Combining Exact And Metaheuristic Techniques For Learning Extended Finite-State Machines From Test Scenarios and Temporal Properties

Daniil Chivilikhin
ITMO University
St. Petersburg, Russia
49 Kronverksky pr.
email: chivdan@rain.ifmo.ru

Vladimir Ulyantsev
ITMO University
St. Petersburg, Russia
49 Kronverksky pr.
email: ulyantsev@rain.ifmo.ru

Anatoly Shalyto
ITMO University
St. Petersburg, Russia
49 Kronverksky pr.
email: shalyto@mail.ifmo.ru

***Abstract*—This paper addresses the problem of learning extended finite-state machines (EFSMs) from user-specified behavior examples (test scenarios) and temporal properties. We show how to combine exact EFSM inference algorithms (that always find a solution if it exists) and metaheuristics to derive an efficient combined EFSM learning algorithm. We also present a new exact EFSM inference algorithm based on Constraint Satisfaction Problem (CSP) solvers. Experimental results are reported showing that the new combined algorithm significantly outperforms a previously used metaheuristic.**

***Keywords*-constraint satisfaction problem, ant colony optimization, model checking, control, hybrid algorithms, finite-state machines**

I. Introduction

In certain applications that require high software reliability, programs should undergo the process of verification in addition to the traditional testing procedure. Verification allows to check whether certain properties hold for all possible computational states of the program. Model checking [1] is a verification framework in which temporal properties are checked for a manually constructed model of the program instead of the program itself. However, the program and its model may not be equivalent, and proving their equivalence might well be impossible.

Automata-based programming [2] offers a way to build correct-by-design event-driven programs for control applications. In this programming paradigm program logic is expressed using finite-state machines. Programs designed using this paradigm can be automatically transformed to models used in model checking [3].

In this paper we consider the problem of inferring extended finite-state machines (EFSMs) compliant with a specified set of test scenarios and a set of temporal properties expressed in Linear Temporal Logic (LTL). We will call a set of scenarios and LTL formulae a *specification*. To the best of our knowledge, only two algorithms for inferring EFSMs from specification that includes temporal properties currently exist: a genetic algorithm [3] and an ant colony algorithm MuACO*sm* [4] [5]. Both these algorithms are metaheuristics based on the same idea of using a fitness function that takes into account the results of verification.

The state-of-the-art algorithm for inferring EFSMs from test scenarios only is, as far as we know, the approach proposed in [6] based on satisfiability (SAT) problem solvers. In that paper the problem of constructing an EFSM with a fixed number of states from a set of test scenarios is translated to SAT, and efficient SAT-solvers are used to find the solution. A drawback of the SAT approach is that the translation is very complex. Another issue is that the SAT language is very low-level, which makes it impossible to express certain useful constraints.

The contribution of this paper is two-fold.

1) First, we propose a new exact EFSM inference algorithm based on constraint satisfaction problem (CSP) solvers. The translation to CSP is much simpler than to SAT. Also, the CSP language allows to specify constraints that cannot be expressed in SAT.
2) Second, we propose a combined algorithm that first uses the proposed CSP-based algorithm to infer an EFSM from a set of scenarios, ignoring the LTL formulae, and then uses the found solution as the initial one for MuACO*sm*.

In general, the EFSM inferred by the CSP-based algorithm from scenarios will not satisfy all specified LTL formulae. However, taking this solution as the initial one for MuACO*sm* may (and, indeed, does) make the overall process of EFSM inference faster.

The rest of this paper is structured as follows. Section II reviews some related work, in Section III-A a formal description of the problem is given with necessary definitions. Section IV describes the new exact CSP-based EFSM inference algorithm. Section III-B briefly reviews the metaheuristic MuACO*sm* algorithm. In Section V we describe how to combine the CSP and MuACO*sm* algorithms. Section VI describes our experiments and results and Section VII concludes.

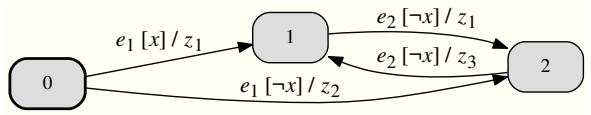


Fig. 1. An example of an EFSM: each transition is marked with an event (e_1, e_2), a Boolean formula over the input variables (x, $\neg x$), and a sequence of output actions (z_1, z_2, z_3)

II. Related Work

A paper most closely related to this one is probably [7]. In the paper a state-merging algorithm is proposed that allows to infer finite-state models from program execution traces, taking into account temporal constraints. The algorithm starts by building an augmented prefix tree acceptor (APTA), which is a tree that contains all traces, and then merges appropriate states of the APTA. Basically, each time two states of the APTA are merged, a model checker is used to see if the merge violates any temporal constraints. If it does, the counterexample is added to the input data and the algorithm is restarted recursively.

The main difference of [7] from the current paper is the type of finite-state models that are inferred. Finite-state machines inferred in [7] are merely models and cannot be used as controllers. The motivation of the paper is, however, different from our motivation: models built in [7] are suggested to be used for verification and validation. In this paper, on the other hand, we concentrate on inferring correct-by-design programs: such programs would require no further verification or testing.

Also, a somewhat relevant approach was proposed in [8] for learning deterministic finite automata (DFA). There a heuristic state-merging algorithm was first applied to the set of training examples, and then the found partial DFA was passed to a SAT-solver based algorithm.

III. Background

A. Learning Extended Finite-State Machines From Test Scenarios and Temporal Properties

In this paper we define an *extended finite-state machine* as a septuple $\langle S, s_0, \Sigma, \Delta, Z, \delta, \lambda \rangle$, where S is a set of states, $s_0 \in S$ is the initial state, Σ is a set of input events and Δ is a set of output actions. Z is a set of Boolean *input variables*, $\delta\colon S \times \Sigma \times 2^Z \to S$ is the *transitions function* and $\lambda\colon S \times \Sigma \times 2^Z \to \Delta^*$ is the *actions function*. An example of an EFSM is shown in Fig. 1. The initial state is bolded. Each transition is marked with an event (e_1, e_2), a Boolean formula over the input variables (x, $\neg x$), and a sequence of output actions (z_1, z_2, z_3).

In this paper we consider EFSM specifications that consist of a set of test scenarios and a set of LTL formulae. A *test scenario* is a sequence of *scenario elements* $\langle e, \varphi, O \rangle$, where $e \in \Sigma$ is an input event, $\varphi \in 2^Z$ is a Boolean formula over the input variables, and $O \in \Delta^*$ is a sequence of output actions. An EFSM is said to be compliant with a scenario element $\langle e, \varphi, O \rangle$ in state s, if in this state it contains a transition marked with event e, sequence of output actions O and a Boolean formula equal to φ. An EFSM is compliant with a scenario if it is compliant with all scenario elements in the corresponding states when the scenario is processed sequentially. For example, the EFSM from Fig. 1 is compliant with scenario $\langle e_1, x, (z_1) \rangle\ \langle e_2, \neg x, (z_1) \rangle$, but is not compliant with scenario $\langle e_1, x, (z_1) \rangle\ \langle e_1, x, (z_1) \rangle$.

The LTL language consists of propositional variables **Prop**, Boolean operators **and**, **or**, **not**, and temporal operators, such as G, F, U, R and X. An LTL formula that expresses a temporal property of an EFSM may contain the following basic propositional variables:

- **wasEvent**$(e), e \in \Sigma$ – indicates a situtation that a transition marked with event e has been triggered;
- **wasAction**$(a), a \in \Delta$ – indicates a situation that a transition marked with action a has been triggered.

The EFSM from Fig. 1 is compliant with LTL formula $G\,(U\,(\textbf{wasEvent}(e_1), \textbf{wasEvent}(e_2)))$, which basically states that a transition marked with event e_1 will always be triggered before a transition marked with event e_2. This is, indeed, true, since both transitions in the initial state are marked with e_1 and all other transitions are marked with e_2. On the other hand, the example EFSM does not comply with LTL formula $G\,(\textbf{wasEvent}(e_1) \to \textbf{wasAction}(z_1))$, since one of the transitions marked with event e_1 is marked with action z_2.

The problem of specification-based EFSM inference can be formulated as: given a set of test scenarios and a set of LTL formulae, find an EFSM with no more than C states compliant with all scenarios and LTL formulae.

B. Learning EFSMs With Mutation-Based Ant Colony Optimization

MuACO*sm* [5] is a metaheuristic for learning finite-state machines, which is based on ant colony optimization [9]. As any metaheuristic algorithm, MuACO*sm* makes use of a fitness function to determine how well a particular candidate solution solves the problem.

The key idea of the algorithm is to represent the search space, which is the set of all EFSMs, as a *mutation graph*, where nodes represent EFSMs and edges represent EFSM *mutations*. A mutation of an EFSM is an operation that causes a rather small change in the EFSM structure. For example, a mutation can change the state a transition leads to.

Each edge (u, v) of the mutation graph has two associated values: heuristic information η_{uv} and pheromone value τ_{uv}. Heuristic information is calculated as $\eta_{uv} = \max\{\eta_{\min}, F(v) - F(u)\}$, where F is the fitness function. Once assigned, heuristic information values are never changed. Pheromone values are initialized with a small positive value and are changed during algorithm execution.

The algorithm starts off with one initial solution that is either generated randomly or can be specified by the user. The initial solution is added to the mutation graph. On each iteration of the algorithm two operations are repeated: building new solutions using a set of ants and pheromone update.

All ants start building solutions from the node associated with the best-so-far solution. Each ant has a limited number of steps, on each step it moves to a new node. The ant can use one of the following operations to select the next node.

1) **Build new solutions**. The ant performs a number of mutations of the solution associated with its current node. It moves to the newly constructed node associated with the solution that has the largest fitness value.
2) **Select from old solutions**. The ant moves to node v from the existing succesor set of the current node u with probability:

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

where $\alpha, \beta \in [0, 1]$.

After all ants finish building solutions, pheromone values of all edges are updated in the way described in [5]. Basically, pheromone values on all graph edges that were visited by ants are increased in proportion to the fitness values of solutions found by the ants. Then, all pheromone values are decreased by a certain value. The algorithm is stopped when it either finds a solution with a large enough fitness value, or it exceeds the time limit. If for a specified number of iterations the algorithm does not improve the best found solution, it is restarted.

IV. EFSM Inference Using CSP Solvers

In this section the new CSP-based algorithm for inferring EFSMs from test scenarios is described. The proposed algorithm is based on the same idea as the previous SAT-based algorithm [6]: find an appropriate coloring of the scenario tree, such that when vertices of the same color are merged, a deterministic EFSM compliant with all test scenarios is constructed.

The only difference is that in the new algorithm we translate the EFSM inference problem to the Constraint Satisfaction Problem (CSP) rather than SAT. This translation is much simpler than the old one and is much more comprehensible due to the fact that the CSP language is a higher level language compared to SAT. The SAT translation uses six types of clauses, while the CSP translation only requires three simple types of constraints. Preliminary experiments showed that the new algorithm does not significantly differ from the old SAT-based approach in terms of efficiency.

In addition, the new translation allows the end user to add constraints that are specific to his problem. For example, the user may request that the target EFSM should be *complete*, i.e. it should contain a transition for each possible state and input event combination. Such constraints cannot be expressed in SAT.

The algorithm consists of five main steps listed below.

1) Scenario tree construction.
2) Consistency graph construction.
3) Constructing a set of constraints on integer variables. The constraints express requirements to the coloring of consistency graph vertices that express the consistency of the sought EFSM's transition system.
4) Solving the set of constraints using a CSP-solver.
5) Constructing an EFSM from a satisfying assignment found by the CSP-solver.

Steps 1 and 2 are identical to the first two steps of the SAT-based algorithm described in [6], so they will be discussed here very briefly.

A. Scenario tree construction

A *scenario tree* is a trie where each edge is marked with an input event, Boolean formula and a sequence of output actions. A scenario tree built from a set of scenarios contains all scenarios and all their prefixes. Basically, a scenario tree is itself an EFSM, but with a large number of states. For example, consider the following set of scenarios:

- $\langle T, x, z_1 \rangle, \langle T, \neg x, z_2 \rangle, \langle T, \neg x, z_3 \rangle, \langle T, x, z_1 \rangle$;
- $\langle T, \neg x, z_2 \rangle, \langle T, \neg x, z_3 \rangle, \langle T, x, z_1 \rangle, \langle T, x, z_1 \rangle$;
- $\langle T, x, z_1 \rangle, \langle T, x, z_1 \rangle, \langle T, \neg x, z_2 \rangle$;
- $\langle T, \neg x, z_2 \rangle, \langle T, \neg x, z_3 \rangle, \langle T, x, z_1 \rangle, \langle T, \neg x, z_2 \rangle$.

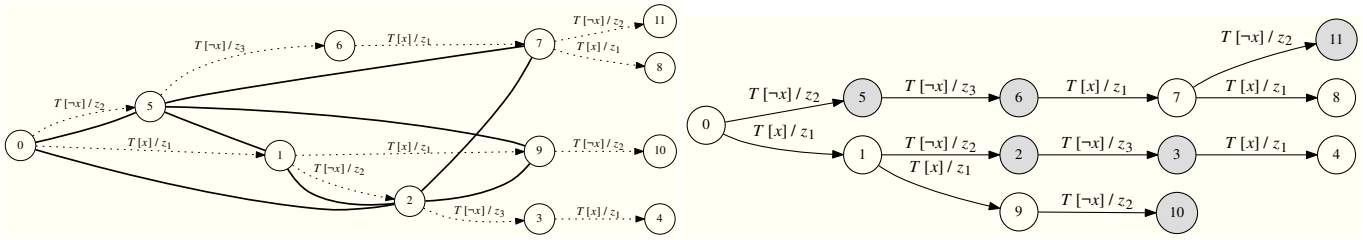
The scenario tree built from this set of scenarios is shown in Fig. 2a with dotted edges.

B. Consistency graph construction

The algorithm is based on computing a specific coloring of the scenario tree, where each color corresponds to one state of the EFSM. In order to formulate constraints on the scenario tree coloring a consistency graph is first constructed. The set of consistency graph vertices is identical to the set of scenario tree vertices. Two consistency graph vertices u and v are connected with an edge (and called *inconsistent*) if there exists a sequence of pairs of events and input variable values $\langle e_1, \mathbf{values}_1 \rangle, \ldots, \langle e_k, \mathbf{values}_k \rangle$ that tells u and v apart. The aforementioned sequence is said to tell u and v apart if all of the following conditions are met:

- there exists a path P_u from vertex u, in which the edges are marked with corresponding events $e_1, \ldots, e_k$ and guard conditions $f_1, \ldots, f_k$ in such a way that the sets of input variable values $\mathbf{values}_1, \ldots, \mathbf{values}_k$ are their satisfying assignments;
- an analogous path P_v exists from vertex v;
- one of the following is true for the last edges of P_u and P_v:
  1) output actions sequences on these edges are different;
  2) guard conditions on these edges have a common satisfying assignment, but they are not equal as Boolean functions.

A dynamic programming algorithm proposed in [6] is used to construct the consistency graph. This algorithm runs in $O(V^2)$, where V is the number of vertices in the scenarios tree. A consistency graph built from the aforementioned scenario tree is shown in Fig. 2a. Dotted edges represent edges of the scenario tree, solid edges belong to the consistency graph. Vertices 2 and 5 are connected with vertices 0, 1, 7, 9 since the sequence $\langle T, x = \text{false} \rangle$ tells them apart.



(a) Consistency graph example. Dotted edges represent edges of the scenario tree, solid edges belong to the consistency graph

(b) Colored scenario tree example

Fig. 2. Consistency graph and colored scenario tree examples

C. Constraint set construction and solving

Denote by F the set of all guard conditions found in scenarios. Let F_e be the set of all guard conditions marked with event $e \in \Sigma$. We will use the following integer variables.

1) $x_v \in [0, C-1]$, $v \in V$ – the color of scenario tree vertex v.
2) $y_{i,e,f} \in [0, C-1]$, $i \in [0, C-1]$, $e \in \Sigma$, $f \in F_e$ – auxiliary variables for specifiyng consistency (determinism) constraints. These variables are auxiliary in the sense that the transitions of the sought EFSM are not determined by the values of these variables. Each such variable corresponds to the state to which the transition of the target EFSM from state i marked with event e and guard condition f leads to.

The following constraints are used.

1) $x_0 = 0$ – the root of the scenario tree has to correspond to the initial state of the EFSM. In this work the initial state, without loss of generality, is chosen to be state 0.
2) $x_v \neq x_u$ (for each inconsistent pair of scenario tree vertices u and v, i.e. connected with and edge in the consistency graph) – constraints guaranteeing the consistency of the target EFSM. Such constraints guarantee that there are no sequences starting from the same state that tell u and v apart. The number of such constraints is $O\left(n^2\right)$ in the worst case, where n is the number of scenario tree vertices.
3) $(x_v = i) \Rightarrow (x_u = y_{i,e,f})$ (for each color i and each scenario tree edge vu marked with input event e and guard condition f) – constraints specifying that the target EFSM is deterministic. That is, if vertex v has color i, then the color of vertex u is the value of the $y_{i,e,f}$ variable, which stores the state to which the transition from state i marked with input event e and guard condition f leads to. The number of such constraints is $C \cdot (n-1)$.

To find an assignment of variables that satisfies the constructed constraints we use the *Choco* [10] *Java* library. *Choco* is an exact CSP-solver in the sense that it, on one hand, always finds a satisfying assignment if it exists, and on the other hand, is able to determine that such an assignment does not exist.

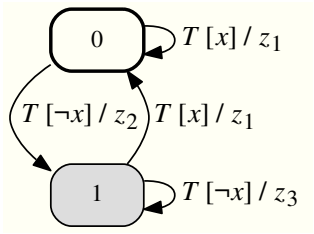


Fig. 3. EFSM constructed by merging vertices of the colored scenario tree from Fig. 2b

D. Constructing an EFSM from a satisfying assignment

If the CSP-solver returned a message that a satisfying assignment does not exist, then it is impossible to construct an EFSM with C states from the set of scenarios. Otherwise, given the values of $x_v, v \in V$, the color of each scenario tree vertex is determined. An example of such a coloring is given in Fig. 2b. All vertices of the same color are then merged into one state of the EFSM; the initial state is the one corresponding to the color of the scenario tree root. For example, merging appropriate vertices of the tree from Fig. 2b results in the EFSM shown in Fig. 3.

V. Combined CSP-MuACO algorithm

In this paper we propose to combine the CSP and MuACO algorithms to make the process of specification-based EFSM inference faster. First, the CSP algorithm is used to infer an EFSM consistent with scenarios only. LTL formulae are not taken into account at this point.

Note that the solution found by the CSP algorithm might by chance satisfy all LTL formulae, though the CSP algorithm is in no way constrained to do so. Also, the CSP algorithm typically finds the solution significantly faster than MuACO*sm* (if used for scenario-based EFSM inference), however, in some cases, it may take a very long time. Therefore, we set a time limit on the CSP algorithm runtime equal to $10 \times C$ seconds. In 6.8 % of all experimental runs the CSP-solver had been cut off after the time limit.

Finally, if the CSP algorithm found a solution that does not satisfy all LTL formulae or if it did not find any solution, the MuACO*sm* algorithm is used. It takes into account both test scenarios and LTL formulae.

TABLE I
WILCOXON TEST p-VALUES

	s_{len} multiplying factor		
C	50	100	200
5	0.543	0.078	0.043
6	0.075	0.071	0.059
7	0.0008	0.005	0.0005
8	0.337	0.001	0.221
9	0.0001	0.001	0.042
10	0.073	0.270	0.112

In the MuACO*sm* we use the fitness function proposed in [3]. This fitness function checks how well an EFSM complies with a set of test scenarios and a set of LTL formulae. A measure based on Levenshtein's string edit distance [11] is used to check the compliance of an EFSM with test scenarios. The Levenshtein's edit distance between two strings is the minimal number of insertions, deletions and substitutions that must be applied to one string in order to transform it to the other one. A special model checker developed by the authors of [3] is used to check LTL properties. For a full description of the fitness function the reader is asked to refer to [3].

Two mutation operators are used in MuACO*sm*.

1) **Change transition destination state**. This operator randomly selects a transition in the EFSM and changes the state it leads to. The new state is selected uniformly and randomly from the set of all states, excluding the current destination state of the transition.
2) **Delete or add transitions**. The existence of a particular transition in a state can make the EFSM incompliant with LTL formulae. This is why it is necessary to be able to delete and add transitions. The mutation operator scans EFSM states and with a certain probability changes the set of transitions in the selected state.
   A transition is either deleted from or added to the selected state. A transition is only added if there is no transition for some event in this state. If it is so, then the new transition's destination state label is selected uniformly and randomly from the set of all states. In case of deleting a transition, a random transition in the current state is selected and deleted.

The solution found by the CSP algorithm is assumed as initial for MuACO*sm*. If the CSP algorithm did not find a solution in the set time limit, then the initial solution for MuACO*sm* is generated randomly. MuACO*sm* was also modified to use the solution built by the CSP algorithm as the initial in the event of algorithm stagnation and restart.

VI. EXPERIMENTAL EVALUATION

A. Data preparation

We varied the number of EFSM states C from 5 to 10. The total scenarios length s_{len} was varied from $50 \times C$ to $200 \times C$ for each value of C.

For each value of C, 50 problem instances were generated. Each time we generated an EFSM with C states and then generated two random LTL formulae that the EFSM is compliant with. Then, for each value of $s_{\text{len}} \in [50, 100, 200] \times C$ a set of scenarios with a corresponding total length was generated from the EFSM using the scenario generator from [6]. Each scenario is a random path in the EFSM with length selected uniformly and randomly from $[C, 3 \cdot C]$.

B. Experiments

MuACO*sm* has a number of parameters that influence its efficiency. In order for the experiments to be meaningful we first used the *irace* [12] package to select good values for these parameters. The CSP algorithm itself, on the other hand, has no such parameters. However, CSP-solvers have tunable parameters, such as different search strategies. These were not investigated in this paper.

In our experiments we compared the combined CSP+MuACO*sm* algorithm with plain MuACO*sm*. The same parameter values were used for MuACO*sm* in both cases. Performing a separate tuning for CSP+MuACO*sm* could potentially result in its better performance, however, this was not investigated in this paper.

Both algorithms were implemented in *Java*, a machine with an AMD Phenom(tm) II x4 955 3.2 GHz processor was used. The CSP algorithm was allotted 4 Gb of RAM, MuACO*sm* was given 2 Gb for all cases except $C = 10$, where it was also allotted 4 Gb.

We measured the CPU time it takes an algorithm to find a perfectly fit EFSM. A time limit of 24 hours was set for each experiment. Plots of median execution time for each value of C and scenarios length are presented in Fig. 4.

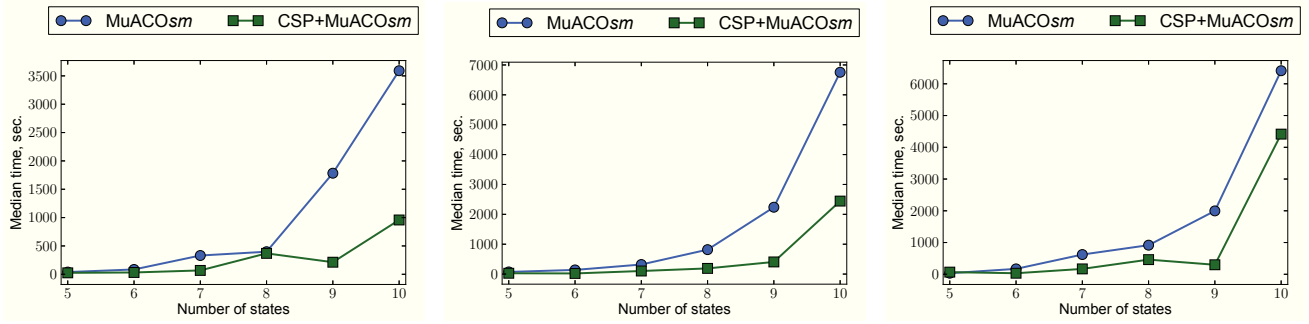
Note that only *hard* runs were taken into account. A run is considered to be hard if the EFSM found by the CSP algorithm is not compliant with all LTL formulae. About 70 % of all runs were hard. In addition, for $C \in [6, 10]$ about 2-4 experiments in each set did not finish in 24 hours. These runs were discarded from the final statistics.

As can be seen from Fig. 4, CSP+MuACO*sm* finds the solution significantly faster than MuACO*sm* in almost all cases. Two exceptions are cases $C = 8, s_{\text{len}} = 50 \times C$ and $C = 5, s_{\text{len}} = 200 \times C$, however, the performance difference here is not statistically significant. CSP+MuACO*sm* is by median runtime 1.5–8.3 times faster than MuACO*sm*. On average across all studied experiments, CSP+MuACO*sm* is 3.8 times faster than MuACO*sm*.

To measure the statistical significance of the difference between algorithm runtimes we used the unpaired version of the Wilcoxon test [13]. Calculated p-values are presented in Table I.

VII. CONCLUSION AND FUTURE WORK

In this paper a new exact algorithm for inferring EFSMs from test scenarios based on CSP-solvers has been proposed. It was demonstrated that combining exact and metaheuristic algorithms substantially improves the efficiency of specification-based EFSM inference.



(a) $s_{\text{len}} = 50 \times C$ (b) $s_{\text{len}} = 100 \times C$ (c) $s_{\text{len}} = 200 \times C$

Fig. 4. Experimental results: median execution time of MuACO*sm* and CSP+MuACO*sm*

There are two straightforward ways to improve achieved results. First of all, a more sophisticated tuning procedure can be used. In this paper tuning was performed on instances that are much simpler than the ones used in the actual experiments. The approach from [14] can be used to deal with this issue. Second, parameters of the CSP algorithm can also be tuned.

In the future we plan to extend the approach proposed here by using bounded model checking [15]. Preliminary research suggests that the problem of learning EFSMs from test scenarios and LTL formulae (in the bounded model checking sense) can be translated to Quantified SAT. After using a Quantified Boolean formula solver to infer an EFSM, we would then apply MuACO*sm* with full model checking to infer the final EFSM.

Acknowledgments

This work was financially supported by the Government of Russian Federation, Grant 074-U01, and also partially supported by RFBR, research project No. 14-07-31337 mol_a.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT press, 1999.

[2] A. A. Shalyto, "Software automation design: Algoritmization and programming of problems of logical control," *Journal of Computer and Systems Sciences International*, no. 6, pp. 899–916, 2000.

[3] F. Tsarev and K. Egorov, "Finite state machine induction using genetic algorithm based on testing and model checking," in *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation companion*, ser. GECCO companion'11. New York, NY, USA: ACM, 2011, pp. 759–762. [Online]. Available: http://doi.acm.org/10.1145/2001858.2002085

[4] D. Chivilikhin and V. Ulyantsev, "Inferring automata-based programs from specification with mutation-based ant colony optimization," in *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, ser. GECCO Comp '14. New York, NY, USA: ACM, 2014, pp. 67–68. [Online]. Available: http://doi.acm.org/10.1145/2598394.2598446

[5] ——, "MuACO*sm*: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO '13. New York, NY, USA: ACM, 2013, pp. 511–518. [Online]. Available: http://doi.acm.org/10.1145/2463372.2463440

[6] V. Ulyantsev and F. Tsarev, "Extended finite-state machine induction using sat-solver," in *Proceedings of the 10th International Conference on Machine Learning and Applications*, vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 346–349. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICMLA.2011.166

[7] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 248–257. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.35

[8] M. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-012-9222-z

[9] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man and Cybernetics.*, vol. 26, no. 1, pp. 29–41, 1996, part B.

[10] "Choco, Java library for constraint satisfaction problems (CSP) and constraint programming (CP)." [Online]. Available: http://www.emn.fr/z-info/choco-solver

[11] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[12] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The `irace` package, iterated race for automatic algorithm configuration," IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004, 2011. [Online]. Available: http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf

[13] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, Dec. 1945. [Online]. Available: http://dx.doi.org/10.2307/3001968

[14] J. Styles, H. Hoos, and M. Mller, "Automatically configuring algorithms for scaling performance," in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science, Y. Hamadi and M. Schoenauer, Eds. Springer Berlin Heidelberg, 2012, pp. 205–219. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34413-8_15

[15] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.