Journal of Universal Computer Science, vol. 14, no. 12 (2008), 1984-2006
submitted: 1/9/07, accepted: 1/6/08, appeared: 28/6/08 © J.UCS

Simulation of Timed Abstract State Machines with Predicate Logic Model-Checking

Anatol Slissenko[1]
Pavel Vasilyev[2]
(Laboratory of Algorithmics, Complexity and Logic (LACL)
University Paris-East (Paris 12), France
{slissenko,vassiliev}@univ-paris12.fr)

Abstract: We describe a prototype of a simulator for reactive timed abstract state machines (ASM) that checks whether the generated runs verify a requirements specification represented as a formula of a First Order Timed Logic (FOTL). The simulator deals with ASM with continuous or discrete time. The time constraints are linear inequalities. It can treat two semantics, one with instantaneous actions and another one with delayed actions, the delays being bounded and non-deterministic.
Key Words: abstract state machine, real-time, simulation, predicate timed logic, model-checking
Category: D.2.1, D.2.4

1 Introduction

Abstract State Machines (ASMs) [Gurevich, 1995], [Gurevich, 2000] without time have shown their efficiency as specification method in numerous practical applications (e.g., see [Michigan ASM], [Börger et al., 2003]). Among the advantages of ASM one can see compactness and good readability of specifications, homogeneous formalism for all levels of abstraction. This makes feasible a verification of specifications by hand or assisted by tools. One of the recent examples is the Mondex specification and verification [Schellhorn et al., 2006] that proved to be more efficient than other known methods. Specification and verification of real-time systems with time constraints involving arithmetics are more complicated. For specifications based on instantaneous actions semantics, first introduced in [Gurevich et al., 1996], the verification looks also feasible (e.g., [Beauquier et al., 2002], [Beauquier et al., 2000], [Beauquier et al., 2006]). Moreover, in some cases it can be done automatically [Beauquier et al., 2003], [Beauquier et al.,2004] though the latter involves a 'heavy machinery' like quantifier elimination procedures for the theory of real addition.

However, semantics with instantaneous actions are not always realistic, and using a more realistic semantics with delays may dramatically complicate the

[1] Member of Scholars Club of Saint-Petersburg Division of Steklov Mathematical Institute, Russian Academy of Sciences.
[2] PhD student in partnership with University Paris 12 and Saint Petersburg State University, Russia.

verification. A correct refinement of a specification with instantaneous actions into a specifications with delayed actions is much more delicate than for specifications without time, e.g., see [Cohen et al., 2008]. Hence, simulation becomes an indispensable tool of specifications validation. Simulation itself generates runs that can be analyzed by hand. This gives some kind of testing. Clearly, it would be much better to automatically check the requirements specification for the generated runs, i.e., to perform an automatic model-checking (we use the term "model-checking" as from logical viewpoint the simulator generates an interpretation of the requirements formula, and checks whether this interpretation is a model, i.e., whether it verifies the formula). It is highly desirable to model-check as complete specification as possible. By "completeness" we mean here the adequacy of the requirements formula to the engineering requirements; it is an intuitive notion — see [Sommerville, 1992]. In the case of real-time specifications a good completeness can be achieved if we use first order logics with time wherein we can express time constraints involving non-trivial arithmetics. All these considerations constitute a motivation for the design of the simulator described in this paper.

The simulator is aimed at the simulation of real-time reactive ASMs. "Reactive" means that the input is infinite (in general case), and the ASM should adequately react to this input. In particular, there is a special input, namely the time that is available via a function CT (as in [Gurevich et al., 1996], derived from *C*urrent *T*ime) though more common notation is now[3]. Time may be discrete or continuous, and time constraints are linear inequalities; the latter immediately brings us out of the scope the existing model-checking tools that use timed automata and rather weak temporal logics.

Semantics of timed ASMs may be different depending on what we specify. We consider two semantics, one with instantaneous actions and another with delayed actions. We describe them for ASMs with simple syntax that are, however, sufficient to represent main difficulties. As far as we are aware, no detailed definition of semantics for general timed ASMs was ever published. Some problems related to the semantics of timed ASM are discussed for example in [Graf et al., 2007], [Slissenko, 2004]; in [Graf et al., 2007] on can find a good bibliography on introducing time in ASMs.

The simulator deals with ASM that are constructed from updates, **if-then**-constructor, parallel and sequential composition. We do not mix specifications with data that are used for simulation. For example, one can add delays directly in the syntax, as it is done, e.g., in [Ouimet et al., 2008]. However, the delays are external to the program and useless for code generation from the specification. So we prefer not to touch in vain the specification that can be used for code

[3] We use this particular notation CT in order not to mix semantics that are different for now in different contexts.

generation, and to specify the data used for simulation outside.

The properties that the tool model-checks, are expressed in a First Order Timed Logic (FOTL) [Beauquier et al., 2002]. Such logic is an extension of the theory of real addition by many-sorted abstract functions. The abstract sorts are finite. Even very simple extensions of the theory of real addition by abstract functions give undecidable, even not enumerable (deductively incomplete) theories; for example, it follows from [Halpern, 1991] that adding two unary predicates to the theory of real addition gives a deductively incomplete theory. For specifications one usually needs much more.

Thus, the simulator is presumed to be light, portable and easily configurable [Vasilyev, 2006], [Vasilyev, 2007]. The simulator checks the existence of a run for a given input, outputs details of the run, and checks the requirements formula for this run.

Related works. Timed ASM (with instantaneous actions) were first introduced in [Gurevich et al., 1996]. A semantics more apt for formal verification, also for instantaneous actions, was developed in [Beauquier et al., 2002]. Semantics with delays can be found in [Börger et al., 1995] (without explicit time), and in [Cohen et al., 2000], [Cohen et al., 2008], [Ouimet et al., 2008]. A general study of timing is in [Graf et al., 2007]. As for simulators, time appears in some limited form in [Ouimet et al., 2008], and in AsmL [MicrosoftAsmL, 2002]. The specification language of [Ouimet et al., 2008] has full-scale expressibility, however, the simulation and verification that are described in previous papers of these authors are done only for a fragment that looks like a sub-class of timed automata. As for [MicrosoftAsmL, 2002], it has no specific tools to treat time; the user should organize all him/herself; so it is not a simulator for timed ASM.

There are tools for simulation, and even code generation, of untimed ASM such as Microsoft AsmL [MicrosoftAsmL, 2002], Distributed ASML [Soloviev et al., 2003], Core ASM [CoreASM], Michigan interpreter [Michigan ASM], XASM [XASM], and ASMGofer [ASMGofer].

To our knowledge no tool model-checks timed ASM for such expressible logics as FOTL, as in our case.

Structure of the paper. The presentation is primarily conceptual. The next section 2 gives a notion of 'basic' timed ASMs and their semantics. Our basic ASM consists of one parallel block of **if** G **then** B, where G is a guard, and B is a parallel block of updates. Such untimed ASMs suffice to prove the sequential ASM Thesis [Gurevich, 2000] about isomorphic modeling of any algorithm without time. The presented framework is based on known approaches to the semantics, though no detailed description of general semantics with delayed actions even for basic ASMs was ever published, as far as we are aware (a description in [Cohen et al., 2000] leaves some details to the reader). On the other hand, the difficulties of defining timed semantics are well seen for these ASMs.

In the same section we describe FOTL (First Order Time Logic) introduced in [Beauquier et al., 2002]. In section 3.2 we briefly describe how the inputs, delays, non-determinism in their choice are dealt with in the simulator. The main illustration of the work of the simulator is an example in section 3.3. In conclusion (section 4) we briefly discuss what development of the simulator could be reasonable. A short Appendix gives main architectural features of the simulator. It is is destined to readers interested in the implementation.

2 Timed Abstract State Machines

Syntactically, an ASM $\mathcal{A}$ is a tuple $(Voc, Init, Prog)$, where Voc is a vocabulary of sorts and functions, $Init$ defines initial values of the functions, and $Prog$ is a program. In theory $Init$ may define a set of initial values at the initial time instant but in our case it gives values of functions at 0 time instant. The type of vocabulary is the same for all ASMs we consider.

2.1 Vocabulary of Timed ASM

The sorts and functions of the vocabulary can be *pre-interpreted* or *abstract*. We consider ASMs with vocabularies of the following kind.

Pre-interpreted sorts: reals $\mathcal{R}$ (interpreted as $\mathbb{R}$), time $\mathcal{T}$ (interpreted as $\mathbb{T} =_{df} \mathbb{R}_{\geq 0}$), natural numbers $\mathcal{N}$ (interpreted as $\mathbb{N}$); boolean values $Bool$ (interpreted as $\{true, false\}$), and a sort $Undef$ with a special element $undef$ to represent 'undefined' values. The equality relation $=$ is assumed to be defined for all sorts. The natural numbers are not considered here as a subsort of reals, they are used to represent finite sorts as initial segments of $\mathbb{N}$.

Abstract sorts are finite, and of a concrete cardinality.

Pre-interpreted constants: Boolean $\{true, false\}$ each of type $\rightarrow Bool$, rational numbers $\mathbb{Q}$ (each of type $\rightarrow \mathcal{R}$), natural numbers $\mathbb{N}$ (each of type $\rightarrow \mathcal{N}$).

Pre-interpreted functions: *arithmetical operations* addition $(+)$, subtraction $(-)$ of reals, and unary multiplications of reals by rational numbers; *arithmetical relations* $=, <, \leq, >, \geq$ over reals; similar *arithmetical relations* and $+ \bmod n$ over natural numbers; *Boolean operations* $\wedge$, $\vee$, $\neg$. The function $CT :\rightarrow \mathcal{T}$, that was mentioned in section 1, represents the current absolute time. This function plays a particular role, and by default, is excluded from our discourses about input functions.

Abstract functions are of types of the form $\mathcal{X} \rightarrow \mathcal{Z}$, where $\mathcal{X}$ is a direct product of abstract sorts, and $\mathcal{Z}$ is any sort.

We need some classification of functions, not so detailed as in [Börger et al., 2003], section 2.2.3. Functions may be *static* or *dynamic*. Static functions are unchanged during the execution of the program; in particular, pre-interpreted functions (except CT) are static. With respect to the capacity of

ASM to change functions, we distinguish *external* and *internal* ones. The external functions cannot be changed by ASM, and the internal ones are those that are updated by ASM during the execution of its program. External functions can be dynamic or static, internal functions are dynamic. External dynamic functions represent *inputs*. For example, current time CT is an input by default. The internal functions are further classified into *output* functions and *proper* internal functions. The input and output functions appear in the requirements specification. The remaining ones are used inside the program. A 'variable' of programming languages is called *'nullary dynamic function'* in ASM terminology. As the abstract sorts that we consider are finite sets of a concrete cardinality, we assume in this section, without loss of generality, that all abstract functions are nullary.

Terms, their types and *formulas* are defined as usually in logic. A *guard* is a formula without quantifiers (over infinite domains).

An update is an expression of the form $f(\theta_1, \ldots, \theta_m) := \theta_0$, where f is an internal function of arity m, and θ_i, $0 \leq i \leq m$ are terms of the appropriate types.

2.2 Basic ASM

The syntax of basic ASMs uses updates, parallel composition and branching in a very limited way. We denote the parallel composition by $[R_1; \ldots; R_m]$, where R_i is a rule. In the case of basic ASM the rules are simple: an update is a rule, a parallel composition of rules is a rule, an expression of the form **if** *Guard* **then** *Rule* is a rule. A *basic ASM* is an ASM with the program of the form:

foreach x **in** S
[**if** G_1 **then** B_1
. .....
if G_k **then** B_k]

where x is a (logical) variable, S is a finite sort, G_i are guards, and B_i are parallel blocks of updates. As usually in ASM there is an implicit infinite external loop (that will be commented below in the description of semantics).

2.2.1 Semantics of Basic ASMs with Instantaneous Actions

The semantics of a basic ASM with instantaneous actions is defined as follows. First, an interpretation of abstract sorts is supposed to be given. Hence, we can forget **foreach** x **in** S assuming that for each value of x the **if-then**-rules are written explicitly. Second, as we consider reactive ASMs, the external functions are supposed to be given, in particular each input function is supposed to be defined for all time instants.

In this section we impose no constraints on the form of inputs. On the other hand, the definition of runs gives internal dynamic functions that are piecewise constant on left-open right-close intervals. For the sake of brevity we say that a

function is defined up to t^+ if it is defined on an interval $[0, t+\varepsilon)$ for some $\varepsilon > 0$. Similar, a formula $\Phi(t^+)$ is true if it is true in $(t, t+\varepsilon)$ for some $\varepsilon > 0$. Notice that piecewise constant functions may be used to represent not piecewise constant ones; for example, a function φ linear in an interval $(a, b]$ may be represented by two functions f_0 and f_1 that are constant in this interval: $\varphi(\tau) = f_0 + f_1(\tau - a)$.

Let $\mathcal{A}$ be an ASM in notations just above (without **foreach**). We define its run ρ by induction. More concretely we try to define the values of internal dynamic functions of $\mathcal{A}$ for all time instants, if possible. At the initial time instant 0 all the dynamic functions are defined due to $Init$. We assume that the internal dynamic functions are defined up to 0^+ by simple extensions of its values; we will see that we do not need to know in what concrete interval $(0, 0+\varepsilon)$ they are defined.

Suppose that ρ is defined up to t^+. The following cases are possible.

Case 1. There exists a time instant $T > t$ whereat at least one guard is true with the values of internal dynamic functions at t^+ (thus, we extend these functions up to T), and no guard is true in (t, T). Let $\{G_i\}$, $i \in I \subseteq \{1, 2, \ldots, k\}$, be all the guards that are true at T. All the updates of $\{B_i\}$, $i \in I$, constitute an update set to process. If these updates are consistent then they are accomplished, and the run in defined up to T^+: the new values are assigned at T^+. Otherwise, the run is defined on $[0, T]$, and undefined after T.

Case 2. There is no time instant $T > t$ whereat at least one guard is true with the values of internal dynamic functions at t^+, and no guard is true in (t, T). Two subcases are possible.

Case 2a. All the guards are false in (t, ∞). In this case the values of internal dynamic functions are extended on (t, ∞), and hence the run is defined everywhere.

Case 2b. There is a guard that is true at some $T > t$, but there is no first time instant after t whereat a guard is true. Take the infimum of such T, let it be T_0. Extend the internal functions on $(t, T_0]$ if $T_0 > t$. Starting from T_0 the run is undefined.

Remark. The case 2b can be illustrated by the following example. Consider an ASM consisting of a rule **if** $x = 0 \wedge 5 < CT$ **then** $x := 1$ with internal function $x :\rightarrow \{0, 1\}$ whose initial value is 0. The instant T_0 of Case 2b is 5. If we define run with $x = 1$ on $(5, \infty)$, we will have a run where the guard is false everywhere but there is a change of value of x. But this change should be fired by a true guard. This is an inconsistency that may imply incorrect conclusions, for example, in the verification. □

2.2.2 Semantics of Basic ASMs with Delayed Actions

In semantics with delayed actions we assume that the calculation of the value of a function or the processing of a constructor (like **if**, **then**, parallel composition,

update) takes some time. This time is bounded by an interval, and within this interval the delay is taken non-deterministically. Such a semantics makes from an ASM, even a basic one, a distributed algorithm with bounded asynchrony.

Here we consider one representative example of semantics with delayed actions. Consider an ASM, as above in section 2.2, i.e., a parallel composition of k statements **if** G_i **then** B_i. Recall that all abstract functions are nullary and that a term is either a Boolean expression or an affine function that we refer to as *arithmetical expression*. An atomic formula is either a Boolean valued function or an inequality $\theta\omega\theta'$, where θ and θ' are arithmetical expressions, and ω is an order relation. All the intervals defining delays are supposed to be finite.

We make the following assumptions about delays:

• For each nullary function f there is attributed an interval (if one wishes more generality — a set) Δ_f of delays of reading its value.

• For each Boolean or arithmetical operation/relation φ there is attributed an interval Δ_φ of delays of calculating its value.

• For :=, i.e., for the update constructor (writing), there is attributed an interval Δ_{upd} of delays; there are no delays for other constructors (the delays introduced above represent the main problems).

• A mode of calculating terms/formulas is fixed. E.g., we can fix a sequential mode when we execute the involved actions (i.e., readings, calculations of the results of operations) sequentially, and thus add their delays to calculate the delay of the result, or a parallel mode when all independent actions are executed in parallel, and we calculate the resulting delay using maximum and addition.

□

For each occurrence of term/formula Φ the delay of its evaluation is calculated from the chosen delays of reading the values of functions and of calculating the results of operations constituting Φ. Suppose that Φ is constituted from functions $f_1, \ldots, f_p$ and operations $\varphi_1, \ldots, \varphi_q$, each may have several occurrences in Φ. We do not distinguish occurrences of f_i from the viewpoint of delays (the read value is available to the program that may use it several times). Suppose that for each f_i there is chosen (non-deterministically) a delay d_i from Δ_{f_i}, and for each occurrence of φ_j there is chosen a delay D_i from Δ_{φ_j}. Thus, we know the instants when the arguments of operations are available, and consequently when the results are calculated. Thus, the final result is defined by Φ, $(d_i)_i$ and $(D_j)_j$ according to the chosen mode of processing. Detailed formalization of these considerations is conceptually simple but rather clumsy, so we omit it.

Concerning the availability of values we make the following precisions. If the program starts reading the value of f at t with a delay d then it gets this value at $(t+d)$. Similar for the calculation of the value of Φ at t with a delay D, assuming that the values of the arguments are available at t: the resulting value of Φ is available at $(t + D)$. The updates are treated differently. If the program starts

an update $f := \theta$ at t with a delay δ, assuming that the value of θ is available at t, then the new value of f is available at $(t + \delta)^+$.

Now we can define a run, given an interpretation of sorts and inputs. We proceed again by induction but now we analyze each external iteration of the parallel block of **if-then**-rules within another, 'internal' induction. At 0 the run, denote it by ρ, is defined. Suppose that ρ is defined up to t^+ with $t \geq 0$ and this t is an instant when the program starts the processing of the parallel block of **if-then**-rules. Suppose that all the delays are chosen. From the chosen delays we can calculate the instants when the values are read, the instants when the values of functions are calculated, and the instants when the updates are fired assuming that the respective guard is true (though it may be not the case in reality). For technical simplicity we assume here that all delays are positive.

Denote by $\mathcal{G}$ all occurrences of guards, and by $\mathcal{U}(G)$, where $G \in \mathcal{G}$, its updates, i.e., the updates in B that is in **if** G **then** B. By T_G we denote the time instant of the evaluation of $G \in \mathcal{G}$, and by T_u the time instant whereat an occurrence u of update should fire if the respective guard is evaluated to true. By $\mathcal{U}(\mathcal{G}')$, where $\mathcal{G}' \subseteq \mathcal{G}$, we denote $\bigcup_{G \in \mathcal{G}'} \mathcal{U}(G)$. The notation $\rho(\tau) \models G$ means that G is evaluated to true with the values of internal function at τ in the current run ρ and the values of inputs taken at the instants of their readings that are defined by the delays.

Set $\tau_0 = t$, $\mathcal{G}_0 = \emptyset$, $\mathcal{U}_0 = \mathcal{U}(\mathcal{G}_0)$. This is the start of the 'internal'induction.

Suppose that there are defined an instant τ_n, a set of guards $\mathcal{G}_n$, and thus $\mathcal{U}_n = \mathcal{U}(\mathcal{G}_n)$, such that

- the run is defined up to τ_n^+ with $n \geq 0$ and $\tau_n < \max\{\{T_u : u \in \mathcal{U}(\mathcal{G})\} \cup \{T_G : G \in \mathcal{G}\}\}$ (i.e., not all guards or updates were analyzed),
- $\mathcal{G}_n$ is the set of guards evaluated to true in $(t, \tau_n]$.

Consider $\tau_{n+1} = \min\{\mathcal{E}_n \cup \mathcal{F}_n\}$, where $\mathcal{E}_n = \{T_u > \tau_n : u \in \mathcal{U}_n\}$ and $\mathcal{F}_n = \{T_G : T_G > \tau_n \wedge \rho(\tau_n) \models G\}$. The following cases are possible.

Case 1: τ_{n+1} is undefined, i.e., $\mathcal{E}_n = \mathcal{F}_n = \emptyset$. We extend the values of internal functions up to $T^+ =_{df} \max\{\max\{T_G : G \in \mathcal{G}\}, \max\{T_u : u \in \mathcal{U}(\mathcal{G})\}\}^+$, terminate the internal induction, and advance the first (external) induction to a new $t = T$.

Case 2: τ_{n+1} is defined. Then run is extended up to τ_{n+1} with the values of internal functions at τ_n. Set $\mathcal{G}_{n+1} = (\mathcal{G}_n \cup \mathcal{F}_n)$. Suppose that $\tau_{n+1} = \min \mathcal{E}_n$. Then the updates u with $T_u = \tau_{n+1}$ should be really executed. This is being done in a standard way: if the set of updates to execute is consistent then the respective updates give new values of the involved functions, and the run is extended to τ_{n+1}^+. Otherwise, the run stops at τ_{n+1}, and is undefined after the latter instant.

One can see that a basic ASM with the described semantics may have rather 'unstable' behavior that is hardly desirable. For example, two occurrences of

the same guard may be evaluated to different values because some function was updated after its old value was read for the evaluation of one occurrence, and its new value was read later for the evaluation of the other occurrence. The general (even basic) ASM does not impose any constraints that permit avoid such behaviors. But the expressive power of ASM, even basic ones, permits to the user to organize a well controllable interaction of processes in his/her own way.

'Submachine' semantics. One can guess that it is technically not simple to define timed semantics for such a general type of interaction between various parallel actions of a machine with a more rich syntax, for example for machines with nested sequential and parallel composition. We notice that sequential composition does not add conceptual difficulties though some technical ones appear, for example in the case of sequential updates without delays.

From pragmatical viewpoint one can incorporate constraints on interactions directly into semantics, and thus, avoid the mentioned complications. One way to do it, used in the simulator, is the following one. At the instant of entering the external loop (at t in the notations above) the program backups the current state. Then each **if-then**-branch acts as a *submachine* using this state, without interaction with other submachines. After the work of all submachines is terminated, the obtained updates are analyzed for their consistency and are processed appropriately.

The submachine semantics permits to describe the behavior of ASMs with the mentioned richer syntax much simpler.

2.3 First Order Timed Logic (FOTL)

The vocabulary of a FOTL consists of a set of *sorts*, a set of *function symbols* and a set of *predicate symbols*. The pre-interpreted sorts, abstract sorts and functions are the same as for ASM.

The logic is aimed at expressing the requirements, and the latter concern the runs. To ensure a correspondence between ASM and logic, we introduce for each abstract dynamic function $f : \mathcal{X} \to \mathcal{Z}$ of ASM its timed version $f^\circ : \mathcal{T} \times \mathcal{X} \to \mathcal{Z}$ with time as its first argument. (The notation of the simulator for timed version is `f‘`, see section 3.3).

A priori, we impose no constraints on the admissible interpretations. Thus, the notions of interpretation, model, satisfiability and validity are treated as in the first order predicate logic modulo the pre-interpreted part of the vocabulary whose interpretation is fixed.

FOTL permits to automatically describe runs of ASM with instantaneous actions in a sufficiently compact form apt for formal analysis [Beauquier et al., 2002], [Beauquier et al., 2000]. As for ASM with delays, one can also describe runs automatically but the delays impose more complicated

formulas with more quantifiers. This formalization was not yet studied from the point of view of its efficiency for formal verification. Decidable classes were neither studied as compared with the case of instantaneous actions ASMs (e.g., [Beauquier et al., 2002], [Beauquier et al., 2006]).

2.3.1 Model-Checking Procedure

We speak about a particular case of model-checking: whether a finite interpretation of a special type verifies (i.e., is a model of) a given FOTL formula. In order to arrive at algorithmically treatable runs we need to specify inputs to make them 'sufficiently constructive'. Though the current version of the simulator can process only piecewise constant inputs we give here a slightly more general framework.

In brief, the verification procedure consists of the following steps: elimination of abstract functions, elimination of quantifiers over time and evaluation of the obtained formula.

As the models generated by the simulator are finite, we should be careful about the correctness of the model-checking procedure. Here are some general considerations.

2.3.1.1 Bounding quantifiers

First, suppose that the property F to verify is of the form $\forall T\, \Phi(T)$, where $T = (t_1, \ldots, t_m)$ is a list of time variables, and Φ is quantifier-free. Usual safety properties have this form. If F is true then it is true for all T from any $R = ([0, T_1] \times \cdots \times [0, T_m])$. Thus, if F is false on some R then F is false, and we have a counter-model.

Now suppose that the ASM under consideration is such that for any finite input (we do not speak about CT here) the run is also finite, and it stabilizes 'soon' after the stabilization of the input. More precisely, the input does not change after some instant t. And after some $t' \geq t$, that is easy to calculate and that is not far from t, all abstract functions also stabilize, i.e., do not change their values. This gives a total interpretation (of finite complexity). In this case we can verify the property for a total interpretation. Not all the properties and programs are alike, for example, an ASM having CT as the only input and producing an infinite periodic output is not of this type.

Liveness properties have usually an existential quantifier. Now let F be of the form $\forall T \exists T' \Phi(T, T')$. For properties of practical interest the existential quantifier can be bounded, and the bounds are not far from the values of T for any given T. So for a given bound on T, the run should be defined up to the bound on existential quantifiers. Again, we may have two possibilities as just above.

2.3.1.2 Elimination of abstract functions.

The elimination of abstract functions when a run is defined over a finite partition of time into intervals, is rather straightforward, e.g., see [Beauquier et al., 2006].

We illustrate the elimination of abstract functions by an example. Assume that there is one partition of time into a finite number of intervals $\{\sigma\}$ such that on each interval σ of this partition any abstract function h is defined by an arithmetical term $U_h^\sigma(t)$. Suppose that the property we consider contains an inequality $a \cdot f(t_1) + b \cdot g(t_2) \leq d$, where a, b and d are rational numbers, f and g are functions depending only on time, and t_1, t_2 are time variables. We can replace the mentioned inequality by

$$\bigwedge_\sigma \big(t_1, t_2 \in \sigma \rightarrow a \cdot U_f^\sigma(t_1) + b \cdot U_g^\sigma(t_2) \leq d\big).$$

The obtained formula is a formula in the theory of real addition. The elimination of abstract predicates is similar but simpler.

2.3.1.3 Verification.

Thus, for a given finitely described run the verification of a property is reduced to the verification of a formula of the theory of real addition. Then we apply a quantifier elimination and get the truth value if the property is represented by a closed formula. If the property contains parameters then we get a quantifier free formula over these parameters. For example, formula $\exists\, x\, (0 \leq x \leq 1 \wedge 1 \leq (x + y) \leq 3)$ can be reduced to $(0 \leq y \leq 3)$.

In the case of partial run that is not defined after some instant, the property formula should be transformed into a formula with bounded quantifiers, see 2.3.1.1.

For the case treated by the simulator, namely for the case of piecewise constant functions, the verification is much simpler. It can be directly reduced to propositional case along the following lines. Suppose that the partition of time into intervals wherein functions are constant is common for all functions. Consider a formula $\forall\, t\, \Phi(t)$. It is equivalent to $\forall\, t\, \bigwedge_i (t \in \sigma_i \rightarrow \Phi(t))$, where $(\sigma_i)_i$ is the mentioned partition (we suppose that the quantifier is bounded, see 2.3.1.1). But on each σ_i all the functions are constant, so t can be eliminated as follows. Let f_i be the value of f in σ_i. Replace all occurrences of $f(t)$ in $\Phi(t)$ by by f_i; do it for all functions depending on t, and denote the result by Φ_i. The initial formula is equivalent to $\Psi =_{df} \forall\, t\, \bigwedge_i (t \in \sigma_i \rightarrow \Phi_i)$. Formula Ψ implies all Φ_i. Indeed for a given i take t in σ_i then as $(t \in \sigma_i \rightarrow \Phi_i)$ and $t \in \sigma_i$ are true, hence Φ_i is true. Thus, Ψ implies $\bigwedge_i \Phi_i$. The latter formula straightforwardly implies Ψ. Thus we arrive at a quantifier-free formula $\bigwedge_i \Phi_i$, where Φ_i does not contain t. Similarly, we can eliminate $\exists$ or several quantifiers.

3 Generation of runs

Before briefly describing the work of the current version of the simulator, we touche the question of calculating the first instant, after a given one, whereat a guard is true. The framework we consider in the next section 3.1 is more general than needed for the current version of the simulator. It is here presented to outline the perspective, and for this reason we consider in section 3.1 piecewise linear inputs.

3.1 Computing Instants of Updates

The definition of run demands to find the first time instant when guards can enable updates. The mathematics behind the calculation of these instants is not complicated. We present it here to illustrate the questions of implementation. Consider the case of instantaneous actions. Recall that the most complicated atomic formula of guards is linear inequality with one occurrence of CT.

Suppose that we are simulating a given ASM on a finite interval of time, so the last instant of simulation is given. Assume for simplicity that guards are conjunctions of atomic formulas.

We advance by induction. Let t be the last defined instant where guards are true, or $t = 0$ if we are starting. The inputs are given, thus the minimal time instant $T > t$ such that the inputs remain unchanged, is known. The values of all predicates and formulas for inputs are known for $\tau \in (t, T]$. Replace the predicates in the guards by their values, eliminate the false guards, and eliminate the value *true*. Replace CT by a time variable, say, τ. Now the guards are reduced to systems of linear inequalities with one time variable τ. For each guard resolve the inequalities with respect to τ. This gives a system of inequalities of the form $\tau \omega c_j$, where c_j is a rational constant and ω is an inequality sign (we assume for simplicity that equalities are eliminated). Look for the *minimal* value of $\tau \in (t, T]$ which satisfies the inequalities. The following cases may appear for a given guard:

1. There is no such instant. Then replace t by T if T is not the last instant to consider and repeat the procedure. If T is the last instant given for the simulation then stop.

2. The minimal instant is not defined though there are instants satisfying the guard. This means that we arrive at an inequality $\tau < c$ or $\tau \leq c$ or $\tau > c$. The run is not defined after t in the case of $\tau < c$ or $\tau \leq c$. In the case of $\tau > c$ we may extend the run up to instant c and then stop.

3. The minimal instant is defined by an inequality of the form $\tau \geq c$ or by a pair $\tau \geq c$ and $\tau \leq c$. Then we have an instant where G is true. (The latter case, when in fact $\tau = c$, gives an instable, non implementable program, as in practice it is impossible to catch exactly a given time instant; we mean here that the real

program gets the time by reading system clock whose read values may miss the time instant in the guard. It is reasonable to detect such situations and to warn the user.)

Among all found instants for different guards take the first one. It may happen then that several guards should be processed simultaneously.

Consider the same ASM but this time with delays. Let t and T be the time instants as above. Take any guard. Suppose that the delays are given. Among the used values is the value of the current time that should be put in the inequalities. If the time is got with a delay δ_{CT} then at instant τ the calculations use the value $(\tau - \delta_{CT})$ that should be, however, greater than t. Let Δ be the maximum of delays of readings. The machine starts the calculation of the truth value of the guard at $(t + \Delta)$, and finishes it at some $(t + \Delta + \Delta')$, where Δ' is the total delay of the calculation of the value. Thus the calculation may go on by discrete steps.

3.2 Simulation (generation of runs)

The current version of the simulator deals with ASM that are more general that the basic ones – see section 3.3. Two types of external loops may be specified: `while...do` and `foreach...in...do`. `if-then`-construction can be nested, as well as sequential composition and parallel composition. Inputs are piecewise constant.

The semantics is the submachine one, see the end of section 2.2.2. The delays are represented in an appropriate configuration file that is reusable. They are defined for functions and constructors, in particular, the delay of update `:=` is always positive. The simulator permits to define, in particular, the following delays:

– Two dedicated delays δ_{ext} (reading of an external function) and δ_{int} (reading of an internal function). They may be set with $\delta_{ext} > 0$, $\delta_{int} > 0$.

– Another way to set the mentioned delays: $\delta_{ext} > 0$, $\delta_{int} = 0$, that is the operations with instantaneous read and write for internal functions.

– Instantaneous actions: $\delta_{ext} = \delta_{int} = 0$.

– Define all delays manually (the simulator uses `d` as function defining delays), for example,

```
d(+)=1; d(=)=2; dext=3; dint=1;
```

Here the numbers give fixed delays, but one can define lists.

The non-determinism is resolved along the following lines. As the simulator is destined for verification, the choice of delays follows usual rules of testing: we take 'extreme' cases and 'typical' cases (though it is not at all simple to describe these cases in practice). The choices of delays may be ordered. To configure the choice of delays, the simulator uses a key word `ndr` (`n`on-`d`eterminism `r`esolution). The following definitions are possible:

– `ndr=first` or `ndr=last`. Take the first or respectively the last element of the list defining possible delays.

– `ndr=min` or `ndr=max`. Take the minimal or respectively the maximal element of the list.

– `ndr=seq start <x> step <y>`. A changeable delay determined by its index starting from `<x>` then going to `<x+y>` modulo the number of elements in the list defining the delays. For example, for the list `(5, 1, 3, 2, 9, 6, 4)` and `<x>=1`, `<y>=3` we get the sequence `(5, 2, 4, 3, 6, 1, 9...)`.

– `ndr=random`. Take an element with random index.

The verification is done by reducing the property formula for a given finite interpretation to a propositional formula, see 2.3.1.3.

We illustrate the work of the simulator by an example in section 3.3.

3.3 Token Example

Here is an example of an ASM specification and its processing by the simulator. The external functions are `Pass`, `d1`, `d2`, and `a`. The `Pass` function defines the next process that should get the token if some time constraints are satisfied. If this is the case, the token is given to the process defined by `Pass`. There are 3 processes: 1, 2, 3.

Here is a specification.

```
// Parameterized token processing
// Created by Pavel Vasilyev, 01/01/07

type ProcessNo = {1..3};
type Proc = ProcessNo -> Boolean;
function Token: Proc;
function Last: Float;

// external (monitored) functions
function Pass: ProcessNo;
function a, d1, d2: Float;

Main() {
 [ Last := 0;
Token(1) := true;
Token(2) := false;
Token(3) := false;
 ]   // initial state definition

 while ( CT <= 6 ) do [
```

```
if ( Token(1) and (Pass != 1) ) then
[ if ( (a*Last+d1 <= CT) and (CT <= a*Last+d2) )
 then [ Token(Pass):=true; Token(1):=false; ]
  Last := CT;
]

if ( Token(2) and (Pass != 2) ) then
[ if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
 then [ Token(Pass):=true; Token(2):=false; ]
  Last := CT;
]

if ( Token(3) and (Pass != 3) ) then
[ if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
 then [ Token(Pass):=true; Token(3):=false; ]
  Last := CT;
]
 ]
}
```

The file of external functions configuration:

```
Pass := (0, 1; 1, 3; 2, 1; 3, 2; 4, 1; 5, 3)
d1   := (0, 0.3; 1, 0.7; 2, 0.2; 3, 0.4; 4, 0.1)
d2   := (0, 1.2; 1, 1.3; 2, 1.5; 3, 1.4; 4, 1.7)
a    := (0, 1; 2, 1.1; 4, 1.2; 5, 0.7)
```

The file of requirements specification:

```
Liveness: forall t in Time holds  exists p in ProcessNo
where  ( Token‘(p, t) )

Safety:   forall t in Time holds  forall p, q in ProcessNo
holds  ( p = q or not Token‘(p, t) or not Token‘(q, t) )
```

Consider the execution of the program by the simulator:

```
TASML Preprocessor:  Reading from file samples/tokenpar.asm
D:\Pavel\Aspirant\Projects\ASM_Simulator\samples\tokenpar.asm
TASML Parser Version 0.1:  ASM specification parsed
```

```
successfully.
Reading functions definitions...
Function: Pass is now external.
Function: d1 is now external.
Function: d2 is now external.
Function: a is now external.

Printing all method names
Main

Printing all type names and their descriptions
Type Float: Float
Type Time: Float
Type ProcessNo: Integer [1, 2, 3]
Type Integer: Integer
Type Boolean: Boolean
Type Proc: ProcessNo, Boolean

Printing all function names and their values
d1[]: Float = {0.0=0.3, 1.0=0.7, 2.0=0.2, 3.0=0.4, 4.0=0.1}
d2[]: Float = {0.0=1.2, 1.0=1.3, 2.0=1.5, 3.0=1.4, 4.0=1.7}
Last: Float = undef.
CT: Time = undef.
a[]: Float = {0.0=1.0, 2.0=1.1, 4.0=1.2, 5.0=0.7}
Token: Proc = undef.
Pass[]: ProcessNo = {0.0=1.0, 1.0=3.0, 2.0=1.0, 3.0=2.0,
4.0=1.0, 5.0=3.0}
```

In the part of the output file above one sees an information about the processing of the specification text, loading of the external functions, and then there go lists of named blocks of rules and of user type definitions. At the end the initial state is reproduced.

Now look at the simulation process:

```
------------------------------ Simulation started
0.0: Visiting Main method
0.0: Visiting seqential block
0.0: Visiting parallel block
0.0: Last[] := 0
0.0: Token[1] := true
0.0: Token[2] := false
```

```
0.0: Token[3] := false
0.4: Visiting PAR WHILE loop
0.4: --- loop cycle #1
------------------------------ Time jump from 0.4 to 1.0
1.0: --- loop cycle #2
1.0: Visiting parallel block
1.0: Visiting IF-THEN-ELSE: Proceed to then/elseif statement
1.0: Visiting parallel block
1.0: Token[3] := true
1.0: Token[1] := false
1.0: Last[] := 1.0
1.4: --- loop cycle #3
...
------------------------------ Simulation stopped
```

In the text just above one sees that at the initial time instant the values of `Last`, `Token(1)`, `Token(2)`, `Token(3)` are changed (the delay of update is 0.4)). In this case we get the initial state though this information could be put outside *Main*. After the initialization the simulator enters the loop with a parallel block of rules. During the first pass of the loop body nothing happens as all the guards are false. At this instant a 'prediction' mechanism is switched on, and it says that there is a change of an external function only at the instant 1.0, and thus, the process can go on. Hence the simulator advances time skipping the interval $[0.4, 1.0]$, and the execution of the program continues. During the second pass of the loop body one of the guards is true, and the token is attributed to process 3. And so on. At the end the program arrives at a state where the external functions do not change their values, and the simulator halts.

Here is the final part of the result of the simulation:

```
Printing all function names and their values
d1[]: Float = {0.0=0.3, 1.0=0.7, 2.0=0.2, 3.0=0.4, 4.0=0.1}
d2[]: Float = {0.0=1.2, 1.0=1.3, 2.0=1.5, 3.0=1.4, 4.0=1.7}
Last[]: Float = {0.0=0.0, 1.0=1.0, 2.0=2.0, 3.0=3.0, 4.0=4.0,
5.0=5.0, 5.4=5.4, 5.8=5.8}
t: Time = undef.
q: ProcessNo = undef.
CT: Time = undef.
p: ProcessNo = undef.
a[]: Float = {0.0=1.0, 2.0=1.1, 4.0=1.2, 5.0=0.7}
Token[3]: Proc = {0.0=false, 1.0=true, 2.0=false}
Token[1]: Proc = {0.0=true, 1.0=false, 2.0=true,
```

```
3.0=false, 4.0=true}
Token[2]: Proc = {0.0=false, 3.0=true, 4.0=false}
Pass[]: ProcessNo = {0.0=1.0, 1.0=3.0, 2.0=1.0, 3.0=2.0,
4.0=1.0, 5.0=3.0}

Starting verification.
Converting property #1 : Liveness
Parameters: t (5), 5 permutations generated.

Converting property #2 : Safety
Parameters: t (5), 5 permutations generated.
```

In this final part there is given the final state and the results of the verification of `Liveness` and `Safety`. These properties are formulated in terms of `Token(,)` only. We see that the interpretation changes only at points {0, 1, 2, 3, 4}, so five time intervals are used for processing the formulas.

4 Conclusion

Basic ASMs may serve for a specification of relatively small controllers, however with very non-trivial real-time constraints that makes their verification a hard problem. The ASMs treated by the simulator are more expressive but still rather limited. In order to advance the simulator towards more complicated problems it should be applicable to ASMs with reacher syntax, including rule declaration, import, multi-agent construction. More general semantics are worth to be treated. Such an extension of the simulator is not an easy task.

More feasible and very useful improvements concern the user interface, as from the viewpoint of representation of the displayed information, as well as from the viewpoint of user-friendly tools of inputs generation, description of delays and specification of non-determinism resolution. All this can be done in terms of simple ASMs that can be interpreted by the same simulator. We illustrate this for the generation of inputs.

Consider piecewise linear input functions. Let $f : \mathcal{X} \to \mathcal{Y}$ be such a function, where $\mathcal{X}$ is a finite sort, and $\mathcal{Y}$ is a finite sort or pre-interpreted sort $\mathcal{R}$ of real numbers. $\mathcal{X}$ can be represented as a finite initial segment of natural numbers $\{0, 1, \ldots, l\}$. As $f(i)$ is piecewise linear then for each given i, $0 \leq i \leq l$, the values of $f(i)$ can be represented by a partition of $\mathbb{T}$ into intervals and by a term of type $\mathcal{Y}$ on each such interval. Suppose we wish to define $f(i)$ on a finite time interval $[0, T]$. Its partition is defined by a finite number of points $t_0^i = 0 < t_1^i < \cdots < t_k^i$.

For each interval $[t^i_{k-1}, t^i_k)$ we give a value f^i_k of type $\mathcal{Y}$:

$f(i) = (t^i_1, f^i_1; t^i_2, f^i_2; \ldots; t^i_k, f^i_k)$.

In the simulator each f^i_k can be defined by a linear expression using 2 special variables: t_a for absolute time and t_r for 'relative' time from $[0, t^i_k - t^i_{k-1})$. For example, formula $f^i_3 = 2t_r + 3.5$ defines a function $\varphi(t) = 2(t - t^i_2) + 3.5$ for $t \in [t^i_2, t^i_3)$.

What we described above is a straightforward list representation of inputs (that is implemented in the current version of the simulator). In practice we wish to quickly generate many various inputs.

One possibility is to generate inputs by an ASM with instantaneous actions, discrete time and without input. The form of such ASM should be simple.

For example, for each input function f as just above, in order to define $f(i)$, we introduce a natural number k_i that gives the number of intervals on which $f(i)$ is defined. The program should define a partition of time by points t^i_k for $1 \le k \le k_i$, and values f^i_k for each interval $\left[t^i_{k-1}, t^i_k\right)$. Here is an example of such an ASM ,where {...;...} denotes sequential composition (the notations are informal but self-explanatory) :

foreach i **in** $\mathcal{X}$ **do**
{$f^i_1 := 2t_r + 3.5$;
foreach $k := 1$ **to** k_i **step** 1
{**if** $k > 1 \wedge k \bmod 3 = 2$ **then** $[t^i_k := t^i_{k-1} + 2,\ \ f^i_k := 3]$;
if $k > 1 \wedge k \bmod 3 = 0$ **then** $[t^i_k := t^i_{k-1} + 0.8(t^i_{k-1} - t^i_{k-2}),\ \ f^i_k := f^i_{k-1}]$;
if $k > 1 \wedge k \bmod 3 = 1$ **then** $[t^i_k := t^i_{k-1} + 0.5 \cdot t^i_{k-2},\ \ f^i_k := t_a - 2]$
}}

However, such an ASM is enough general to have bugs that are not easily visible. It may give meaningless results, like not increasing instants that should define a partition of time, non admissible expressions for functions, incorrect recursive equations etc. So the question is how to help the user to detect such errors.

A solution is to limit ourselves by particular programs and to impose easily verifiable properties of their functioning. Consider as an example the programs of the following type. A program contains one **foreach** external constructor that quantifiers only arguments of the input we define. Next constructors are sequential compositions of updates or loops. Each loop is explicitly bounded and has as its body a sequential composition of **if** *Guard* **then** *ParallelBlockOfUpdates*. The example above is a program of this type.

In such a program we have an almost explicit consecutive numbering of intervals and terms defining the function in the appropriate interval. Thus, after each passage of the loop body we can check the mentioned properties and detect errors.

One more necessary option is random choices of values from given intervals that can be applied for partition definition and as well to function definition.

In order to specify the delays (in a more general way than in the current version of the simulator) by an ASM of some simple type like the one sketched above, one needs more primitives. In addition to those used in the previous example, the following primitives are obviously useful:

• Primitives that permit to easily refer to occurrences of program nodes. They may be labels or names in some standard, automatically producible naming of program nodes that the user can see.

• Terms to calculate delays. Any term in the vocabulary of the ASM is a priory admissible.

An ASM that specifies the delays calculations may use usual guards. All this should permit to express statements like "if CT is in an interval σ and $f < g$ then the delay of reading x is the nth value in the list L", where f, g and x are functions of the simulated ASM, and L is the list of numbers declared in the ASM that calculates delays.

The user may write an ASM that produces meaningless delays, for example, delays that are not in the specified intervals. So some verification is useful. For example, one can declare simple global constraints that the simulator checks for each generated delay.

References

[ASMGofer] ASM Gofer. http://www.tydo.de/AsmGofer/.

[CoreASM] The CoreASM project. http://www.coreasm.org/.

[Michigan ASM] University of Michigan, ASM homepage. http://www.eecs.umich.edu/gasm/.

[XASM] XASM project. http://www.xasm.org/.

[MicrosoftAsmL, 2002] (2002). *AsmL: The Abstract State Machine Language*. Foundations of Software Engineering, Microsoft Research, Microsoft Corporation. http://research.microsoft.com/fse/asml/.

[Beauquier et al., 2003] Beauquier, D., Crolard, T., Prokofieva, E. (2003). Automatic verification of real time systems: A case study. In *Third Workshop on Automated Verification of Critical Systems (AVoCS'2003)*, pages 98–108. University of Southampton.

[Beauquier et al.,2004] Beauquier, D., Crolard, T., Prokofieva, E. (2004). Automatic parametric verification of a root contention protocol based on abstract state machines and first order timed logic. In Jensen, K. and Podelski, A., editors, *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*, pages 372–387. Springer-Verlag Heidelberg.

[Beauquier et al., 2000] Beauquier, D., Crolard, T., Slissenko, A. (2000). A predicate logic framework for mechanical verification of real-time Gurevich Abstract State Machines: A case study with PVS. Technical Report 00–25, University Paris 12, Department of Informatics. Available at http://www.univ-paris12.fr/lacl/.

[Beauquier et al., 2002] Beauquier, D., Slissenko, A. (2002). A first order logic for specification of timed algorithms: Basic properties and a decidable class. *Annals of Pure and Aplied Logic*, 113(1–3):13–52.

[Beauquier et al., 2006] Beauquier, D., Slissenko, A. (2006). Periodicity based decidable classes in a first order timed logic. *Annals of Pure and Applied Logic*, 139(1–3):43–73.

[Börger et al., 2003] Börger, E., Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

[Börger et al., 1995] Börger, E. Gurevich, Y., Rosenzweig, D. (1995). The bakery algorithm: yet another specification and verification. In Börger, E., editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press.

[Cohen et al., 2000] Cohen, J., Slissenko, A. (2000). On verification of refinements of timed distributed algorithms. In Gurevich, Y., Kutter, P., Odersky, M., and Thiele, L., editors, *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*, pages 34–49. Springer-Verlag.

[Cohen et al., 2008] Cohen, J., Slissenko, A. (2008). Implementation of sturdy real-time abstract state machines by machines with delays. Technical Report TR-LACL-2008–02, University Paris 12, Laboratory for Algorithmics, Complexity and Logic (LACL). Submitted. Available at http://www.univ-paris12.fr/lacl/.

[Graf et al., 2007] Graf, S., Prinz, A. (2007). Time in abstract state machines. *Fundamenta Informaticae*, 77(1–2):143–174.

[Gurevich, 1995] Gurevich, Y. (1995). Evolving algebra 1993: Lipari guide. In Börger, E., editor, *Specification and Validation Methods*, pages 9–93. Oxford University Press.

[Gurevich, 2000] Gurevich, Y. (2000). Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111.

[Gurevich et al., 1996] Gurevich, Y., Huggins, J. (1996). The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In Buening, H. K., editor, *Computer Science Logics, Selected papers from CSL'95*, pages 266–290. Springer-Verlag. Lect. Notes in Comput. Sci., vol. 1092.

[Halpern, 1991] Halpern, J. (1991). Presburger arithmetic with unary predicates is π^1_1-complete. *J. of symbolic Logic*, 56:637–642.

[Ouimet et al., 2008] Ouimet, M., Lundqvist, K. (2008). The timed abstract state machine language: Abstract state machines for real-time system engineering. This issue.

[Schellhorn et al., 2006] Schellhorn, G., Grandy, H., Haneberg, D., Reif, W. (2006). The mondex challenge: Machine checked proofs for an electronic purse. In *Proc. Of Symposium on Formal Methods (FM'2006). Lecture Notes in Computer Science*, volume 4085, pages 16–31. Springer-Verlag.

[Slissenko, 2004] Slissenko, A. (2004). A logic framework for verification of timed algorithms. *Fundamenta Informaticae*, 62(1):29–67.

[Soloviev et al., 2003] Soloviev, I., Usov, A. (2003). The language of interpreter of distributed abstract state machines. *Tools for Mathematical Modeling. Mathematical Research.*, 10:161–170.

[Sommerville, 1992] Sommerville, I. (1992). *Software Engineering*. Addison-Wesley, 4th edition.

[Vasilyev, 2006] Vasilyev, P. (2006). Simulator for real-time abstract state machines. In Asarin, E. and Bouyer, P., editors, *FORMATS, Paris*, volume 4202 of *Lecture Notes in Computer Science*, pages 337–351. Springer.

[Vasilyev, 2007] Vasilyev, P. (2007). Simulator-model checker for reactive real-time abstract state machines. In *Proceedings of the ASM'07 the 14th International ASM Workshop, Grimstad, Norway*, pages 128–140.

Appendix: Architecture of the Simulator

A detailed description of the syntax and semantics of the language used by the simulator is in P.Vasilyev PhD Thesis (in Russian). A version of the simulator is available at http://vpax.narod.ru/asmsim.html. Here are main features of the simulator architecture.

The simulator is a platform independent tool implemented in Java. It consists of a kernel and of a graphical user interface. The kernel accomplishes all the tasks of processing the user information and of the result representation. The user information and the result are in text format. The kernel consists of:

- Parser of timed ASM.
- Parser of FOTL formulas.
- Loader of execution parameters.
- Trace repository.
- Interpreter of timed ASM.
- Verifier of the timed ASM properties.

Simulator kernel. The kernel of the simulator is a complete Java-application that contains all necessary for processing ASM specifications and checking the properties. As an input it gets user specifications, namely an ASM specification and a requirements specification as FOTL formulas. The configuration file contains setup data: inputs, delays, resolution of non-determinism and some other. The results of simulation are saved in the trace repository. These results are available for the user or for other tools like verifier or a tool for graphical representation of results.

The kernel architecture also permits to encapsulate its copy in program shell and redirect input-output streams. Thus one can use the simulator as an individual application or as a part of a graphical shell of a tool for formal specifications processing. On having been launched, the simulator checks whether an external container with graphical shell is available, and then it proceeds depending on the type of launching.

Parser of the input language. There is a parser that analyzes the input ASM specification as well as constraints. The parser is produced with the help of JavaCC version 4.0 and its pre-processor JJTree. The full grammar is in P. Vasilyev PhD Thesis.

The parser builds an abstract syntactic tree, fills the vocabularies of the trace repository, builds a list of function symbols with their types. This phase builds also a vocabulary of user data types and initialization data. Each node of the syntactic tree keeps the following information about the respective syntactic element: type, contents, delay and access modifiers.

Blocks of updates can be named (to be used later as sub-machines) and gathered in a special list inside the specification file with references to the points

of their definition. By default the program is executed starting with a block `Main`.

Loader of parameters of execution. The user can define some parameters of the ASM execution such that delays of operation execution, resolution of inconsistency of updates, resolution of non-determinism. Delays are defined as pairs each consisting of the name of a syntactic construction (e.g., arithmetical operation, read, write) and of the value of delay ascribed to this construction. If external abstract functions are used, then their definitions are loaded from a special file at the beginning.

Trace repository. This repository plays the role of storage of runs of the ASM that is being simulated. The trace of a function is a mapping from $\mathbb{T}$ to an interpretation of the function. After the end of simulation each function is defined on time interval from 0 to the final given time instant. All the changes of the function values are stored in the repository.

Interpreter of timed ASM. The interpreter executes the ASM program represented a syntactic tree. First the initialization is done. Then the interpreter repeats the steps of execution. A step consists of:

- Determining the type of the next syntactic construction or operation.
- Executing the operation that corresponds to current processed syntactic element.
- If the operation presumes updates then the consistency of updates is checked.
- If the current syntactic element has embedded elements then the control pass recursively to these elements.
- The delay is calculated according to the rules of its calculation for composed operations if the operation is not simple.
- In the case of non-zero delay the system time is respectively updated.

The step of operation execution (the second item above) depends on the construction under treatment: computing a numerical value, update (assignment), guarded rule, **foreach**-rule, sequential composition, parallel composition, loop with parallel composition, loop with sequential composition.

Verifier. The verification is a model-checking of the produced model.

Graphical interface. The graphical interface is implemented with the help of Java Swing as a Java-applet that allows to work with specifications either locally or via internet.