. on Machine Learning and Applications*, Detroit, 2014, pp. 350–355.
15. Chivilikhin, D., Ulyantsev, V., and Shalyto, A., Extended Finite-State Machine Inference with Parallel Ant Colony Based Algorithms, *Proc. 5th Int. Student Workshop on Bioinspired Optimization Methods and Their Applications*, Ljubljana, 2014, pp. 117–126.
16. Chivilikhin, D. and Ulyantsev, V., Inferring Automata-Based Programs from Specification with Mutation-Based Ant Colony Optimization, *Proc. 16th Conf. on Genetic and Evolutionary Computation Companion*, New York, 2014, pp. 67–68.
17. Egorov, K.V., Generating Controlling Finite State Machines with Genetic Programming and Verification, PhD Dissertation, SPSU IFMO, 2013, available at http://is.ifmo.ru/disser/egorov_disser.pdf.
18. Levenshtein, V.I., Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, *Soviet Physics Dokl.*, vol. 10, no. 8, pp. 707–710, 1966.

19. Baker, J., Reducing Bias and Inefficiency in the Selection Algorithm, *Proc. 2nd Int. Conf. on Genetic Algorithms and their Applications*, Hillsdale, 1987, pp. 14–21.

20. Dorigo, M., Maniezzo, V., and Colorni, A., Ant System: Optimization by a Colony of Cooperating Agents, *IEEE Trans. Systems, Man Cybernetics*, 1996, vol. 25, no. 1, pp. 29–41.

21. Duret-Lutz, A., Manipulating LTL Formulas using SPOT 1.0., in *Automated Technology for Verification and Analysis*, *Lect. Notes in Computer Science*, Hung, D. and Ogawa, M., Eds., 2013, vol. 8172, pp. 442–445.

22. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., et al., *The* `irace` *Package, Iterated Race for Automatic Algorithm Configuration*, Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, 2011.

23. Wilcoxon, F., Individual Comparisons by Ranking Methods, *Biometrics Bull.*, 1945, vol. 1, no. 6, pp. 80–83.

This paper was recommended for publication by O.P. Kuznetsov, a member of the Editorial Board