

Simulator-Model Checker for Reactive 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. A concept and design of a simulator of Real-Time Abstract State Machines is described. 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. The simulator is easily configurable. Simulation tasks can be generated according to descriptions in a special language. The simulator is used for verification of formulas in an expressible timed predicate logic. Several features that facilitate the simulation are described: external functions definition, delays settings, constraints specification, and others.

1 Introduction

We know that all the phases of development are accompanied by this or that validation and the cost of errors at the specification level is much more expensive than that of implementation. Therefore, we are interested in the validation by simulation of a program specification with respect to the requirements. The specification languages based on the formalism of Abstract State Machines [1] have been proved to be a very powerful technique for specifying algorithms, formal semantics and interactive systems. Even rather simple, “basic” ASMs [2] are sufficient to represent any algorithmic state machine with exact isomorphic modeling of runs. This formalism bridges human understanding, programming and logic. The ASM specification method has a number of successful practical applications in specifying formal semantics, protocols, interactive systems. Some of these specifications were validated by hand or by simulation/testing with the help of tools like AsmL [3]. However these systems are not real-time. Our goal is the simulation of systems with real-time constraints involving arithmetics operations.

Real-time systems imply using time (explicit or implicit) and time constraints in the specification language and in user requirements. By default the ASM formalism doesn't have a fixed time model. Several approaches exist, the first was [4] then followed [5,6]. In some works the explicit using of time function is used like in [3]. In the work [7] the properties of a real-time ASM semantics are

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

described. Some of the works propose the ideas of time delays treatments [8, 6, 9]. In [9] time is a kind of resource consumed by the specified system.

In this article we consider timed reactive abstract state machines and describe a simulator-model checker which generates finite runs for given inputs and checks properties for these runs.

To express properties of real-time ASMs we use First Order Timed Logic (FOTL [5]), a predicate logic with arithmetics. This logic is clearly undecidable. There exist practical decidable classes [5, 10, 11], however the decidability algorithm is not so simple to apply. We know from the 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. However, the user should have convenient means to construct inputs.

Thus, we design our simulator as some kind of partial, bounded model-checker. It is presumed to be light, portable and easily configurable. The simulator provides a language to describe a timed semantics and to generate inputs. 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, 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 as Microsoft AsmL [3], Distributed ASML [12], Core ASM project [13], Michigan interpreter [14], XASM [15], and ASMGofor [16].

These systems do not deal with real-time ASMs or predicate logic requirements. To simulate real-time ASMs and we tried to develop an easily configurable simulator. To do that we needed some additional features such as explicit time manipulation, built-in features for time propagation and delays maintenance, and more powerful capabilities for constraints and properties defining. To summarize the problem we consider the following features :

- real-time reactive systems with continuous or discrete time,
- time constraints expressed by linear inequalities,
- programs are specified as ASMs,
- requirements are expressed in FOTL.

2 Timed ASM

In this paper by an ASM we mean a simple type of timed abstract state machine. A timed ASM is a tuple $(VOC, INIT, PROG)$, where VOC is a vocabulary, $INIT$ 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.

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. Both discrete time (natural numbers including zero) and continuous time (we use only finite sets of intervals and points) are supported by the simulator. A run of an ASM is a mapping from time to states. Each state is an interpretation of the vocabulary of the ASM. Thus, from the “run viewpoint” a dynamic function is a function of time.

In this work we admit only piecewise linear inputs defined on sets of left-closed right-open intervals $[t_k, t_{k+1})$, where t_k are time points, k is natural. The internal functions will be also piecewise linear defined on sets of left-open right-closed $(t_k, t_{k+1}]$ (it is implied by the structure of ASM).

3 Timed ASM syntax

The program of the timed ASM is defined in a usual way as a sequence of instructions of several types. Here we show the main constructions though there are also **while-do** and **repeat-until** loops, **forall** and **choose** statements, etc.:

- A single *update rule* in the form of an assignment $A = \{f(x_1, \dots, x_k) := \theta\}$;
- A *parallel block* of update rules $[A_1; \dots; A_m]$ which are executed simultaneously (this block is called an *update block*);
- A *sequential block* of update rules $\{A_1; \dots; A_m\}$ which are executed in the order they are written;
- A *guarded rule*, where $Guard_i, i \in 1, \dots, n$ are guard conditions and $A_i, i \in 1, \dots, n + 1$ are statements:
if $Guard_1$ **then** A_1 **elseif** $Guard_2$ **then** A_2 **... else** A_{n+1}

In this section most of the constructs of the Timed ASM language are described. The Timed ASM language syntax is similar to MS AsmL or Distributed ASML as we tried to preserve the existing syntactical features.

3.1 Sequential and parallel blocks of statements

A sequential block of statements is simply a sequence of statements which are executed one after another. We have chosen the following way to specify a sequential composition of statements P and Q like:

```
{ P Q }
```

A parallel block is a set of statements for which the order of statements is not important. All statements in a parallel block are executed simultaneously. We have chosen this short variant:

```
[ P Q ]
```

3.2 Types

To define a new type as a set of N constants the following construction should be used:

```
type new_type_1 = {val1, val2, ... valN};
```

The following definition specifies a mapping from a product of N types to the type *result_type*.

```
type new_type_2 = type1, type2, ... typeN -> result_type;
```

3.3 Functions

All functions, both of zero and non-zero arity, are defined with the help of *var* keyword. If the type of the defined function is simple (Integer, Real, a set of constants) one can use an optional initialization.

```
var variable_name [= initial_value] : variable_type;
```

In general the following definition is used (*type_name* is a name of predefined or user-defined type):

```
var function_name : type_name;
```

3.4 Iterators and selection

For the instructions which perform an action S over a set U in parallel or over an element of a set we use this syntax ($b(u)$ is an optional condition):

```
foreach u in U where b(u) do S  
choose u in U where b(u) do S
```

3.5 Predicate formulas

To represent a predicate formula we will use a common syntax that is used in most of the ASM based languages:

```
forall x in Y holds F(x)  
exists x in Y where F(x)
```

Both keywords "in" and ":" can be used to specify the membership of a variable x to the set Y . The predicate formula itself is specified by $F(x)$ — a FOTL expression of Boolean type.

3.6 Token example

Here is an example of an ASM specification. It consists of two guarded updates in a loop. As an input it reads the values of the external functions *Pass*, *d1*, and *d2*. The *Pass* function defines the next process that wants the token. If the conditions turn to true, the token is given to the next process.

```

type ProcessNo = {0..4}; // the set of all processes
var Curr = 0: ProcessNo; // the number of current process
var Last: Time;         // the time moment of last update
var Pass: ProcessNo;   // external function
var d1, d2: Float;     // external functions

Main() {
  Last := 0;
  while (CT <= 10) do [
    if ((Pass > 0) and (Pass != Curr)) then [
      if (CT >= Last+d1 and CT <= Last+d2) then Curr := Pass;
      Last := CT;
    ]
  ]
}

```

4 External functions definition

An external function definition looks as follows: $f : \mathcal{X} \rightarrow \mathcal{Y}$, the corresponding timed version of the function is the following: $f^\circ : \mathcal{T} \times \mathcal{X} \rightarrow \mathcal{Y}$, where f is the name of the function, \mathcal{X} is an abstract sort or a direct product of two abstract sorts, \mathcal{T} is the pre-interpreted time sort, \mathcal{Y} is an abstract sort or pre-interpreted sort \mathcal{R} . The timed version of a function f° can not be used explicitly in an ASM specification, it is used in requirements. A function can change its interpretation during the run of the ASM, therefore the time dimension \mathcal{T} allows us to use the time variable as an independent variable. Definite time variable values split the whole run of an ASM into a sequence of “slices” in which the interpretations are constant.

The sort \mathcal{X} can be enumerated as a sequence of natural numbers. Thus, we can define a function as follows:

$$f(i) := (t_1^i, f_{12}^i; t_2^i, f_{23}^i; \dots; t_k^i, f_{kk+1}^i; \dots)$$

where $i \in 1, \dots, n$, n is the number of intervals or the cardinality of the sort \mathcal{X} , $t_1, t_2, \dots, t_k, \dots$ are start points of intervals, $f_{12}^i, f_{23}^i, \dots, f_{kk+1}^i, \dots$ are function values defined on the time left-closed right-open intervals. If \mathcal{X} is a product of two abstract sorts, $\mathcal{X} = \mathcal{X}' \times \mathcal{X}''$ then the definition will look as follows:

$$f(i, j) := (t_1^{i,j}, f_{12}^{i,j}; t_2^{i,j}, f_{23}^{i,j}; \dots; t_k^{i,j}, f_{kk+1}^{i,j}; \dots)$$

where $i \in 1, \dots, n, j \in 1, \dots, m$, n is the number of intervals or the cardinality of the sort \mathcal{X}' , m is the cardinality of the sort \mathcal{X}'' , and all other designations are analogous to the ones in the previous sequence.

Each function can be also defined by an expression. The following additional pre-interpreted variables can be used in the definitions: t_a — absolute time, t_r — relative time, varying in the time interval $[0, t_{k+1} - t_k)$, for each k . Thus, for $[t_k, t_{k+1})$ the following holds: $t_a = t_k + t_r$.

5 Delay settings

All updates in ASM [1] are instantaneous. In reality, it may take some time to perform an update. We model this by non-deterministic bounded delays between actions that remain instantaneous. In the current implementation of the simulator the delay is calculated deterministically. The value of a delay depends on the complexity and quantity of instructions used in the specification. Non-deterministic bounded delays will appear in the next version of the simulator. Some ideas of managing the process of time propagation already appeared, for example in [8, 6]. We propose several ways of dealing with delays depending on the purposes of the user. We define a function of time delay δ on the set of all statements and expressions which will be denoted as \mathcal{S} . The function of time delay can be specified as $\delta : \mathcal{S} \rightarrow \mathcal{T}$. For each operation the value of time delay function can be set individually and it will be used in the process of simulation.

There are two different types of calculations: sequential composition and parallel composition. For the sequential composition $f \circ g$ the delay is calculated as a sum of delays of each function, i.e. $\delta(f \circ g) = \delta(f) + \delta(g)$. For the parallel composition $\parallel f_i$ the delay is the maximum of the delays of all functions, i.e. $\delta(\parallel f_i) = \max_i \delta(f_i)$.

This function δ can be used at different levels of abstraction. For example, it can be used at the level where we have only two different time delays for slow and fast operations making difference between operations with internal and *shared variables*. The notion of shared variables is used to mark the variables that can be accessed from several processes. This kind of variables is one of the methods for implementing inter-process communication. Operations with shared variables are usually slower than operations with internal variables of a process, as they imply several inter-process operations. In our Bakery algorithm the field **number** of the **Bakery** class is a shared variable. Thus, we can specify delays in various ways, e.g.:

- Two dedicated delays δ_{ext} (operations with external functions) and δ_{int} (operations which deal only with internal functions), $\delta_{ext} > 0$, $\delta_{int} > 0$;
- The same as the previous one but all internal operations are instantaneous: $\delta_{ext} > 0$, $\delta_{int} = 0$;
- No delay, all operations are instantaneous: $\delta_{ext} = \delta_{int} = 0$. In this degenerated case we have a standard abstract state machine;

- A generalized case: there are no predefined delays for all operations. We have a set of delays each of them corresponds to one function or operation. All of them have to be defined manually.

Examples (in the configuration file we write $d(\langle \text{instruction} \rangle) = \langle \text{delay} \rangle$):

$d(+)$ = 1; $d(*)$ = 2; $d\text{ext}$ = 3; $d\text{int}$ = 1;

6 Resolving non-determinisms

A typical example of nondeterministic choice is the **choose** statement when one of the possible values has to be chosen. We consider this situation as a tuple of all possible choices with several ways of element selection. The method should be provided in the configuration of the simulator. It is set by one assignment in the configuration file (the abbreviation **ndr** stands for "non-determinisms resolution"). Here some methods are shown:

- The first (last) element of the tuple: **ndr** = **first**,
- The minimal (maximal) element of the tuple, if some order is defined for the elements: **ndr** = **min**,
- The element defined by its index — a sequence of natural numbers with a predefined step: **ndr** = **seq start** $\langle x \rangle$ **step** $\langle y \rangle$,
- The element defined by its index; the index is provided by a random numbers generator: **ndr** = **random**.

For example, consider a tuple (5, 1, 3, 2, 9, 6, 4). The first and the last elements will be 5 and 4. The minimal and the maximal elements will be 1 and 9. If we define a sequence of natural numbers starting from 1 with the step of 3 we will get a resulting sequence as follows: 5, 2, 4, 9, 3, 6, 1.

7 Properties

To express requirements for ASMs one needs a powerful logic. In this work we consider a First Order Timed Logic (FOTL) [5,10] for representing the requirements. Here is an example of a FOTL property for the Token example. It defines the liveness property.

$$\forall t_1 t_2 (last^\circ(t_1) < last^\circ(t_2) \wedge d_1^\circ(t_1) \leq last^\circ(t_2) - last^\circ(t_1) \leq d_2^\circ(t_1) \Rightarrow curr^\circ(t_1) \neq curr^\circ(t_2))$$

7.1 First Order Timed Logic

The main idea of FOTL is to choose a decidable theory to work with arithmetics or other mathematical functions, and then to extend it by abstract functions of time that are needed to specify the problems taken under consideration. In some way, the theory must be minimal to be sufficient for a good expressivity. For the purposes of the present work we take the theory of mixed real/integer addition with rational constants and unary multiplications by rational numbers. This theory is known to be decidable [17]. Though we can consider either discrete time as non negative integers or continuous time as non negative reals, we take the case of continuous time.

Syntax of FOTL The vocabulary W of a FOTL consists of a finite set of sorts, a finite set of function symbols and a finite set of predicate symbols. To each sort a set of variables is attributed. Some sorts are pre-interpreted and have a fixed interpretation. Such sorts are real numbers \mathcal{R} and the time sort \mathcal{T} . Therefore, we have the same sorts as in the description of the Timed ASM including the time sort \mathcal{T} . Other sorts are finite and if a finite sort has a fixed cardinality it can be considered as a pre-interpreted sort. Some functions and predicates are also pre-interpreted. These are addition, subtraction, and scalar multiplication of reals by rational numbers. The pre-interpreted predicates are $=, \leq, <$ over real numbers. The vocabulary of FOTL also contains equality for all types of objects and a CT° function to define the time. Any abstract function is of the type $\mathcal{T} \times \mathcal{X} \rightarrow \mathcal{Y}$, where \mathcal{X} is a finite sort or a direct product of two finite sorts, and \mathcal{Y} is a finite sort or the pre-interpreted sort \mathcal{R} . An abstract predicate is of the following type $\mathcal{T} \times \mathcal{X} \rightarrow Bool$ with the same type of sort \mathcal{X} .

Semantics of FOTL For FOTL the notions of interpretation, model, satisfiability and validity are treated as in first order predicate logic except the pre-interpreted symbols of the vocabulary. Therefore, $\mathcal{M} \models F, \mathcal{M} \not\models F$, and $\models F$ where \mathcal{M} is an interpretation and F is a formula, denote correspondingly that \mathcal{M} is a model of F , \mathcal{M} is a counter-model of F , and F is valid.

8 Timed Abstract State Machine Semantics

8.1 Simulation process

In brief the process of simulating a system defined by a Timed ASM 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.

Similar to “C” programming language and Java, the `Main()` method is used as the top-level entry point of a specification. From the beginning of this method the simulation of the top-level abstract state machine starts. At first, the main ASM on the top level of the specification is executed. Further the process of simulation runs as follows.

8.2 Sequential execution

In the sequential mode of execution the next operation is taken, all required calculations are made and the state of the machine is changed. Then the next construct or instruction is taken and so on.

If the user has specified the time delays for Timed ASM operations then the time value is changed according to the given configuration. The current time value is simply incremented by the value of current instruction time delay.

If the delay is zero then the update will be instantaneous, so the value of the function will be defined in the certain point of time.

8.3 Block of parallel instructions

All instructions in a parallel block are considered to be executed in parallel. We consider it as block of *sub-machines* running in the same way as the top-level ASM but in its own space of states. Each sub-machine does not interact or affect other sub-machines. If several sub-machines have been executed from a parallel block they have the same initial state but their end states after the execution, in general, are different. When all of them finish their execution the total state change of the upper level sub-machine has to be calculated. Let each sub-machine be designated as SM_i , $i = 1, 2, \dots, N$, where N is the number of parallel instructions in the block. Let SC_i be the final state change of each sub-machine. The total change of state will be calculated as $\bigcup_{i=1}^N SC_i$. If $\bigcup_{i \neq j} SC_i \cap SC_j = \emptyset$, the state changes are consistent and the situation is entirely correct. Otherwise, we have some inconsistencies and the execution will be stopped. But, if the user has configured the non-determinism resolving procedure the execution can be continued. In this case the new values of the functions from the intersection set are chosen in accordance to the rules of simulation configuration. After the total state change is calculated all further instructions start from the new time point (see Delay settings). For example, consider the following parallel block:

```
[ x := 1;
  y := 2;
  z := 3;
  x := x + y + z;
]
```

It is clear that the variable x is affected two times and the user can choose if the simulation process has to be stopped or it will be continued like in the situation of a non-deterministic choose operation. If the first assignment $x := 1$; is chosen the total time delay of the whole parallel block should be smaller than the total time delay of the parallel block with the second assignment $x := x + y + z$;

8.4 Looped parallel block

If a parallel block is looped it can happen that there are no instructions for the current time moment to be executed. In this case 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. Of course, we can increment the time value until some of the guards turns to true but it is not efficient and it is hard to choose the increment step. Therefore, this problem is solved analytically. This time point is calculated and if it is not the time to exit the loop, the instructions concerning this time point are executed. If there are several guards that become true at this time point then all of the guarded instructions are executed. If such a time point does not exist the execution will be stopped.

The execution of parallel instructions is performed just as in the case of a basic parallel block. The summary time delay and the resulting state change is calculated according to the rules specified for the parallel block. After the state is updated the simulation proceeds to the next iteration with a new time value. 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. The following is a simple example with guarded updates in a loop.

```

{ x := 0; y := 0; z := 0;
  while (CT < 16) do [
    if (CT >= 12) then x := x + 1;
    if (CT >= 8) then y := y + 1;
    if (CT >= 17) then z := z + 1;
  ]
}

```

In this example it is clear that the third guarded update will not be executed. The others will be executed if the value of CT is below 16 when the simulator starts processing the loop statement. Each of these updates will be executed several times depending on the value of time delay specified for operations used in the update. In the first update there are three operations: read the value of x, add 1 to the value of x, save the calculated value to x. Therefore, the time delay for both of updates is the same and the time intervals are (12,16) and (8, 16) correspondingly. So if the value of CT was below 8 when the loop was entered the first guarded update will be executed about two times less than the second one.

8.5 Verification

In brief, the verification feature consists of the following steps: parsing a formula, elimination of abstract functions, elimination of other functions, elimination of quantifiers over the time and evaluation of the obtained formula. Let F be a FOTL formula defining a property.

$$F \equiv \forall t_1 \dots t_k \bigvee_i \bigwedge_j \left(\sum_k H_{ijk}(T_{ijk}) \leq 0 \right)$$

where H_{ijk} is a term and T_{ijk} is a linear combination of time variables $t_1 \dots t_k$. Let I_m are the intervals where the values of functions are constant and there is only one term $H_{ij}(T_{ij})$. Therefore, the atomic formula

$$H_{ij}(T_{ij}) \leq 0 \equiv \bigwedge_m (T_{ij} \in I_m \Rightarrow H_{ij}(T_{ij}) \leq 0) \equiv \bigwedge_m (T_{ij} \notin I_m \vee H_{ij}(T_{ij}) \leq 0)$$

Now we can calculate the values of $H_{ij}(T_{ij}) \leq 0$ as they are constant and we get a big system of inequalities:

$$\forall t_1 \dots t_k \bigvee_i \bigwedge_{jm_k} (T_{ij} \notin I_{m_k})$$

So, the modified includes only quantifiers over time and a system of linear inequalities with time variables. After the quantifier elimination procedure we can get an expression with constants that can be easily calculated.

9 Conclusion

In this paper the most important features concerning the simulator of the Timed ASM language were described. The simulator is based on the described semantics features that 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 semantics of Send and Receive statements is described. These statements are useful for encapsulation of data in a class and provide a method for communication between agents and synchronising the processes.

At the moment a simulator prototype is ready, which implements most of the specified features. The following is the list of the implemented features:

- Lexical and syntactical analysis with building a parse tree;
- Loading definitions of external functions from a file;
- Loading the simulation parameters;
- Simulation of most constructs and operations of Timed ASM;
- Checking the specified formulas during the simulation;
- Simulation results output with the history of all state changes;
- Verification of simulation results against the user-defined properties.

10 Acknowledgements

I'd like to thank professor Danièle Beauquier and professor Anatol Slissenko for their help and useful comments concerning the work on the simulator.

References

1. Gurevich, Y.: Evolving algebras 1993: Lipari Guide. In Egon, B., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
2. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic* **1**(1) (2000) 77–111
3. Foundations of Software Engineering — Microsoft Research, Microsoft Corporation: AsmL: The Abstract State Machine Language. (2002) <http://research.microsoft.com/fse/asml/>.
4. Gurevich, Y., Huggins, J.: The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In: Proceedings of CSL'95 (Computer Science Logic). Volume 1092 of LNCS., Springer (1996) 266–290
5. 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**(1–3) (2002) 13–52

6. 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
7. Graf, S., Prinz, A.: Time in abstract state machines. to appear in *Fundamentae Informaticae*, Special issue on ASM 2005, Paris (2006)
8. 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
9. Ouimet, M., Nolin, M., Lundqvist, K.: Timed abstract state machines: An executable specification language for reactive real-time systems. Technical report, Massachusetts Institute of Technology (2006)
10. Beauquier, D., Slissenko, A.: Periodicity based decidable classes in a first order timed logic. *ANNALSPAL: Annals of Pure and Applied Logic* **139** (2006)
11. Beauquier, D., Crolard, T., Prokofieva, E.: Automatic parametric verification of a root contention protocol based on abstract state machines and first order timed logic. In Jensen, K., Podelski, A., eds.: *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Barcelona, Spain, March 29 – April 2, 2004*. Lect. Notes in Comput. Sci., vol. 2988, Springer-Verlag Heidelberg (2004) 372–387
12. Soloviev, I., Usov, A.: The language of interpreter of distributed abstract state machines. *Tools for Mathematical Modeling. Mathematical Research*. **10** (2003) 161–170
13. <http://www.coreasm.org/>: (The CoreASM project)
14. <http://www.eecs.umich.edu/gasm/>: (University of Michigan, ASM homepage)
15. <http://www.xasm.org/>: (XASM project)
16. <http://www.tydo.de/AsmGofer/>: (ASM Gofer)
17. Weispfenning, V.: Mixed real-integer linear quantifier elimination. In: Proc. of the 1999 Int. Symp. on Symbolic and Algebraic Computations (ISSAC'99), ACM Press (1999) 129–136