Time in State Machines

Susanne Graf
Verimag
2, avenue de Vignate
Grenoble, France
Susanne.Graf@imag.fr

Andreas Prinz
Agder University College
Grooseveien 36
Grimstad, Norway
Andreas.Prinz@HIA.no

Abstract. State machines are considered a very general means of expressing computations in an implementation-independent way. There are also ways to extend the general state machine framework with distribution aspects. However, there is still no agreement when it comes to handling time in this framework. In this article we take a look at existing ways to enhance state machine frameworks. Based on this we propose a general framework of time extensions for state machines, which we relate to existing approaches. Our work is mainly based on time approaches for ASM, because ASM are considered a very general state machine model. Taking this into account, our approach is valid for state-transition systems in general.

1 Introduction

State machines are considered a very general means of expressing computations. This is used in the formalism of Abstract State Machines (ASM) [1] that are considered appropriate for giving semantics of a system in terms of its set of possible executions. ASM have been used to define a formal semantics for programming languages (e.g. C [15] and Java [16]). It has also been used to define the formal semantics of SDL 2000 [17], the most recent version of SDL [3]. SDL includes an explicit notion of time and time progress, whereas ASM, at least in their initial version, have no explicit means to deal with real-time. Being a very general formalism, it is however possible to express any notion of time. For the definition of the SDL semantics in ASM, we used the ASM time semantics as proposed in [2] and encountered some problems. The SDL view on time progress is slightly different from the one formalized in [2]. Moreover, ideas to extend the SDL timing to include models with continuous changes lead to a more thorough examination of this issue. This motivated us to summarize some essential properties of timed computations and to derive a methodology for dealing with time in ASM, which we present in this paper. We have studied the main models for timed computations used outside the domain of ASM and we have considered the existing time extensions of ASM. Our aim is to provide a framework for defining timed computations in ASM. For this purpose, we define also a set of generic properties of timed computations, which may be used to restrict the set of “well-formed timed computations”.

We tackle the problem by considering the needs for modeling the semantics of concurrent and possibly distributed systems as in the context of model based verification and testing. We want to provide a means to describe executions of timed systems using ASM in a convenient manner. In particular, we do not want to restrict ourselves to the description of time dependent behavior – as in real-time programming languages – but we want to describe a set of timed computations which means also to express constraints on time progress with respect to system progress. In other words, we do know that some computations take some amount of time and want to specify this.

In section 2, we discuss how to introduce time into the domain of ASM. Moreover, we give some ideas how to express timed properties. Section 3 provides a set of time properties that can be chosen in addition to the basic properties of section 2. Section 4 discusses the relation of our approach to the existing approaches. In Section 5, we collect the properties we have given to form one possible use case for ASM. The presentation is closed by some conclusions.

2 An Overview on ASM and Timed Systems

In this section, we discuss general problems occurring when introducing time in a computation model, and then consider the particular case of ASM. Also, our aim is not just to be able to describe a set of timed computations, but to get a means to construct this set of computations (executability). We give an overview on the principles of ASM and discuss some general problems of timed systems.

2.1 An overview of ASM and their computation model

Abstract State Machines (ASM) [1] were introduced as a general computation model taking concurrency into account explicitly. They are defined starting with a sequential variant and going on to the general distributed case.

A *sequential* ASM algorithm is described by a set of rules applied to states. The notion of state in ASM is very general[1], but for the discussion of time extensions one can consider without loss of generality that a state is defined by the values of a set of variables (also called *locations*). The notion of "atomic step" (called a *move*) is defined by means of a (arbitrarily complex) rule for assigning a new value to some variables depending on the old state values. In order to model inputs and non determinism, a part of the state space might consist of "monitored variables" which are under the control of "the environment" and the values of which can only be read by "the system". The possible evolutions of the values of monitored variables can be explicitly restricted by *constraints*. That means, the behavior of the environment may be described in a declarative way, whereas the system description is provided in a constructive way: for any state, a means to construct the (set of) next states is given

1 A state is an algebra and a rule describes a transformation between algebras.

by the set of possible moves. The semantics of the system is given as the set of possible *executions*, i.e. countable sequences of states representing possible evolutions from the initial state according to the rules and the constraints on monitored variables.

In a *distributed ASM*, there exist several agents (the number of which may evolve over time), which can read and write some part of the global state and have their own update rules. The semantics of distributed ASMs is given by a set of *partially ordered runs*, where each of these runs consists of a set of moves, partially ordered by a relation denoted <. Absence of order between moves expresses independence. In addition, it is required that in any run the moves of each agent are strictly ordered and that rule application is confluent (coherence condition). The set of executions is the set of state sequences induced by all possible linearizations of all runs.

In ASM, there exists no predefined notion of *trigger*. Each agent has "its rule" which can be applied to any state independently of any other agent or the changes of the environment. However, sometimes the application of the rule does not change the state (empty update set). We will consider these rules to be *guarded rules*, representing partial functions from global states to global states, and consider only runs without such empty moves.

Based on this general setup in ASM, we consider a system description to consist of (a) a description of the initial state (defined as the valuation of a vector of variables), and (b) a description of the state change rules. Rules are described per agent. A rule defines a set of moves ($m \in M$). A run is a subset of moves ($R \subseteq M$) satisfying the above mentioned constraints, where each move belongs to an agent.

Definition: A *prefix F* of a run *R* is a subset of moves closed for <, that is,

$$\forall m \in F \; \forall m' \in R \bullet m' < m \Rightarrow m' \in F$$

2.2 Adding Time to State Transition Systems

The first decision we have to take when adding time concerns the kind of models we consider. There is the general distinction between state-based and event-based models. As we are mainly interested in ASM, we will focus in state-based models. However, the following statements are very general and can probably also be extended to other kinds of models, which we do not do here.

Another preliminary consideration concerns domains. We suppose the existence of two related possibly dense domains *Time* and *Duration* with appropriate operations between them (notice that axiomatizations of these domains have been given in earlier work, for an example see [11]).

In state-based systems, runs are characterized by states and moves, which are alternating. The next decision is therefore where to attach the time: to states, or to moves, or to both. Moreover, we can also attach time intervals to each of states and moves. This gives us three alternatives: (a) time points for moves, time intervals for states; (b) time points for states, time intervals for moves; (c) time intervals for moves and states.

If time intervals are attached to moves (time is allowed to proceed), we are faced with durative moves, and otherwise we have instantaneous moves. We take the view, that moves are describing a change of the state, and that we can associate a *time point* with such a state change. As we are handling distributed computations, it is essential to have a well defined (partial) state at any time. This boils down to excluding durative moves, because then there is clearly no state defined during the time the move is performed. If durative moves have to be modeled they should be modeled with an explicit intermediate state (and there is space for different frameworks with different choices). This means the only remaining alternative is (a), where time points are attached to moves and time intervals to states.

We might also want to start from the other end, looking at states first. If we add time to states, it turns out that this alternative requires for compositionality and for distributed agents a maximal granularity of time steps (when there is no a priori existing discretization, time steps need to be dense). This is done in a very consequent way within the formalism of HASM. However, we think that this is too complex to be useful for practical applications. Therefore this rules out the alternative (b) in favor of alternative (a), which leads to a simpler framework. For reasons of simplicity, (a) is then also better than (c).

This brings us to a first property of timed computations: *time points are attached to moves*. In order to achieve this in a sound way, we have to extend our semantic model. We start with the partially ordered runs of the original ASM definition. Attaching time to moves is relatively simple by introducing a function `when` as defined below.

Property 1: A state change takes place at a *point of time*, i.e. we introduce a function `when` providing the time point of a move:

$$\texttt{when}: M \rightarrow \mathit{Time}$$

This means that time is not part of the state, but there is a semantic function associating a time point with moves; the duration of a state can be defined implicitly as the time passing between the occurrence times of its two adjacent events (if this interval is considered to be open, left-closed, right closed or closed leads to different frameworks). The next property concerns the relationship between the time-induced order and the partial order of moves. A natural minimal requirement is given by

Property 2: In every run, all causally ordered events are also ordered in time, i.e.

$$\forall\, m_1, m_2 \in M \bullet m_1 < m_2 \Rightarrow \texttt{when}(m_1) \leq \texttt{when}(m_2).$$

More restrictive requirements on the time ordering are discussed in section 3.

3 Choice of a Timed Framework by a Set of Properties

So far, we have shown how to extend an ASM system with time semantically in the form of timed runs and have given one minimal property of the resulting timed

computations. Most frameworks for modeling timed systems suggest that a timed computation should have some additional properties. Such constraints are often either directly reflected in syntactic or semantic restrictions of the framework (for example, in VHDL the fact that causally related events must have a time distance of at least some infinitesimal δ) or imposed as verification conditions in the form of assumptions (this is in particular the case for fairness or non Zenoness constraints). This means that these properties represent axioms of particular timing frameworks which in the context of ASM can be added as constraints.

In this section, we discuss a number of such constraints, which we have found in existing frameworks. For each property, we try to evaluate if it is executable or not, i.e. if it is possible to extend an executable framework without the axiom into one including it. In the case of distributed ASM and time the most important question of executability is about which agent is to be executed next. Without timing, this question is easily answered in that any minimal[2] move can be taken next. When time constraints between arbitrary events are allowed, executability is not guaranteed anymore: as to obtain a consistent timing of all moves in some run, the set of possible timepoints of some move m, may a priori not be determined by only looking at the prefix of the run.

3.1 Global and Local Time and additional Ordering Constraints

In most existing frameworks time is global, that is the time domain is totally ordered[3]. As a consequence, any two timed events are either temporally ordered or simultaneous, which decreases the power of partially ordered runs. A notion of local time allows to introduce a partially ordered time domain where some time points may be *incomparable*, that is neither ordered nor simultaneous. This distinction motivates some variants of property 2 defined in Section 2.2.

Property 2 states that in a timed run the time order is not smaller than the causal order. In order to define global time, this restriction must be strengthened.

Property 2a: *Global Time*: all time stamps of a run are totally ordered, i.e. the partial order of the moves is extended to a total preorder for their occurrence times, i.e.

$$\forall m_1,m_2 \in M \bullet \text{when}(m_1) \leq \text{when}(m_2) \text{ or } \text{when}(m_2) \leq \text{when}(m_1)$$

Independently of the choice of local or global time, some frameworks impose an even stronger constraint on causally ordered events.

Property 2b: *Strict time progress along causal chains*: whenever two moves are causally ordered, their occurrence times are *strictly* ordered, i.e.

$$\forall m_1,m_2 \in M \bullet m_1 < m_2 \Rightarrow \texttt{when}(m_1) < \texttt{when}(m_2).$$

This forbids reaction chains in zero time. When distinct agents are used to represent physical distribution, this property may make sense, but often distinct agents are used

2 With respect to the causal order

3 In general, real numbers are chosen representing a dense time domain.

to express purely descriptive parallelism in a compositional manner, and here reaction chains may express synchronization or just conjunction of constraints.

Some frameworks for distributed systems with a strict notion of local time require an even stronger property implying that time distances can only be measured between causally related moves.

Property 2c: Timed order is not stronger than causal order: causally non related events are not comparable in the timed order, i.e.

$$\forall\, m_1, m_2: M \bullet m_1 \leq m_2 \text{ iff } \texttt{when}(m_1) \leq \texttt{when}(m_2).$$

This requirement is very strong as it forbids even incidental timed ordering of non causally related moves. The only means to satisfy this axiom is by choosing a partially ordered time domain.

All these axioms represent safety properties which can be used to strengthen the constraint on the occurrence time of the immediate successors of each move. Note that some of these properties may be combined to create stronger constraints.

3.2 Zeno Computations

So far, we did not justify the choice of a dense time domain but just chose it to be dense. Indeed, a dense time domain allows arbitrary action refinement (making more internal steps visible) because between any two moves always an intermediate move at an intermediate time point can be inserted without redefining the time scale.

What is the meaning of density in an individual timed computation of countable length? For example, do we want to allow computations where the occurrence times of the events of a computation converge to some finite time point? This is called a Zeno computation, and in most frameworks considered as an *invalid* computation, as expressed by the following property.

Property 3a: Absence of Zeno computations: In any infinite run, there is no upper bound of the time values attached with moves, i.e.

$$\forall R\ \mathit{isInfinite}(R) \Rightarrow \forall t \in \mathit{Time}\ \exists m \in R \bullet \texttt{when}(m) > t.$$

Often, we need density only to say that we do not want to require any global bound on the minimal distance between two ordered events or to be able to make refinement easy. In any single computation, it makes sense to require the existence of some discrete duration δ such that by observing the state every δ time units, no local state change is missed[4].

Property 3b: Existence of a discretization: There is a lower bound of the time differences between non simultaneous causally ordered moves, i.e.

$$\exists \delta \in \mathit{Duration}\ \forall m_1, m_2 \in M \bullet m_1 < m_2 \Rightarrow \texttt{when}(m_2) - \texttt{when}(m_1) > \delta.$$

4 In this context, multiple state changes at the same time are considered as a single state change.

Property 3b is strictly stronger than 3a, as 3a admits computations where the events of some agent are at time points t_n, with $t_{n+1} = t_n + 1/n$. This computation is not Zeno, as t_n grows over all bounds, but it does not satisfy property 3b. Notice that in the context of global time, 3b does not exclude the existence of "time races", that is, moves in different agents which are arbitrarily close, so that there exists no uniform δ to separate any pair of moves by a discrete observation.

We can give an even sharper way of defining the time points of moves which defines time as discrete.

Property 3c: Events at discrete steps: Any two moves occur either at the same instant or the time differences between their occurrence times are a multiple of a given value,

$$\exists \delta \in Duration \forall m_1, m_2 \in M \; \exists k \in N \bullet \texttt{when}(m_1) - \texttt{when}(m_2) = k * \delta$$

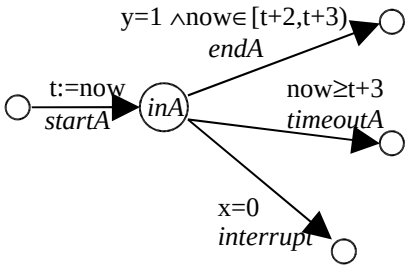
In the context of timed automata, sufficient syntactic conditions on rules have been given guaranteeing the absence of Zeno computations and verification methods exists for checking that all finite prefixes can be extended to non Zeno computations.

Concerning executability, properties 3a is meaningless for finite prefixes, property 3b coincides with 2b for finite prefixes, and property 3c is easily executed because it is a safety property[5].

3.3 Maximal Progress and Urgency

In general, a set of time constraints allows several alternative time stamps to be attached with a move. In this case, the question arises if one should implicitly choose a particular one or make the choice non deterministically. This brings us to the question if one wants to impose some *urgency*, for example by requiring maximal progress: Should transitions be taken at the *earliest possible* time point with respect to all the constraints?

Example: Let us consider the example in the figure below to illustrate the questions related to urgency. The Figure shows some agent *Ag* and its choices in state *inA* (meaning that action *A* has been started). There are 3 possible terminations (a move with three distinct futures), depending on *if and when* some other agents set the variables *x* and *y*; notice that we only represent guards here.



An interrupt transition is usually interpreted *urgent*: as soon as *x* is set to *0* by the environment, let's say a move m_I in an agent Ag_I, the interrupt or one of the alternatives follows in immediate reaction. A timed run in which the interrupt occurs not "*immediately after*" the

5 A safety property is a property that can be checked on all finite prefixes of runs, or in the setting of temporal logic, in the form of "always (some property on the past)"

enabling move, but later, is to be considered *invalid*[6]. For the other two transitions several interpretations make sense:

- An "*urgent*" interpretation of the *endA* transition means, it occurs "as soon as the environment sets *y* to *1*, let's say a move m_N in an agent Ag_N, but only between *2* and *3* time units after entering the state". In particular, this means that a run in which a move m_N occurs, and some time later a move m_I and then the *interrupt* move, is **not** allowed.
- A "*delayable*" interpretation of the *endA* transition means it occurs "at any moment between *2* and *3* time units after entering the state, but only under the condition that m_N has occurred before". This means in particular, that even when m_N occurred, the *interrupt* may be chosen but only before *3* time units have passed in state *inA*, whereas the *timeout* is impossible.
- Finally the "*lazy*" interpretation of the *endA* transition means, "if m_N occurs in the right time interval, *endA* or alternatively *timeout* may be the form of the next move of agent *Ag"*.

In order to achieve the "lazy" interpretation of all transitions, no additional axiom is needed. Notice that – even if this is sometimes quite cumbersome - the "urgent" interpretation of *all* transitions allows achieving the other interpretations by the introduction of explicit moves delaying the concerned move by a duration *d* in some interval when the environment makes true its enabling condition[7].

Urgency is expressed simply by the following property:

Property 4: Urgency: The time of each state change of each run is minimal.

$$\forall R \bullet \texttt{when} \text{ is minimal}$$

Several remarks can be made:

- First, urgency makes timing deterministic - the time point of each move is determined as the earliest point of time at which it becomes enabled and satisfies its time guard. Time non determinism can only be introduced by explicit choices in moves of the form "choose to wait *d*"[8] and guarantees therefore executability under time constraints.

6 The interpretation of "immediately after" depends on the choice of the time progress model. If property 2b is not considered, "immediately after" means in fact "at the same point of time".

7 In the example, *endA* is made *lazy* by splitting it into to successive moves "*y=0 -> choose d∈[now,3)*" followed by "*now=d -> endA* or *x=0 -> interrupt*", that means a soon as m_N occurs, timeout becomes impossible, but the possibility of the occurrence of an interrupt is maintained

8 or alternatively "force another agent to wait *d*"

- Second, the choice of the time domain to be reals, makes property 4 incompatible with property 2b[9]; it also makes illegal strong time constraints for the form "*m* takes place at time *t>2*"

We could also try to not introduce Property 4, but rather to express delayable and urgent transitions by extra moves or conditions in moves. This is not possible, as "as early as possible" is a semantic level notation, not expressible at "program level".

In general, the introduction of time constraints can make some untimed runs impossible, and this risk is increased by the introduction of property 4[10]. This is sometimes considered as undesirable. From the example, it becomes clear that forbidding elimination of runes by timing is not compatible with defensive programming, which consists, among others, by addition of timeouts. We consider this a methodology issue as, a property saying that "for all untimed runs must exist a timed run" is not a constraint on the possible time points of the moves in each run, but a restriction on the possible ASM specifications.

4 Related Work

In this section, we discuss several frameworks for modeling of timed systems and relate them to the set of properties introduced in the preceding section.

4.1 Timed and Hybrid automata

Timed automata [9] are a model focusing exclusively on timing aspects. A system is represented as a graph where each transition representing a state change is labeled by a constraint on when it can take place. The semantics of a timed automaton is formally defined as a transition system on states defined by a pair consisting of a control state and a time point, where transitions are either discrete moves without state change or time progress moves in the same state. Executions as we have defined them in section 2 are obtained from a timed automaton execution by the stuttering reduction of the projection to a sequence of discrete states, where the time point at which each discrete move take place is used to define the function `when`.

In timed automata, clocks can be reset to zero in transitions; from then on, at any time they represent the duration since this transition or event until they are reset for the next time.

The time model of timed automata is that of global but relative time, where the absolute value of time may or may not be defined in any particular system. With

9 except if an adequate explicit timing is introduced explicitly to constrain the occurrence time of each reaction

10 in the example, when the m_N transition occurs necessarily before time point 3 in all timed runs, all those runs are impossible which depend on the choice of the *timeout* variant of the move of agent *Ag* in the example.

respect to the set of properties of section 3: timed automata impose only the minimal restriction on timed behaviors (property 2), that is, causal chains in zero time are possible. Only non Zeno computations are valid (property 3a). Timed automata do not require maximal progress (property 4), but have several means to express a less restrictive notion of urgency. *Invariants* associated with states can force the system to leave a state on a time condition (like the timeout transition in the example of section 3.4, but without forcing immediate reaction) or by an explicit *urgency attribute* associated with transitions, specifying when transitions *must* be taken.

Hybrid automata are a generalization of timed automata adding continuous entities other than time. Again, states represent "*modes*" in which continuous entities evolve with time according to certain laws – expressed in terms of differential equations – whereas state changes represent discrete changes between different modes - in which the continuous entities may evolve according to different laws. The above mentioned variants of timed automata can also be defined for hybrid automata.

4.2 Timed Petrinets and Time Extensions in Process Algebras

The aim of time extensions of Process algebras [19] and Petrinets is to express constraints on the occurrence times of synchronization events. These formalisms have Property 1. They are based on global time and impose only the minimal restriction on timed computations as formulated by Property 2. Properties of infinite sequences, such as non Zenoness are not part of the framework, but are often added at analysis time as verification conditions or hypotheses.

The timed process algebra E-Lotos [21] attaches interval waiting constraints with actions (similar as timed automata). The semantics is defined in a compositional manner: partial actions (which are able to synchronize with other subsystems) are *delayable*, whereas actions which cannot synchronize with other actions are urgent (property 4).

A Petrinet [20] defines a set of partially ordered runs, and time extensions express constraints on the occurrence times of synchronization events. Petrinet places are waiting states and "transitions" are actions which may take a variable amount of time. There are several variants of timed Petrinets. We mention here the most interesting one, preserving free choice: an enabled transition (all ingoing places have the required amount of tokens) take place after a variable amount of time defined by a constraint associated with this transition; nevertheless during this "blocking time" the choice between conflicting transitions remains open, even for those transitions which become enabled later, as long as they can take place not later than the "blocked" transition. That is synchronization transitions are delayable, In order to express a set of conflicting Petrinet synchronizations, this must be done in ASM either directly in a single agent or by some agreement protocol amongst individual agents or agents representing the transitions.

4.3 Time Extensions of ASM

In the ASM framework, the notion of state is central, whereas in the definition of partially ordered runs, events (moves) play the central role. The time extensions for ASM consider time as a part of the state; this is the only way in which it can be done without extending ASM, but when time is global this leads to many global time progress steps (synchronizations).

Extension 1
The framework defined in [2] is the one we have used for the definition of the semantics of SDL.

In this framework, there is not only a time value associated with each state but also a state with each time value, meaning that consecutive states in a execution must have different times, thus enforcing property 2b. This poses the problem of immediate reaction for more than two consecutive state changes.

Extension 2
In [4], the concepts of [2] have been modified by allowing a state change to take time, meaning that this framework does not use property 1.

This model is due to the observation that the application of a set of update rules defining a state change does often not correspond to an event, but to an *activity* which might have *duration*. It is not made explicit in [4] what the status of the variables changed during an update is; it may

- either keep the old value during the update time (supposing that they remain visible for the environment)
- have an undefined value (a read by the environment leads to error)
- or have an arbitrary value in the domain (then, it is often better to consider finer granularity of computation)

As already mentioned in section 2, such a "transition", in the untimed setting considered as a *state change*, should become in the timed setting a *state*, the state "*in transition*" with some duration. There are two instantaneous state changes associated with an update: "*start transition*" - which might render undefined a set of variables, but has otherwise no effect on the state - and "*end transition*" - which leads to the updated state.

Extension 3: AsmL
AsmL [8] is an implementation of ASM for execution and testing of ASM specification. Its real-time concept is based on external time (monitored), where in practice the system time of the computer used for the execution of the model is used as the external source for providing the value of time: it is possible to read time and to react depending on its value. It is also possible to define triggers on time – which in any particular execution may be missed (those actions are lazy and property 4 is not enforced) - and waiting conditions. This model enforces property 2c. As finite experiments cannot express properties of infinite sequences, properties on infinite

sequences (properties 3) are irrelevant. The problem of this approach (which is chosen also in other real-time modeling tools, for example the UML based CASE tool Rhapsody) is that time is not abstract but concrete, and the obtained timed execution sequences are only relevant if the execution time for a given set of rules has a well defined relationship to the execution time of the implementation of the model executed at run-time in its target environment. Notice that, the smaller the observed time intervals are, the more important becomes the probing effect from the model, even if the measured times are relevant for the target platform. Our experiments with Rhapsody confirmed that the results are extremely sensitive to scaling, which is unsurprising, but disappointing from the point of view of the modeler. These kinds of models are useful for reactive systems where execution times are negligible with respect to reactivity constraints.

Extension 4: HASM

The Hybrid ASM or HASM approach [6] tries to marry the discrete world of state changes with continuous changes of a part of the state. This is done by introducing hyper-real numbers, especially infinitesimal numbers. Using an infinite but hyper-natural number of steps, these infinitesimals add up to "real" numbers that in turn represent time. This framework imposes a weakened form of property 2b which – due to the fact that computations are countable - is covered by the non Zenoness requirement 3a: an infinite number of steps cannot be done in bounded time. In a simpler form, a similar framework has been chosen in VHDL, where causally related events must be separated by at least some infinitesimal duration, denoted δ, and where only an infinite amount of δ steps have a measurable duration.

The main drawback of this framework is that there is no validation method being able to profit from this fine grained modeling concept.

4.4 SDL Time

SDL is promoted for the specification and design of distributed real-time systems. Its support for real-time behavior is essentially limited to timers and the underlying notion of global system time. In SDL, time is considered as external to the system and the system can only react on time progress. Time may pass everywhere, in states, in tasks, in evaluations of guards, and in communications and there is no means to limit these durations (such means are meant to be left to particular implementations). There are two exceptions to this general concept: there exists a notion of "*nodelay*" channel where sending and receiving a signal takes place at the same point of time. Also, once a transition has been detected as enabled, it *must* be taken *immediately* (property 4)

The formal semantics of SDL 2000 has been defined in terms of ASM [3]. In this work we have introduced *time* as a monitored variable of type real with only very few constraints, in particular the one saying that the values of time must be

increasing[11]. We had problems to define sending over *nodelay* channels (which is at the semantic level represented by a sequence of actions) due to the fact that all existing ASM frameworks insist on enforcing property 2c (non zero delay between causally related events).

5 A Proposal for Representing and Handling Time in ASM

In ASM, there exists no predefined notion of time, but an appropriate representation of time may be introduced explicitly, depending on the nature of the system to be modeled. We propose the introduction of some primitive and derived features defining a framework compatible with properties 1 and 2a, and where the other properties can be chosen freely.

Reading time

In order to be able to constrain the occurrence time of state changes in the rules depending on the occurrence times of earlier event occurrences, we have to make available the occurrence time of events in the states. In ASM, this is done in a similar way as the `Self` function is defined: we introduce a new function name `now` into the vocabulary and fix the interpretation of `now` to be the value of the `when` function. This allows then to store the occurrence time of an event in a variable for later comparison.

```
monitored now: Time
```

This function is monitored, that is defined by the environment which must respect the chosen set of axioms. The occurrence time of an event may be constrained also by system dependent guards.

Local clocks and storing time

In order to obtain an expressive framework, guards must be able to depend on occurrence times of any causally previous event. Timed automata use clocks for making these instants syntactically accessible.

In order to introduce similar clocks in ASM, we propose to introduce a domain of `MoveNames` similar to `ProgramNames`. In addition, we introduce a controlled function `time`:

```
domain MoveNames

controlled time: MoveNames -> Time
```

11 In fact, this is very similar to the approach in [2], mentioned earlier.

Using this function, the occurrence time of a move can be recorded in a move by a usual assignment `time(mN):=now`, where `MN` is a move name

Urgency, Updates taking time
As explained in section 3.3, in a compositional framework for simulated time, it is interesting to be able to distinguish between the moments when a transition is possible and the moments in which the transition is *urgent* (immediate, if enabled). In [11], it has been shown that the distinction of "urgent" and "lazy" transitions, as defined in the context of timed automata, is sufficient.

In order to simplify the expression of models where the application of update rules are interpreted as activities taking time, it is interesting to define a notion of "*timed update rules*", which associates with an update rule a *time guard*, restricting when it can be started, and an *urgency constraint*, expressing when it *must* be taken. This can be seen as a "macro" which can be translated into a sequence of moves: whenever the environment makes the enabling and the urgency condition true, the action of the transition is executed immediately and the sequence of moves terminated; when the enabling condition becomes true, but not urgent a legal waiting time is decided, but during waiting, when the urgency becomes true, the transition action is executed immediately. The timed update rule representing the three alternative transitions of the figure in section 3.3, has an enabling condition "*x=1 and now>=3 or y=1 or now>=3*", an urgency "*x=1 or now=3*" and an update depending also on `now`. A simpler notion of timed update rule which determines a waiting delay when the enabling condition becomes true and then just "waits" does not preserve free choice. Real-time programming languages distinguish for this purpose between *normal* signals – which are only taken into account after the "ongoing update" is terminated (run-to-completion policy) and interrupts – which are taken into account also within an "ongoing update" (that is between an above mentioned start and end move). Indeed if one considers determinism of un update to be important, then such interrupt constructs may be considered as undesirable.

6 Summary and Conclusions

We have discussed a set of properties which can be used to characterize and distinguish existing formalisms for the description of timed behaviours.

We have come to the conclusion that the best way of adding time to a system is:

1. No time progress in moves, they are instantaneous, whereas time progresses in states.
2. Time values are attached to moves.

Apart from these general guidelines, there are several ways to adapt the time mechanism to the application, and some high level derived notations are proposed, which have been proven useful in practise and which lead to more readable models.
Another conclusion is that the fact that in ASM there exists no explicit notion of input, but only a state, shared between system and environment, complicates the

introduction of an appropriate notion of time within this framework. In particular, if time is part of the state, time progress is a state change leading to state changes in all agents. A framework in which time passes in states and transitions represent instants is more flexible than one in which transitions "take time", and most formalisms existing in the literature are of this kind. In order to obtain such a framework for ASM in a convenient manner, we consider time as visible only in transitions and not in states.

A final observation is that frameworks for modelling timed systems defined independently of ASM do generally admit causal chains taking zero time, at least as long as there are only finitely many zero time steps in a row. All the papers on time extensions of ASM we have studied make an important point out of the requirement of absence of such zero time steps. Maybe this should be reconsidered.

We thank the ASM community for many helpful remarks during the period of writing of this article and for many interesting discussions around this topic.

References

1. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, Oxford University Press, 1995
2. Y. Gurevich and J. Huggins: The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proc. of CSL'95, volume 1092 of LNCS*, pages 266-290, 1996
3. SDL Formal Semantics Project. ITU-T Study Group 10: SDL Semantics Group. URL: http://rn.informatik.uni-kl.de/projects/sdl/
4. E. Börger, Y. Gurevich, D. Rosenzweig: The Bakery Algorithm: Yet another Specification and Verification, In: *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995
5. N. Lynch, R. Segala, F. Vaandrager, H.B. Weinberg: Hybrid I/O Automata. In R. Alur, Th. Henzinger, E. Sontag: Hybrid Systems III. LNCS 1066, Springer-Verlag, 1996.
6. H. Rust: Hybrid Abstract State Machines: Using the Hyperreals for Describing Continuous Changes in a Discrete Notation. In: Y. Gurevich, Ph. W. Kutter, M. Odersky, L. Thiele (Eds.) *Abstract State Machines - ASM2000*, TIK Report 87, 2000, ETH Zürich
7. D. Beauquier, A. Slissenko: On Semantics of Algorithms with Continuous Time. Technical Report 97-15, Dep. of Informatics, University Paris 12, 1997.
8. Y. Gurevich, W. Schulte, C. Campbell, W. Grieskamp: AsmL: The Abstract State Machine Language, Version 2.0, Microsoft Research, Redmond, 2002.
9. R. Alur, D. Dill: A Theory of Timed Automata, Proceedings of the 17th Int. Colloquium on Automata, Languages, and Programming, 1990.
10. N. Lynch: Distributed Algorithms, Morgan Kaufman Publishers Inc., San Francisco 1996.
11. S. Bornot, J. Sifakis: An Algebraic Framework for Urgency In *Information and Computation* vol. 163, 2000
12. S. Bornot, G. Gößler, J. Sifakis: On the Construction of Live Timed Systems In S. Graf, M. Schwartzbach (Eds.) *Proc. TACAS 2000* LNCS vol. 1785, Springer, 2000
13. M. Bozga S. Graf, L. Mounier: IF-2.0: A Validation Environment for Component-Based Real-Time Systems In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen* LNCS 2404, Springer, 2002

14. R. Alur, C. Courcoubetis, Th. Henzinger, Pei-Hsin Ho: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In *Hybrid Systems 1992*, LNCS 736, 1992
15. Y. Gurevich and J. K. Huggins: The Semantics of the C Programming Language. in Selected papers from CSL'92 (Computer Science Logic), Springer LNCS 702, 1993, 274—308
16. R. Stärk, J. Schmid, and E. Börger: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001
17. SDL 2000 – ITU-T Standard Z100
18 A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs,P. Le Guernic, R. de Simone: The synchronous languages, 12 years later. Proc. of the IEEE , Volume 91 Issue: 1, Jan 2003
19. J. A. Bergstra, A. Ponse, and S. A. Smolka, Editors: Handbook of Process Algebra. Elsevier, ISBN: 0-444-82830-3, 2001
20. J. Wang: Timed Petri Nets, Theory and Application, Kluwer Academic Publishers 1998
21. ISO/IEC: E-LOTOS (enhanced LOTOS), document ISO/IEC 15437:2001, 2001
22. G. Gößler, J. Sifakis: Composition for Component-Based Modelling (PDF), Proceedings of FMCO'02, held Nov 5–8, 2002, Leiden, LNCS 2852, pp 443-466.