Extended Finite-State Machine Induction using SAT-Solver

Vladimir I. Ulyantsev * **Fedor N. Tsarev** **

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

Abstract: In the paper we describe the extended finite-state machine (EFSM) induction method which uses SAT-solver. Input data for the induction algorithm is a set of test scenarios. The algorithm consists of several steps: scenarios tree construction, compatibility graph construction, Boolean formula construction, SAT-solver invocation and finite-state machine construction from the satisfying assignment. These extended finite-state machines can be used in automata-based programming, where programs are designed using automated controlled objects. Each controlled object contains a finite-state machine and a controlled object. The described method has been tested on randomly generated scenario sets containing 250 to 1500 elements and on the alarm clock controlling EFSM induction problem where it has greatly outperformed genetic algorithm.

Keywords: Extended finite-state machine induction, SAT problem, SAT-solver, automata-based programming, testing

1. INTRODUCTION

Extended finite-state machines (EFSM) are widely used in computer science and reactive systems modeling. One of the EFSM application areas is automata-based programming (Polikarpova and Shalyto (2009), Shalyto (2001), Gurov et al. (2007)) where EFSMs are used as a software systems core component.

EFSM induction methods usage greatly increases automation level in automata-based program development. In previous works genetic algorithms and genetic programming are used for EFSM induction (Tsarev (2010), Tsarev and Egorov (2011)). These algorithms have a major drawback because of ability to process only relatively small test sets and produce only relatively small finite-state machines.

In this paper we present the EFSM induction algorithm based on translation to Boolean formula satisfiability problem (SAT) which can handle larger test sets and EFSMs.

The paper is structured as follows. Section 2 gives a short description of automata-based programming. Section 3 gives an overview of existing finite-state machines induction methods. Section 4 gives definitions of test scenarios and describes input data for the algorithm. Section 5 describes the algorithm. Section 6 gives experimental results. The paper finishes with the conclusion and description of future work.

2. AUTOMATA-BASED PROGRAMMING

Automata-based programming is the programming paradigm which proposes to design and implement software systems as systems of interacting automated controlled objects. Each automated controlled object consists of a finite-state machine and a controlled object.

A finite-state machine (FSM) has a set of states, a transition function and an action function. A controlled object has commands and requests (implemented by its methods) and a set of computational states.

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

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

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

3. FINITE-STATE MACHINE INDUCTION

Finite-state machine induction with genetic algorithms has been studied by several researchers.

In Spears and Gordon (2000) genetic algorithm is used to learn finite-state machines for "Competition for Resources" game. In this game two agents compete for resources (represented by cells of a field) on a toroidal field. One of the agents has a stochastic strategy and the other one is controlled by a finite-state machine. Finite-state machines in the genetic algorithm were represented using transition tables of size $80n$ (n is the number of states). Uniform mutation and crossover are used.

Spears and Gordon test the algorithm with $n = 1..10$. Experiments show that finite-state machines with two or more states perform better in this problem. Best finite-state machines with 3 to 10 states perform equally well – they win about 90% of games. Spears and Gordon also conduct experiments where state addition and deletion are allowed. These experiments show that these operations are too "destructive" to be used in evolutionary methods for finite-state machine induction.

Spears and Gordon also use dynamic verification to eliminate cyclic behavior of agent controlled by a finite-state machine. It allows creating a finite-state machine which wins 96% of games.

Finite-state machines for regular language recognition have been also studied. In Lucas and Reynolds (2003) two approaches to finite-state machines induction from examples are compared. One of them is Evidence-Driven State Merging (EDSM) and another one is based on evolutionary algorithms. A Finite-state machine is represented using the transition table. An initial state always has the number "0". Lucas and Reynolds take into consideration automata over binary alphabet only, so the total number of finite-state machines with n states is n^{2n}. The search space size is $2^n \cdot n^{2n}$ because each state can be either accepting or not.

To reduce the search space size Lucas and Reynolds propose "smart state labeling" algorithm to determine which states are accepting and which are not. This algorithm reduces the search space size to n^{2n}.

Lucas and Reynolds use a (1+1) evolutionary strategy. Its comparison with EDSM shows that the evolutionary method outperforms EDSM when finite-state machines are relatively small.

In Lucas and Reynolds (2005) authors develop their approach further. They propose two new mutation methods which they call *sampled* and *quick-sampled* mutation. They compare four algorithms — plain (which does not use state labeling), smart state labeling, sampled and quick-sampled.

Experiments show that sampled algorithm has the best performance, second place goes to the algorithm which uses mutation described in Lucas and Reynolds (2003), third place goes to quick-sampled. The algorithm which does not use smart state labeling demonstrates the worst performance.

In Heule and Verwer (2010) "translation to SAT" method for deterministic finite automaton (DFA) induction was applied. Their method includes several stages:

- augmented prefix tree construction;
- consistency graph construction;
- Boolean formula construction;
- SAT-solver invocation;
- DFA construction from Boolean variables values.

Their experiments show that "translation to SAT" method outperforms EDSM on Abbadingo One Lang et al. (1998) competition test set.

The finite-state transducers induction which is similar to the problem investigated in this paper is less studied.

In Lucas (2003) (1+1) evolutionary strategy is used to the finite-state transducers induction from the set of tests. In the algorithm the finite-state transducer is encoded using two tables: transition table and output table. Lucas and Reynolds use three types of fitness function based on:

- strict comparison of strings;
- Hamming distance (Hamming (1950));
- edit distance (Levenshtein (1966)).

Experiments show that the edit distance function has the best performance, second place goes to Hamming distance function and strict comparison is the worst.

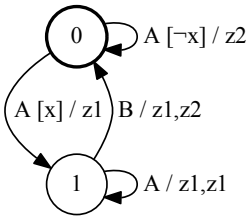

Fig. 1. Example of the extended finite-state machine

4. DEFINITIONS AND PROBLEM STATEMENT

In EFMSs considered in this paper each transition is labeled with an event, an output actions sequence and a guard condition which is a Boolean formula depending on input variables. States of the finite-state machine are not divided into accepting and non-accepting. EFSM example is shown on the Fig. 1. In this paper on this figure and all similar ones transition labels have the following format: "event [guard condition] /output actions sequence". If the guard condition is not specified the transition is performed unconditionally.

For the EFSM shown on figure the set of events is $\{A, B\}$, guard conditions depend only on one input variable x, set of output actions is $\{z1, z2\}$. In this EFSM and in all EFSMs considered in this paper initial state has the number 0.

Input data for EFSM induction is the set of test scenarios. A test scenario is a sequence of triples $T_1 \dots T_n$, $T_i = \langle e_i, f_i, A_i \rangle$, where e_i is an event, f_i is a Boolean formula of input variables, defining the guard condition, A_i is the sequence of output actions. Each of these triples T_i is called a scenario element.

We will say that a finite-state machine complies with the scenario element T_i in state s if there is a transition from state s labeled with event e_i, output actions sequence A_i and guard condition equal to f_i as Boolean formula. An extended finite-state machine complies with the scenario $T_1 \dots T_n$, if it complies with each of the scenario elements in states of path formed by transitions used while processing this scenario.

For example, the finite-state machine shown on Fig. 1, complies with the scenario $\langle A, \neg x, (z2)\rangle\langle A, x, (z1)\rangle$, and does not comply with following scenarios:

- $\langle B, \texttt{true}, (z2)\rangle$, because from the starting state no transition labeled with event B can be found;
- $\langle A, x, (z1)\rangle\langle A, x, (z1, z1)\rangle$, because from the state 1 no transition labeled with event A and guard condition equal to "x" as Boolean formula can be found;
- $\langle A, x, (z2)\rangle$, because the output action $z1$ (not $z2$) will be executed on the transition from a starting state.

In this paper the following problem is considered: you are given an integer number C and a set of scenarios Sc. The goal is to construct the finite-state machine with C state complying with all the scenarios from Sc. Note that the problem of induction of the FSM with minimal number of states can be reduced to the considered problem by applying binary search.

5. ALGORITHM DESCRIPTION

A EFSM induction algorithm proposed in this paper is based on ideas from Heule and Verwer (2010). First of all, all given scenarios are used to construct the scenarios tree. To construct the finite-state machine vertices of this tree are to be colored by the given number of colors (equal to the number of states). Vertices of the same color will be merged into a single state of the finite-state machine. Outgoing transitions for all states will be formed as a union of outgoing edges for vertices of the corresponding color. So, the EFSM induction algorithm has five stages:

(1) Scenarios tree construction.
(2) Consistency graph construction.
(3) Boolean CNF-formula construction. This formula represents requirements for the coloring of the graph and consistency requirement for finite-state machine transitions.
(4) SAT-solver invocation.
(5) Finite-state machine construction from the satisfying assignment.

5.1 Scenarios tree construction

Scenarios tree is a tree, each edge of which is labeled with an event, a guard condition and a sequence of output actions. The scenarios tree construction algorithm is described below.

At the start of the algorithm the scenarios tree contains only one vertex — the root. Each scenario is processed separately in this algorithm.

Each of the scenarios is inserted element by element starting from the first one to the tree. During this process two variables will be stored: the number of the current vertex of tree v and the number i of first not yet processed element of scenario.

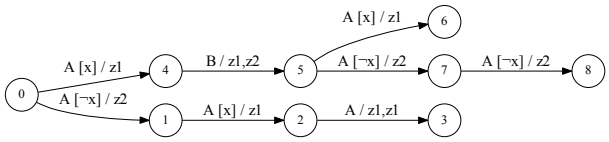

Fig. 2. Scenarios tree

At the start of the scenario insertion v is the root and $i = 1$. On each step the existence of the outgoing edge E from the vertex v, labeled with event e_i and a guard condition equal to f_i as Boolean function is checked. If such an edge does not exist then a new vertex u and a new edge from v to u are created. This new edge is labeled by triple $\langle e_i, f_i, A_i\rangle$. After that u becomes the current vertex and i is increased by one.

If such an edge exists then sequence of output actions A_i is compared with the sequence A', by which edge E is labeled. If $A_i = A'$, then the vertex to which edge E goes becomes the current one and i is increased by one.

If these sequences are not equal then the scenarios set Sc is contradictory. In this case the algorithm stops and the corresponding message is shown to the user.

Guard conditions check is performed after insertion of all scenarios to the tree. All pairs of outgoing edges are checked for each vertex of the tree. If there exists a pair of edges labeled with the same event such that their guard conditions are not equal as Boolean functions but have a common satisfying assignment, then the scenarios set defines the non-deterministic behavior. In this case the algorithm stops and the corresponding message is shown to user.

Fig. 2 shows the scenarios tree constructed from the following set of scenarios:

- $\langle A, \neg x, (z2)\rangle\langle A, x, (z1)\rangle\langle A, \texttt{true}, (z1, z1)\rangle$;
- $\langle A, x, (z1)\rangle\langle B, \texttt{true}, (z1, z2)\rangle\langle A, x, (z1)\rangle$;
- $\langle A, x, (z1)\rangle\langle B, \texttt{true}, (z1, z2)\rangle\langle A, \neg x, (z2)\rangle$ $\langle A, \neg x, (z2)\rangle$.

The EFSM shown of Fig. 1 complies with this scenarios set.

5.2 Consistency Graph Construction

Consistency graph vertices set is the same as scenarios tree vertices set; therefore we will not distinguish between graph and tree vertices. Graph edges are constructed in the following way. Two vertices u and v are connected by an edge (such vertices are called inconsistent), if there exists a sequence of events and variables values sets pairs $\langle e_1, \mathsf{values}_1\rangle \dots \langle e_k, \mathsf{values}_k\rangle$, which tells them apart. We will say, that this sequence tells vertices u and v apart if each of the following conditions holds:

- in the tree there is a path P_u from vertex u, whose edges are labeled with events $e_1 \dots e_k$ and such guard conditions $f_1 \dots f_k$, that values_1 is a satisfying assign-

ment for f_1, values_2 is a satisfying assignment for f_2, ..., values_k is a satisfying assignment for f_k;
- the similar path P_v starts from vertex v;
- for last edges of paths P_u and P_v at least one of the following conditions holds:
  - · labels of this edges differ in output actions;
  - · guard conditions of this edges have a common satisfying assignment, but are not equal as Boolean functions.

The consistency graph construction algorithm is based on dynamic programming. For each scenarios tree vertex v we compute the set $S(v)$ of vertices inconsistent with it. These sets are computed starting from tree leaves. For each leaf u the set $S(u)$ is empty by definition (because there are no paths starting from a leaf).

Suppose that this set is already computed for all children of some vertex v. Set $S(v)$ can be computed the following way. We check all vertices of tree — vertex u should be included into the set $S(v)$ if there exists a pair of edges ux (labeled with event e, formula f_1 and output action sequence A_1) and vy (labeled with the same event e, formula f_2 and output action sequence A_2) such that at least one of the following conditions holds:

- formulae f_1 and f_2 have a common satisfying assignment, but are not equal as Boolean functions. It means that $\langle e, \mathsf{values}\rangle$ is a sequence which tells apart u and v (here by values the satisfying assignment of f_1 is denoted);
- formulae f_1 and f_2 are equal as Boolean functions, but sequences A_1 and A_2 are not equal. It means that $\langle e, \mathsf{values}\rangle$ is a sequence which tells apart u and v;
- formulae f_1 and f_2 are equal as Boolean functions and vertex x is included into $S(y)$, which is already computed. It means that there exists a sequence $\langle e_1, \mathsf{values}_1\rangle \dots \langle e_k, \mathsf{values}_k\rangle$, telling apart x and y, and vertices v and u are told apart by the sequence $\langle e, \mathsf{values}\rangle\langle e_1, \mathsf{values}_1\rangle \dots \langle e_k, \mathsf{values}_k\rangle$.

The running time of this stage of the EFSM induction algorithm is $O(n^2)$ (n denotes the number of vertices in scenarios tree is denoted), because each pair of scenarios tree edges will be considered at most once.

To achieve this running time we make a certain precomputation. For each pair of guard conditions f and g occurring in the scenarios we:

- find if f is equal to g as Boolean function;
- find if f and g have a common satisfying assignment.

In the worst case the running time of the precomputation step is $O(2^{2m-1} \cdot n^2)$, where m is the maximal number of input variables used in one formula. In practice m do not exceed 5.

Fig. 3 shows the consistency graph for the scenarios tree from Fig. 2.

5.3 Boolean CNF-formula Construction

The CNF-formula construction algorithm is based on ideas from Heule and Verwer (2010). The CNF-formula contains following variables:

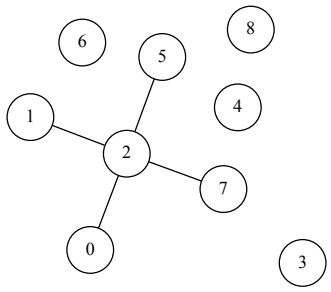

Fig. 3. Consistency graph

- $x_{v,i}$ (for each vertex v of scenarios tree and color i which is a number from 1 to C) — if it is true that vertex v has color i;
- $y_{a,b,e,f}$ (for each pair of resulting EFSM states (a, b), each event e and each formula f occurring in scenarios) — is it true that in resulting EFSM a transition from state a to state b labeled with event e and formula f exists.

CNF-formula contains following clauses:

- $(x_{v,1} \vee \dots \vee x_{v,C})$ (for each vertex v) — vertex v should be colored with at least one color;
- $(\neg x_{v,i} \vee \neg x_{v,j})$ (for each vertex v, each pair of colors $i < j$) — vertex v can not be colored with colors i and j simultaneously;
- $(\neg x_{v,i} \vee \neg x_{u,i})$ (for each pair of inconsistent vertices u and v and each color i) — no pair of inconsistent vertices can be colored with same color;
- $(\neg y_{a,b,e,f} \vee \neg y_{a,d,e,f})$ (for each triple a, b and d $(b < d)$ of colors, each event e, each formula f occurring in scenarios) — there is at most one outgoing transition labeled by event e and formula f from state of EFSM labelled by each color;
- $(y_{a,b,e,f} \vee \neg x_{v,a} \vee \neg x_{u,b})$ (for each edge vu of scenarios tree, each event e, each formula f and each pair of colors (a, b)) — if following conditions hold:
  - · this edge is labeled with event e and formula f;
  - · vertex v is colored with color a;
  - · vertex u is colored with color b;

  then a transition from state a to state b, labeled with event e and formula f must exist in resulting finite-state machine;
- $(\neg y_{a,b,e,f} \vee \neg x_{v,a} \vee x_{u,b})$ (for each edge vu of scenarios tree, each event e, each formula f and each pair of colors (a, b)) — if following conditions hold:
  - · edge from vertex v to vertex u is labeled with event e and formula f;
  - · vertex v is colored with color a;
  - · a transition from state a to state b and this transition is labeled with event e and formula f must exist in resulting finite-state machine;

  then vertex u has the color b.

6. USAGE OF SAT-SOLVER TO FIND SATISFYING ASSIGNMENT FOR THE CNF-BOOLEAN FORMULA

To find the satisfying assignment for the constructed CNF-formula we use *cryptominisat* SAT-solver (Soos (2010)),

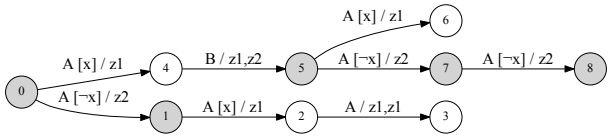

Fig. 4. Scenarios tree coloring

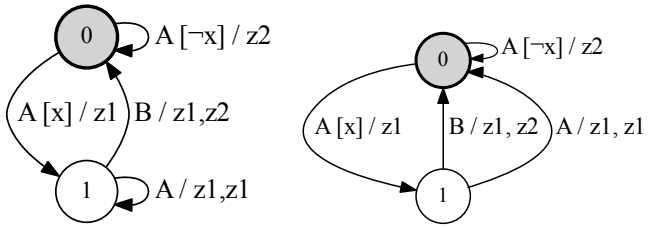

Fig. 5. Extended finite-state machines obtained by merging vertices of trees

the winner of SAT RACE 2010 (`http://baldur.iti.uka.de/sat-race-2010`). DIMACS format (`http://www.satlib.org/ubcsat/satformat.pdf`) is used to represent the formula.

If a SAT-solver does not find the satisfying assignment then the EFSM with C states complying with the given set of scenarios Sc does not exist. In other case, we determine scenarios tree vertices colors from $x_{v,i}$ values. Fig. 4 shows the scenarios tree coloring from Fig. 2. Note that value C is fixed and is not an objective for optimization.

After that all vertices of the same color are merged into a single state of the finite-state machine. Starting state is the state corresponding to the color of tree root.

Note that since coloring of the scenarios tree is not necessarily unique there can be several EFSMs with C states complying with the given set of scenarios. Finite-state machines shown on the left of Fig. 5 is obtained by merging vertices of the tree shown on Fig. 4. Note, that this finite-state machine is isomorphic to the machine shown on Fig. 1.

If we change the color of vertex 3 of this tree we obtain the EFSM shown on the right of Fig. 5. This EFSM is not isomorphic to the machine shown on Fig. 1.

7. EXPERIMENTS

First experiment has been performed on the EFSM for alarm clock controlling induction problem (Tsarev (2010), Tsarev and Egorov (2011)). This clock has three buttons (marked with letters "H", "M", "A"), a timer and three modes of operation: "alarm is off", "alarm is on", "setting alarm time". Button A is used for switching between these modes, buttons H and M — to adjust the time.

Alarm clock has four events:

- H – button is "H" pressed;
- M – button is "M" pressed;
- A – button is "A" pressed;
- T – occurs on each timer tick.

It has two input variables:

- x_1 – if it is true that current time is equal to alarm time;
- x_2 – if it is true that current time is one minute more than the alarm time.

It also has seven output actions:

- z_1 – increase current time hours;
- z_2 – increase current time minutes;
- z_3 – increase alarm time hours;
- z_4 – increase alarm time minutes;
- z_5 – increase current time by one minute;
- z_6 – turn on the buzzer;
- z_7 – turn off the buzzer.

The control system for this alarm clock can be described by a manually designed finite-state machine.

The test set for this problem contains 38 scenarios (total size of scenarios is 242 scenario elements) describing the finite-state machine behavior in different modes of operation. On this problem algorithm described in this paper inducts the correct EFSM in less than one second, while the genetic algorithm from Tsarev (2010) and Tsarev and Egorov (2011) needs about five minutes on the same computer with Intel Core 2 Quad Q9400 processor and 4 GB of RAM.

Second experiment measures the algorithm performance on larger sets of scenarios. This experiment contains six stages:

- random EFSM with n states generation. This EFSM is denoted by A;
- test scenarios generation — each scenario is a random path in the EFSM — each scenario is a random path in the EFSM with length from n to $3n$. Total size of scenarios is denoted by l;
- EFSM with n states induction from the generated set of scenarios with the described algorithm. Running time of the whole induction process (including solving SAT with *cryptominisat*) is recorded. The resulting EFSM is denoted by A';
- A and A' are checked for isomorphism;
- $1000n$ random scenarios of length $4n$ are generated from A. After that A' is checked against each of these scenarios. The part of scenarios A' complies with is recorded. This stage is called "forward check";
- $1000n$ random scenarios of length $4n$ are generated from A'. After that A is checked against each of these scenarios. The part of scenarios A complies with is recorded. This stage is called "backward check".

EFSMs are generated with the following parameters:

- number of events is equal to two;
- number of output actions is equal to two, minimal size of output actions sequence is equal to one, maximal size of output actions sequence is equal to three. So the number of different output actions sequences is $2^1 + 2^2 + 2^3 = 14$;
- guard conditions depend on at most two input variables and may contain only AND operator. So the number of different guard conditions is equal to $1 + 2 \cdot 2 + 2^2 = 9$;
- each EFSM contains $4n$ transitions (half of possible transitions).

For each combination of $n = 5, 10, 15, 20$ and $l = 250, 500, 750, 1000, 1250, 1500$ thirty experiments were conducted. Source code used to run experiments is available online: `http://rain.ifmo.ru/~ulyantsev/EFSM.zip`.

These experiments have been conducted using computer with Intel Core i5-2520M (2.5 GHz) processor. On generated scenario sets the algorithm used up to 3.5 GB of RAM, and CNF-formula contained up to 35000 variables and 9000000 clauses. Table 1 contains the summary of experiment results.

This table contains following infromation:

- runtime — mean EFSM construction time (in seconds);
- isom — part of experiments in which A' is isomorphic to A (in percent);
- forw — results of forward check (in percent);
- backw — result of backward check (in percent).

For "runtime", "forw" and "backw" standard deviation is also given.

Table 1. Experiment results

n	l	runtime	isom	forw	backw
5	250	0.4 (0.1)	33.3	94.3 (13.2)	35.0 (46.1)
5	500	0.9 (0.1)	36.7	97.8 (5.2)	37.6 (47.5)
5	750	1.5 (0.2)	36.7	98.9 (4.2)	38.6 (47.0)
5	1000	1.8 (0.1)	46.7	99.6 (1.4)	52.1 (47.7)
5	1250	2.0 (0.2)	30.0	100.0 (0.0)	30.7 (45.4)
5	1500	2.5 (0.3)	43.3	99.4 (2.3)	46.0 (48.2)
10	250	1.6 (0.1)	0.0	18.3 (24.0)	0.1 (0.2)
10	500	2.3 (0.5)	0.0	63.7 (22.1)	3.0 (13.1)
10	750	3.3 (0.4)	6.7	86.4 (14.0)	7.5 (25.1)
10	1000	4.9 (0.5)	10.0	93.8 (11.8)	10.4 (29.9)
10	1250	6.5 (0.6)	6.7	95.5 (7.6)	8.0 (24.9)
10	1500	9.1 (1.2)	3.3	97.9 (4.5)	3.9 (17.9)
15	500	7.4 (8.5)	0.0	18.1 (19.0)	1.4 (6.6)
15	750	10.0 (9.3)	0.0	51.7 (22.4)	3.3 (17.9)
15	1000	12.2 (8.1)	0.0	66.7 (19.6)	0.1 (0.3)
15	1250	15.2 (3.9)	0.0	79.0 (17.7)	0.1 (0.4)
15	1500	20.2 (2.6)	0.0	87.8 (13.1)	0.1 (0.4)
20	750	54.2 (82.7)	0.0	21.9 (23.4)	0.1 (0.5)
20	1000	67.7 (136.5)	0.0	40.0 (21.1)	0.0 (0.0)
20	1250	63.4 (144.8)	0.0	56.3 (21.7)	0.0 (0.0)
20	1500	75.2 (63.6)	0.0	62.3 (17.5)	0.0 (0.0)

Note that table does not contain lines for ($n = 15$, $l = 250$), ($n = 20$, $l = 250$) and ($n = 20$, $l = 500$) because these experiments did not finish in 18 hours.

Experiment results show that we cannot guarantee that A' is isomorphic to A, but if the size of scenarios set is big enough A' complies with big part of scenarios generated from A (forward check). At the same time A' usually complies with a lot of additional scenarios not compliant with A (backward check).

8. CONCLUSION

The EFSM induction algorithm based on translation to SAT is described in the paper. When compared with genetic algorithm this algorithm performs faster up to an order of magnitude.

Future work includes constraint satisfiability problem (CSP) solver application instead of SAT-solver and usage of verification methods in the EFSM induction process.

REFERENCES

Gurov, V., Mazin, M., Narvsky, A., and Shalyto, A. (2007). Tools for support of automata-based programming. *Programming and Computer Software*, 33(6), 343–355.

Hamming, R.W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2), 147–160.

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

Lang, K.J., Pearlmutter, B.A., and Price, R.A. (1998). Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *ICGI*, volume 1433 of *Lecture Notes in Computer Science*, 1–12. Springer.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8), 707–710.

Lucas, S. (2003). Evolving finite state transducers: Some initial explorations. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa (eds.), *Genetic Programming*, volume 2610 of *Lecture Notes in Computer Science*, 241–257. Springer Berlin / Heidelberg.

Lucas, S. and Reynolds, J. (2003). Learning dfa: Evolution versus evidence driven state merging. *The 2003 Congress on Evolutionary Computation (CEC '03)*, 1, 351–358.

Lucas, S. and Reynolds, J. (2005). Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27, 1063–1074.

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

Shalyto, A. (2001). Logic control and reactive systems: Algorithmization and programming. *Automation and Remote Control*, 62(1), 1–29.

Soos, M. (2010). Cryptominisat 2.5.0. In *SAT Race competitive event booklet*.

Spears, W.M. and Gordon, D.F. (2000). Evolving finite-state machine strategies for protecting resources. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems 2000. ACM Special Interest Group on Artificial Intelligence*, 166–175. Springer-Verlag.

Tsarev, F. (2010). Method of finite state machine induction from tests with genetic programming. *Information and Control Systems (Informatsionno-upravljajuschie sistemy, in Russian)*, (5), 31–36.

Tsarev, F. and Egorov, K. (2011). 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*, GECCO '11, 759–762. ACM, New York, NY, USA.