Simulator for Real-Time Abstract State Machines

P. Vasilyev[1,2⋆]

[1] Laboratory of Algorithmics, Complexity and Logic, Department of Informatics, University Paris-12, France
[2] Computer Science Department, University of Saint Petersburg, Russia

Abstract. We describe a concept and design of a simulator of Real-Time Abstract State Machines. Time can be continuous or discrete. Time constraints are defined by linear inequalities. Two semantics are considered: with and without non-deterministic bounded delays between actions. Simulation tasks can be generated according to descriptions in a special language. The simulator will be used for on-the-fly verification of formulas in an expressible timed predicate logic. Several features facilitating the simulation are described: external functions definition, delays settings, constraints specification, and others.

1 Introduction

Usually the process of software development consists of several main steps: analysis, design, specification, implementation, and testing. The steps can be iterated several times and accompanied by this or that validation. We are interested in the validation by simulation of program specifications with respect to the given requirements. We consider real-time reactive systems with continuous or discrete time. Time constraints are expressed by linear inequalities and programs are specified as Abstract State Machines (ASM) [1]. The requirements are expressed in a First Order Timed Logic (FOTL) [2, 3]. The specification languages we consider are very powerful. Even rather simple, "basic" ASMs [4] are sufficient to represent any algorithmic state machine with exact isomorphic modeling of runs. This formalism bridges human understanding, programming and logic. The ASM method has a number of successful practical applications, e.g., SDL semantics, UPnP protocol specification, semantics of VHDL, C, C++, Java, Prolog (see [5, 6]).

To express properties of real-time ASMs we use FOTL. This logic is clearly undecidable. There exist practical decidable classes [2, 3]. We know from practice that most errors in software can be revealed on rather simple inputs ; for reactive systems this means that finite models of small complexity are usually sufficient to find very serious errors.

Thus, we design our simulator as some kind of partial, bounded, on-the-fly model-checker. We consider two semantics, both with instantaneous actions, one without delays between actions, another one, more realistic, with bounded non deterministic delays between actions (by action here we mean an update of the current state, we make it more precise below). The simulator checks the existence of a run for a given input,

⋆ Partially supported by ECO-NET project No 08112WJ.

Vasilyev P. (2006).
Simulator for Real-Time Abstract State Machines.
In *Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 202-205
DOI: 10.5220/0002501002020205

outputs details of the run that can be specified in a special language, and checks the requirements formula for this run. There are several implementations of ASM interpreter or compiler, such as the Microsoft AsmL [7], Distributed ASML [8], Michigan interpreter, Gem-Mex. These systems do not deal with real-time ASMs or predicate logic requirements.

2 Timed ASM

In this paper by an ASM we mean *timed basic Gurevich abstract state machine*. A timed ASM is a tuple $(V, IS, Prog)$, where V is a vocabulary, IS is a description of the initial state, and $Prog$ is a program. The vocabulary consists of a set of sorts, a set of function symbols, and a set of predicate symbols. The following pre-interpreted sorts are included: $\mathcal{R}$ is the set of reals; $\mathcal{Z}$ is the set of integers; $\mathcal{N}$ is the set of natural numbers; $Bool$ is the set of Boolean values: *true* and *false*; $\mathcal{T} = \mathcal{R}_+$ special time sort; $Undef = \{undef\}$ is a special sort used to represent the *undefined* values.

All functions of an ASM are divided into two categories: *internal* and *external functions*. Internal functions can be modified by the ASM. External functions cannot be changed by the ASM. On the other hand, the functions can also be divided into *static* and *dynamic functions*. Dynamic external functions represent the input of the ASM. A static function has a constant interpretation during any run of the ASM. Among the static pre-interpreted functions of the vocabulary are arithmetical operations, relations, and boolean operations. The equality relation "=" is assumed to be defined for all types. The timed ASMs use a special time sort $\mathcal{T}$ and a nullary function $CT : \ \rightarrow \mathcal{T}$ which returns the current "physical" time. Only addition, subtraction, multiplication and division by a rational constant, standard equality and inequality relations are supported.

The program of the timed ASM is defined in a usual way as a sequence of instructions of several types. The main constructions are: **while-do** and **repeat-until** loops, **forall** and **choose** statements, etc.: a single *update rule* in the form of an assignment $A = \{f(x_1, \ldots, x_k) := \theta\}$; a *parallel block* of update rules $[A_1; \ldots; A_m]$ which are executed simultaneously (this block is called an *update block*); a *sequential block* of update rules $\{A_1; \ldots; A_m\}$ which are executed in the order they are written; a *guarded rule*, where $Guard_i, i \in 1, \ldots, n$ are guard conditions and $A_i, i \in 1, \ldots, n+1$ are statements:
if $Guard_1$ **then** A_1 **elseif** $Guard_2$ **then** A_2 $\ldots$ **else** A_{n+1}

3 Simulator Configuration

An external function definition looks as follows: $f : \mathcal{X} \rightarrow \mathcal{Y}$, where f is its name, $\mathcal{X}$ is an abstract sort, $\mathcal{T}$ is the pre-interpreted time sort, $\mathcal{Y}$ is an abstract sort or pre-interpreted sort $\mathcal{R}$. The sort $\mathcal{X}$ can be enumerated as a sequence of natural numbers. Thus, we can define a function as follows: $f(i) := (t_1^i, f_{12}^i; t_2^i, f_{23}^i; \ldots; t_k^i, f_{kk+1}^i; \ldots)$ where $i \in 1, \ldots, n$, n is the cardinality of the sort $\mathcal{X}$, $t_1, t_2, \ldots, t_k, \ldots$ are start time points of intervals, $f_{12}^i, f_{23}^i, \ldots, f_{kk+1}^i, \ldots$ are function values defined on the time left-closed right-open intervals. Each function can be also defined by an expression.

All updates in ASM [1] are instantaneous. In reality, it may take some time to perform an action. We model this by non-deterministic bounded delays between actions that remain instantaneous. Some ideas of managing the process of time propagation already appeared, for example in [9, 10]. We define a function of time delay $\delta : \mathcal{S} \rightarrow \mathcal{T}$ on the set of all statements and expressions which is denoted by $\mathcal{S}$. For the sequential composition the delay is calculated as a sum of delays of each function. For the parallel composition the delay is the maximum of the delays of all functions. The delays can be used in different ways. For example, we can have only two different time delays for slow and fast operations making difference between operations with internal and *shared variables* which are usually slower than operations with internal variables of a process. Another way is to specify delays manually for each function or operation.

We consider a nondeterministic situation as a tuple of all possible choices with several ways of element selection. The method should be defined in the configuration file of the simulator by one assignment (the abbreviation **ndr** stands for non-determinisms resolution). It can be the first (last), minimal (maximal) element of the tuple.

To express requirements for ASMs one needs a powerful logic. In this work we consider a First Order Timed Logic (FOTL) [2, 3] for representing the requirements. For the Lamport's Bakery [11] the following properties are required: *Safety:* $\forall pq\, \forall t(p \neq q \rightarrow \neg(CS_p^\circ(t) \wedge CS_q^\circ(t))$
Liveness: $\exists c_1 c_2\, \forall pt(n_p^\circ(t) > 0 \rightarrow \exists t'(t < t' < t + c_1 \cdot \delta_{int} + c_2 \cdot \delta_{ext} \wedge CS_p^\circ(t')))$ where $\delta_{int} > 0, \delta_{ext} > 0$ (in the degenerated case we should use another formula).

4 Timed Abstract State Machine Semantics

In brief the process of simulating a system defined by an ASML specification in our approach consists of the following steps: 1. Calculation of the next time point in which at least one guard is true. 2. State update, which is represented by one or more statement blocks. 3. Evaluation of constraints before and after the state update.

In the sequential mode of execution the next operation is taken, all required calculations are made and the state of the machine is changed. The current time value is simply incremented by the value of current instruction time delay.

We consider a block of parallel instructions as a block of *sub-machines* running in the same way as the top-level ASM but in its own space of states. Each sub-machine has its own state change. The total change of state will be calculated as a union of state changes of all sub-machines. If all pairwise intersections of state changes are empty then they are consistent. If a parallel block is looped it can happen that there are no instructions for the current time moment to be executed. To avoid infinite looping we have to wait for the closest time moment at which at least one of the guards turns to true and some instructions can be executed. If such a time point exists it is calculated and if it is not the time to exit the loop, the instructions concerning this time point are executed. A case when all the delays are set to zero is detected by the simulator and a warning message can be sent to the user.

5 Conclusion

In this paper several new important features concerning the semantics of the ASM based language were described. These features will help us to build and verify specifications of real-time systems via a customizable simulation of the models. The most important parameters of simulation can be configured, i.e. external functions, time delays for language operations and constructs, non-determinism resolving. The whole project is aimed at development of a simulator for the described version of ASM language extension where the results of the current work are used. At the moment a simulator prototype is ready, which implements most of the specified features: lexical and syntactical analysis, building a parse tree containing full information, loading definitions of external functions from a file, loading the simulation parameters, simulation of most constructs and operations of Timed ASML, output the results of simulation.

References

1. Gurevich, Y.: Evolving algebras 1993: Lipari Guide. In Egon, B., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
2. Beauquier, D., Slissenko, A.: A first order logic for specification of timed algorithms: Basic properties and a decidable class. Annals of Pure and Applied Logic **113** (2002) 13–52
3. Beauquier, D., Slissenko, A.: Periodicity based decidable classes in a first order timed logic. (Annals of Pure and Applied Logic) 38 pages. To appear.
4. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Transactions on Computational Logic **1** (2000) 77–111
5. Huggins, J.: (University of Michigan, ASM homepage) http://www.eecs.umich.edu/gasm/.
6. Börger, E. Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. (2003)
7. Foundations of Software Engineering — Microsoft Research, Microsoft Corporation: AsmL: The Abstract State Machine Language. (2002) http://research.microsoft.com/fse/asml/.
8. Soloviev, I. Usov, A.: The language of interpreter of distributed abstract state machines. Tools for Mathematical Modeling. Mathematical Research. **10** (2003) 161–170
9. Börger, E. Gurevich, Y., Rosenzweig, D.: The bakery algorithm: yet another specification and verification. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 231–243
10. Cohen, J., Slissenko, A.: On verification of refinements of timed distributed algorithms. In Gurevich, Y., Kutter, P., Odersky, M., Thiele, L., eds.: Proc. of the Intern. Workshop on Abstract State Machines (ASM'2000), March 20–24, 2000, Switzerland, Monte Verita, Ticino. Lect. Notes in Comput. Sci., vol. 1912, Springer-Verlag (2000) 34–49
11. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. In: Communications of ACM, 17(8). (1974) 453–455