2011 10th International Conference on Machine Learning and Applications

Extended Finite-State Machine Induction using SAT-Solver

Vladimir Ulyantsev
St. Petersburg National Research University of Information Technologies, Mechanics and Optics
Computer Technologies Department
197101, St. Petersburg, Russia
Kronverkskiy pr., 49
Email: ulyantsev@rain.ifmo.ru

Fedor Tsarev
St. Petersburg National Research University of Infromation Technologies, Mechanics and Optics
Computer Technologies Department
197101, St. Petersburg, Russia
Kronverkskiy pr., 49
Email: tsarev@rain.ifmo.ru

***Abstract*—In the paper we describe the extended finite-state machine (EFSM) induction method that 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 satisfying assignment. These extended finite-state machines can be used in automata-based programming, where programs are designed as automated controlled objects. Each automated controlled object contains a finite-state machine and a controlled object. The method described has been tested on randomly generated scenario sets of size from 250 to 2000 and on the alarm clock controlling EFSM induction problem where it has greatly outperformed genetic algorithm.**

I. Introduction

Extended finite-state machines (EFSM) are widely used in linguistics, computer science, philosophy, biology, mathematics, logic and reactive systems modeling. One of the EFSM application areas is automata-based programming [1], [2], [3] 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 [4]. 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.

II. Automata-based Programming

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

A finite-state machine 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 finite-state machine takes events and input variables as input. They can come from other parts of the system as well as from the controlled object. After receiving an event and an input variable the finite-state machine makes transition on which some output action is sent to 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 (and even infinite). 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 the controlled object, events and output actions are predefined and our task is to design the finite-state machine.

III. Finite-state Machines Induction

Finite-state machines induction with genetic algorithms has been studied by several researchers. In [5] Spears and Gordon use genetic algorithm 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

978-0-7695-4607-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ICMLA.2011.166

IEEE computer society

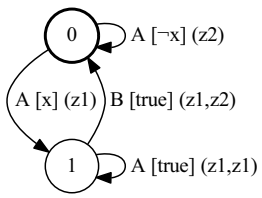

Fig. 1. Example of the extended finite-state machine

with 3 to 10 states perform equally well – they win about 90% of games.

Finite-state machines for regular languages recognition have also been studied. In [6] Heule and Verwer apply "translation to SAT" method for deterministic finite automaton (DFA) induction. Their experiments show that method outperforms EDSM on Abbadingo One [7] competition test set.

IV. 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. An example of an EFSM 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)". The guard condition "true" means that the transition is performed unconditionally.

For the EFSM shown on the Fig. 1 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 the state with number 0 is starting.

Input data for EFSM induction is a 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 a path formed by transitions used while processing this scenario.

In this paper the following problem is considered: you are given an integer number C and a set of scenarios Sc. You are to construct the finite-state machine with C states complying with all the scenarios from Sc.

V. Algorithm Description

EFSM induction algorithm is based on ideas from [6]. First of all, all given scenarios are used to construct a scenarios tree. To construct the finite-state machine vertices of this tree are to be painted by the given number of colors (equal to the number of states). Vertices of one color will be merged into one 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, EFSM induction algorithm has five stages:

1) Scenarios tree construction.
2) Consistency graph construction.
3) Boolean CNF-formula construction.
4) SAT-solver invocation.
5) EFSM construction from the satisfying assignment.

A. Scenarios tree construction

A 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 into 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 the scenario.

At the start of the scenario insertion v is the root and $i = 1$. On each step we check the existence of the outgoing edge E from vertex v, labeled with event e_i and a guard condition equal to f_i as Boolean function. 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 the 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 the 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 current 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 user.

Guard conditions check is performed after the insertion of all scenarios into 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 a scenarios tree constructed from three scenarios. The EFSM shown on Fig. 1 complies with these scenarios.

B. 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

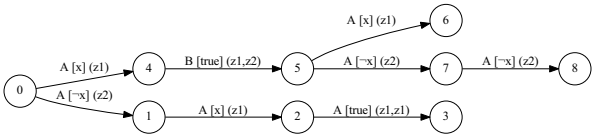

Fig. 2. Scenarios tree

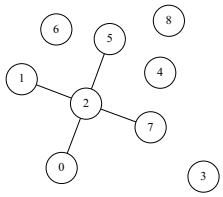

Fig. 3. Consistency graph

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, the edges of which are labeled with events $e_1 \dots e_k$ and such guard conditions $f_1 \dots f_k$, that values_1 is a satisfying assignment for f_1, values_2 is a satisfying assignment for f_2, …, values_k is a satisfying assignment for f_k;
- a similar path P_v starts from vertex v;
- for the last edges of paths P_u and P_v at least one of the following conditions holds:
  - labels of these edges differ in output actions;
  - guard conditions of these edges have a common satisfying assignment, but are not equal as Boolean functions.

The main idea of the consistency graph construction algorithm is 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 in the following way. We check all vertices of the 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 actions sequence A_1) and vy (labeled with the same event e, formula f_2 and output action sequence A_2) such that 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 $O(n^2)$ (by n the number of vertices in scenarios tree is denoted), because each pair of scenarios tree edges will be considered no more than 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 a 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.

C. Boolean CNF-formula Construction

The CNF-formula construction algorithm is based on ideas from [6]. The CNF-formula contains six types of clauses depending from the following variables:

- $x_{v,i}$ (for each vertex v of the 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.

D. 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 [8], the winner of SAT RACE 2010 (http://baldur.iti.uka.de/sat-race-2010). DIAMAX format (http://www.satlib.org/ubcsat/satformat.pdf) is used to represent the formula.

If 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 the other case, we determine scenarios tree vertices colors from $x_{v,i}$ values. Fig. 4 shows the scenarios tree coloring from Fig. 2.

After that all vertices of the same color are merged into one state of the finite-state machine. Starting state is the state corresponding to the color of tree root. For example, after merging vertices of the tree shown on Fig. 4 we obtain the

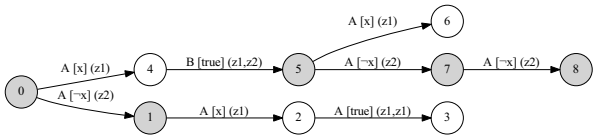

Fig. 4. Scenarios tree coloring

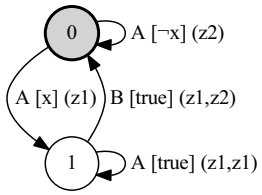

Fig. 5. Extended finite-state machine obtained by merging vertices of tree

finite-state machine shown on Fig. 5. Note, that this finite-state machine is isomorphic to the machine shown on Fig. 1.

VI. Experiments

First experiment has been performed on the EFSM for alarm clock controlling induction problem [4]. 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. The alarm clock has four events, two input variables and seven output actions.

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 the algorithm described in this paper inducts the correct EFSM in less than one second, while the genetic algorithm from [4] 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 three stages:

- generation of a random EFSM with n states;
- test scenarios generation — each scenario is a random path in the EFSM;
- induction of a EFSM with n states from the generated set of scenarios with the described algorithm. Running time of the whole induction process (including solving SAT with *cryptominisat*) is recorded.

Ten EFSMs with 5, 10, 15 and 20 states have been generated. From each EFSM eight scenario sets with size of 20, 40, 60, 80, 100, 120, 140, 160 scenarios and total size of 250, 500, 750, 1000, 1250, 1500, 1750, 2000 scenario elements were generated. Size of each scenario does not exceed 25.

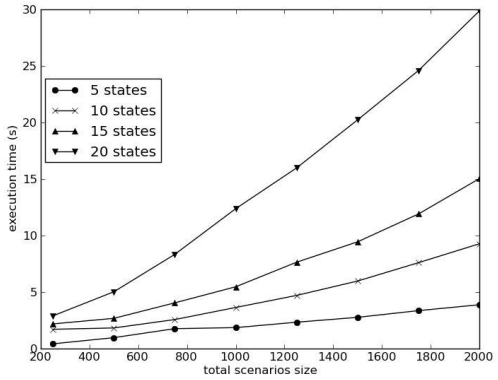

Fig. 6. Execution time for EFSMs with 5, 10, 15 and 20 states

This experiment has been also conducted using a computer with Intel Core 2 Quad Q9400 processor. On generated scenario sets the algorithm used up to 3.5 GB of RAM, and CNF-formula contained up to 25000 variables and 5000000 clauses. Fig. 6 shows dependency of execution time from the total scenarios size for EFSM with 5, 10, 15 and 20 states.

VII. 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.

This research is supported by the Ministry of Education and Science of Russian Federation under contract 16.740.11.0455.

References

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

[2] A. Shalyto, "Logic control and reactive systems: Algorithmization and programming," *Automation and Remote Control*, vol. 62, no. 1, pp. 1–29, 2001.

[3] V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto, "Tools for support of automata-based programming," *Programming and Computer Software*, vol. 33, no. 6, pp. 343–355, 2007.

[4] F. Tsarev, "Method of finite state machine induction from tests with genetic programming," *Information and Control Systems (Informatsionno-upravljajuschie sistemy, in Russian)*, no. 5, pp. 31–36, 2010.

[5] W. M. Spears and D. F. Gordon, "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*. Springer-Verlag, 2000, pp. 166–175.

[6] M. Heule and S. Verwer, "Exact dfa identification using sat solvers," in *Grammatical Inference: Theoretical Results and Applications 10th International Colloquium, ICGI 2010*, ser. Lecture Notes in Computer Science, J. M. Sempere and P. Garca, Eds., vol. 6339. Springer, 2010, pp. 66–79. [Online]. Available: http://www.st.ewi.tudelft.nl/~marijn/publications/DFA_ICGI.pdf

[7] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm," in *ICGI*, ser. Lecture Notes in Computer Science, vol. 1433. Springer, 1998, pp. 1–12.

[8] M. Soos, "Cryptominisat 2.5.0," in *SAT Race competitive event booklet*, July 2010.