Verification of automata-based programs

Evgeny Kurbatsky
St. Petersburg State University of Information Technologies,
Mechanics and Optics
Computer Technologies Department
Sablinskaya Street 14, St. Petersburg, Russia
kurbatsky@gmail.com

***Abstract*—this paper describes a verification method of automata based programs [1] based on symbolic model checking algorithms [2]. Author makes an attempt to develop verification method that can automate process of verification and can be useful for peoples unacquainted with model checking algorithms or tools.**

I. Introduction

MODEL checking [2] is a powerful technique for verifying reactive systems. Able to find subtle errors in real commercial designs, it is gaining wide industrial acceptance. Compared to other formal verification techniques (e.g. theorem proving) model checking is largely automatic. In model checking, the specification is expressed in temporal logic and the system is modeled as transition system with finite number of states.

Existing verification programs can be divided into three categories:

- The first category contains verifiers which takes a model as input. Popular verifiers SMV [3] and SPIN [4] can be mentioned as examples. In this kind of verifiers, model constructing, determining it's properties and understanding of counterexamples are human tasks. This approach has following disadvantages:
  - Doing all mentioned activities can take a lot of time and effort.
  - Mistakes have been done during model constructing decrease check effectiveness.
- The second category contains verifiers of Turing complete programming languages. Examples of such verifiers are Java Pathfinder [5], BLAST [6], Microsoft Static Driver Verifier [7]. This kind of verifiers automatically constructs abstract models from program. The main problem of this kind of verifiers is model adequacy. Model can lost some properties of program.
- The third category contains verifiers of programming languages which are not Turing complete. This type of systems has following advantages:
  - Program is not an abstract model and can be executed.
  - Program is adequate to it's model.

  The main disadvantage of this languages is small expressiveness.

This work describes applying third approach for verifying automata based programs. In work [8] was shoved how to verify simple automata based programs, that contains one finite state machine. Often real programs are described as systems of finite state machines. In this paper we consider algorithms for verifying such systems.

The research is supervised by Anatoly Shalyto, PhD, professor at the Computer Technologies Department, St. Petersburg State University of Information Technologies, Mechanics and Optics Computer Technologies Department Sablinskaya street 14, St. Petersburg, Russia shalyto@mail.ifmo.ru

II. Automata based programming

In context of automata based programming it is recommended to build programs like automated systems, witch consist of control system (system of cooperating finite state machines), controlled objects and feedback loops. Finite state machine transits between states using input actions (events and input variables) and form output actions which correspond to controlled objects' methods. Such a view on programming is natural while solving different controls problems including reactive systems. In automata based programs states are divided in two categories: logic states and calculation states [9]. Program have finite number of logic states and infinite number of calculation states.

III. Related works

A. Rebeca

Rebeca [10] (Reactive Objects Language) is an actor-based language with a formal foundation, designed in an effort to bridge the gap between formal verification approaches and real applications. It can be considered as a reference model for concurrent computation, based on an operational interpretation of the actor model. It is also a platform for developing object-based concurrent systems in practice.

B. Statemate model checker

The Statemate model checker is a tool for the System Development CASE Tool Statemate Magnum. It supports robustness checks and standard analysis to prevent the user from typical design errors. Tool generates a scenario where this kind of error occurs.

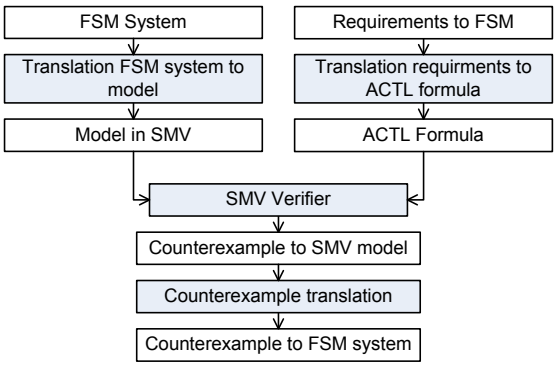

Fig. 1. Schema of the method

IV. Model

In automata based programming programm is described as finite state machine system. Finite state machine system used in this work is a set of a finite state machines (FSM) $A = \{A_1, A_2, ..., A_n\}$ and a controlled object O.

Controlled object O is pair $\{Z, X\}$:

- Z – set of actions $z_1, z_2, ..., z_{n'}$;
- X – set of input variables $x_1, x_2, ..., x_m$.

Each FSM A_i can be described as $\{\Sigma_i, \Gamma_i, S_i, s_{i0}, \delta_i, \omega_i\}$ where:

- Σ_i – finite set of input alphabet;
- Γ_i – finite set of output alphabet;
- S_i – finite set of states;
- s_{i0} – start state $s_{i0} \in S_i$;
- δ_i – transition function $\Sigma \times V_1 \times V_2 \times ... \times V_m \times S_1 \times S_2 \times ... \times S_n \rightarrow S_i$, where $V_1, V_2, ..., V_m$ - ranges of values of variables $x_1, x_2, ..., x_n$, $S_1, S_2, ..., S_n$ - ranges of states of automatons $A_1, A_2, ..., A_n$ accordingly;
- ω_i – output function $\Sigma \times V_1 \times v_2 \times ... \times V_m \times S_1 \times S_2 \times ... \times S_n \rightarrow \Gamma_i$. Each element of Γ_i is list of actions $\{a_1, a_2, ..., a_m\}$, where a_k is action one of two types:
  - $e_k \in \Sigma_j$ – sending event e_k to the FSM A_j.
  - $z_k \in Z$ – making action z_k.

V. Requirements

Following requirements of finite state machines can be checked:

- system always reaches state f;
- system never reaches state f;
- system always reaches state f after state g;
- system always reaches state f before state g.

Where f, g state formulas with this parts:

- `A in S` – automaton A in state S;
- `A.a` – automaton A doing action a;
- `A.e` – automaton A receives event e.

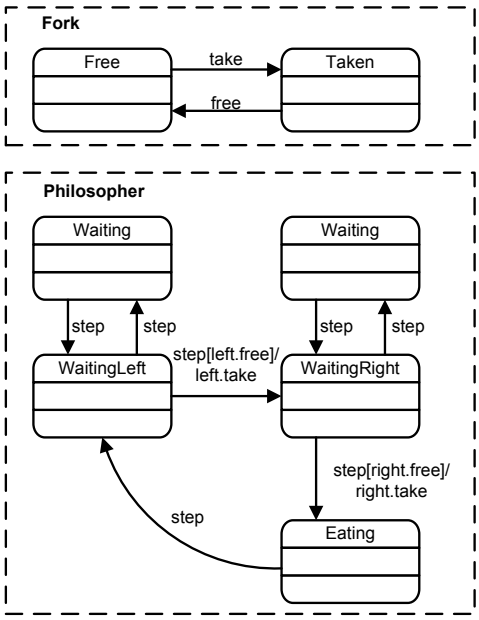

Fig. 2. Finite state machines representing philosopher and fork.

VI. Proposed method

In proposed approach verification is divides on this steps:

- Translate automata system to model.
- Translate requirement for finite state machine system to ACTL formula for transition system.
- Run model verification tool.
- Translate counterexample to model to counterexample to finite state machine system.

On figure 1 showed schema of finite state machine verification. Translation finite state machine system to model and counterexample translation are automated. Model verification provided by SMV [3] or NuSMV [11] verifier. This verifiers uses Symbolic Model Checking algorithm based on OBDDS.

VII. Experimental results

As result of work verification tool was developed. This tool provide verification of automata based programs. It takes transition diagrams describing program and requirements.

Tool were tested at the dining philosophers problem. The dining philosophers problem is summarized as five philosophers sitting at a table doing one of two things - eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each philosopher, and as such, each philosopher has one fork to his or her left and one fork to his or her right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his or her immediate left or right.

This problem was described as system of finite state machines showed at picture 2. Table I shows results of testing.

TABLE I
RESULTS OF TESTS

Automatons	Total states	Time
4	100	0.2s
6	1000	1.4s
8	10000	9s
10	100000	42s
12	1000000	4m
14	10000000	10m
16	100000000	27m
18	1000000000	76m

VIII. FUTURE WORK

In works [12], [13] was proposed usage μ-calculus for describing system requirements. It was shown how recursive finite state machines can be verified without stack modeling. This method can decrease number of model states. In future work it is planed to apply this method to automata based programming and upgrade tool to use μ-calculus algorithms for model checking.

IX. CONCLUSION

Results that were achieved show that applying model checking verification method to automata based programs can be perspective, because normally such programs has no more than billion of logical states and can be successfully checked.

REFERENCES

[1] A. A. Shalyto, *Switch-Technology Algorithmization and Programming of Logic Control*. Science (Nauka), 1998. [Online]. Available: http://is.ifmo.ru/books/switch/1/

[2] E. Clarke, O. Glumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] "Symbolic model verifier." [Online]. Available: http://www.cs.cmu.edu/~modelcheck/smv.html

[4] "Spin model checker." [Online]. Available: http://spinroot.com/spin/whatispin.html

[5] "Java pathfinder." [Online]. Available: http://javapathfinder.sourceforge.net/

[6] "Berkeley lazy abstraction software verification tool (blast)." [Online]. Available: http://mtc.epfl.ch/software-tools/blast/

[7] "Microsoft static driver verifier." [Online]. Available: http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

[8] S. E. Velder and A. A. Shalyto, "Introdaction in verificatin of automaton based programs using model checking," *Information control systems*, no. 3, pp. 27–38, 2007. [Online]. Available: http://is.ifmo.ru/download/27-38.pdf

[9] A. A. Shalyto and H. I. Tukkel, "From turing programming to automata-based programming," *PC World (Russia)*, no. 2, 2002.

[10] "Reative object language (rebeca)." [Online]. Available: http://khorshid.ece.ut.ac.ir/~rebeca/index.htm

[11] "New symbolic model verifier." [Online]. Available: http://nusmv.irst.itc.it/

[12] R. Alur, S. Chaudhuri, and P.Madhusudan, "A fixpoint calculus for local and global program flows," in *33rd Annual Symposium on Principles of Programming Languages (POPL)*, 2006. [Online]. Available: http://www.cse.psu.edu/~swarat/pubs/popl06.pdf

[13] S. Chaudhuri, "Subcubic algorithms for recursive state machines," in *35th Annual Symposium on Principles of Programming Languages (POPL)*, 2008. [Online]. Available: http://www.cse.psu.edu/~swarat/pubs/popl08.pdf