A Framework for Time in FDTs

Susanne Graf

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

Andreas Prinz

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

Abstract. ASM is recognized as a useful formalism for defining the semantics of programming languages and modeling formalisms, but it has not initially been intended for expressing non functional properties. Also, during the work with the SDL formal semantics, problems of expression of time progress occurred at several places. Several proposals for timed ASM based on quite different approaches have been made. On the other hand, there exist a number of frameworks for modeling timed computations outside the world of ASM. It seems useful to try to elaborate a common understanding of time in ASM. We propose a general time model based on timed events, where time passes only in states, to which the existing frameworks can be reduced. Furthermore, we provide a number of options for defining useful time frameworks and discuss some of their consequences. We use this framework to give an overview on a number of approaches on modeling time in and outside the context of ASM. As ASM are a very general framework for specifying state-transition systems, our approach is valid for state-transition systems in general.

1 Introduction

Abstract State Machines (ASM) [1] are considered a good formalism 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 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 a behavior depending on time – as done 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. 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. We give an overview on the principles of ASMs 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 real time extension one can consider without loss of generality that a state is defined by the values of a set of variables. The notion of "atomic step" (called a *move*) is defined by means of (arbitrarily complex) rules for assigning a new value to each variable depending on the old values of all variables. The general idea is close to the one of synchronous languages [18], and like there, the result of an update is required to be deterministic. 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 not be written by "the system". The monitored values evolve a priori in an unconstrained fashion, but the possible evolutions of their values can be explicitly restricted by constraints. That means, the behavior of the environment may

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

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 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. *Teams*, consisting of a set of agents, are also viewed as agents and represent synchronization actions as they exist in process algebras [19]. The semantics of distributed ASMs is given by a partially ordered set of moves, denoted <, and a conflict relation on moves, denoted #, where the order and the conflict relation are disjoint. This defines the set of *runs* as the maximal sets of moves without conflicting events, where absence of order between moves expresses independence. In addition it is required that in any run the moves of each agent are strictly ordered, meaning within the set of moves of an agent, all minimal successors of a move are in conflict, that is represent alternatives. The set of executions is the set of state sequences induced by all possible linearizations of all runs. Notice that in the formalization of ASM, the conflict relation is implicitly defined by an enabledness predicate on states. The use of an explicit conflict relation as in the well-studied theory of event structures allows expressing some properties at a more abstract level and avoids introducing states.

In ASM, there exists no predefined notion of *time* or *trigger*. Each agent has "its rule" which can be applied on any state independently of any other agent or the changes of the environment. The existence of *guarded rules,* representing partial functions from global states to global states, allows constraining the possible interleavings of moves of different agents and the environment.

Based on this general setup in ASM, we consider a system description to consist of (a) a description of the state (defined as the valuation of a vector of variables), and (b) a description of the state changes 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 is a subset of moves closed for <, that is,

$$\forall m \in F\ (\exists m' \in R \wedge m' < m \ \Rightarrow\ m' \in F)$$

The set of runs satisfies the following closure condition, that for every possible extension of a prefix according to <, there exists indeed a run:

Property 0: $F \in \mathrm{prefix}(R) \wedge \exists m, m' \bullet m \in F \wedge m' \notin F \wedge m < m' \wedge m' \in \min\{m'' \mid m < m''\}$
$\forall m' \in \min\{m'' \mid m < m''\} \bullet \notin F \Rightarrow \exists R' \bullet F \cup \{m'\} \in \mathrm{prefix}(R')$.

2.2 Timed Systems

We want to add timing information to a system described in terms of a set of ordered or partially ordered purely functional "*observations*" which represent states and transitions moves (or moves, in other contexts also called *events*). We need to decide

which observations to extend to timed observations and how. Both states and moves may be interpreted as time points (instants) or durations.

For a user level modeling formalism or a programming language, a very natural interpretation is often the one attaching *duration* with both states and moves. There exist ASM frameworks where only states are extended with time points (see section 4), but this poses the problem that the set of possible states may be dense, and so a computation. This is in contradiction with the requirement of ASM that computations contain countably many steps. There are at least two reasons why it is preferable at the semantic level to interpret states as having some extension in time and events as instants. The first reason is that the explicit distinction between "consuming time by waiting" and "consuming time in activity" can always be made by distinguishing two different *kinds of states*, those in which a given agent is reactive to changes in the environment, and those ("in activity") in which the previous state is "frozen" and only a chosen move can occur after some delay.

The second reason is that associating duration with moves poses the problem of the definition of the state during the move, as visible from other agents: indeed, other agents may be able to move during the move of a particular agent, meaning that some state must be defined anyway. Furthermore, when transitions are instantaneous, interrupts can be modeled in a much more abstract manner: by definition, an agent is only reactive to its environment in states, and moves are atomic. If time consuming activities of a system are modeled by moves with duration, there exists no means to model the "interruption of a time consuming activity" at an abstract level. The only means consists in refining the activity into sub-activities (taking less time) with intermediate states in which the interrupt can be received. Obviously, this is the reality in an operating system, but at model level one would like to express the concept of "interrupting an activity at *any point of time*". This can be done by interpreting activities as states which can exclusively react to "interrupts", and moves as the "instants at which the result of an activity becomes visible to the outside world".

Another interpretation comes from the world of reactive systems, where timing is only important in the environment. The timed behavior of the system depends exclusively on the timed behavior of the inputs – which are supposed to be time dependent piecewise constant functions. Under this assumption, the system is supposed to react only to *changes* of the environment, and this reaction can be considered as instantaneous. In this case, moves are interpreted as instantaneous because the reaction time of the system is abstracted away (it is supposed that the reaction of the system is always visible before the environment changes again) and states represent the time intervals during which the environment, or the system's view of it, does not change.

This brings us to a first property of timed computations. 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 several earlier works, for an example see [11]).

Property 1: A state change takes place at a *point of time*, i.e. with the moves are attached time points, by extending M to $M_T \subseteq M \times Time$ and introducing a function `when` providing the time point of a move as the projection on the second argument:

$$\texttt{when}: M_T \rightarrow Time$$

This means that time is not part of the state, but there is an external function associating a time point with moves; the duration of a state is defined implicitly as the time passing between the occurrence times of its two adjacent events. The next property concerns the relationship between the time-induced order and the partial order of the moves induced by the causality relation. A natural minimal requirement is given below.

Property 2: In every run each event has at most one occurrence time, the conflict relation on M extends to M_T, and finally all causally ordered events are also ordered in time, i.e.

$$\forall (m,t),(m,t') \in M_T \bullet (m,t)\#(m,t')$$
$$\forall (m,t),(m',t') \in M_T \bullet m\#m' \Rightarrow (m,t)\#(m,t')$$
$$\forall\, m_1, m_2 \in M_T \bullet m_1 < m_2 \;\Rightarrow\; \mathtt{when}(m_1) \leq \mathtt{when}(m_2).$$

A consequence of the first constraint is that, for any given run R, when can be defined as a (partial) function $\mathtt{when}^R$: $M \rightarrow$ *Time*. More restrictive requirements on the time ordering are discussed in section 3.

3 Choice of a Timed Framework defined by a set of properties

So far, we have shown how to define a timed system syntactically in ASM, and how to interpret this extension semantically in the form of timed runs. Most frameworks for modeling timed systems suggest that a timed computation should have some additional properties not yet captured by these definitions. Such constraints are then 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 used in the 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 on the monitored time progress.

In this section we discuss a number of such constraints, which we have found in different frameworks.

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[2]. In this case, 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.

The 3rd item of 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 is strengthened.

[2] In general, real numbers are chosen representing a dense time domain.

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

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

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 events are causally ordered, their occurrence times are *strictly* ordered, i.e.

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

This forbids reaction chains in zero time. When distinct agents may be used to represent physical distribution, this property may make sense, but often distinct agents are used to express purely descriptive parallelism in a compositional manner, and here reaction chains may express synchronization or just conjunction of constraints.
Some frameworks with a strict notion of local time require even a stronger property implying that time distances can only be measured between causally related events.

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

$$\forall\ m_1, m_2 \colon M_T \bullet m_1 < m_2 \ \text{ iff } \ \texttt{when}(m_1)\ \ \texttt{when}(m_2).$$

All these axioms represent safety properties which can be used to strengthen the constraint on the immediate successor of each state. Note that property 2a is in general incompatible with property 2c.

3.2 Zeno Computations

So far, we did not justify the choice 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 non Zeno computations: In any infinite run, there is no upper bound of the time values attached with moves, i.e.

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

Often, we need density only to say that we do not want to require any global limit on the 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, no local state change is missed[3].

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

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

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,

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

Amongst the properties of this section, only 3c represents a safety property which can be checked locally and satisfied by restricting the order relation < and increasing the conflict relation #. Notice however, that these properties are to be taken as axioms defining valid runs as a subset of the runs induced by $(M,<,\#)$. This means that it is not necessary to check that a given run satisfies them, but it is sufficient to verify that valid runs satisfy a set of specified requirements.
In the context of timed automata, sufficient syntactic conditions on rules have been obtained guaranteeing the absence of Zeno computations and verification methods exists for checking that that all finite prefixes can be extended to non Zeno computations.

3.3 Maximal Progress and Urgency

In general, a set of time constraints allows several alternative time stamps to be attached with any event. 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?

Property 4a: Maximal progress: Each state change takes place as soon as it is enabled or is disabled by a conflicting transition which is a constraint on the set of continuations of a prefix:

$$\forall F, F \cup \{(m,t)\} \text{ a prefix } \Rightarrow \quad .$$

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

$$\neg\exists(m,t') \bullet t \le t' \wedge F \cup \{(m,t')\} \text{ a prefix } \wedge$$
$$\exists R.F \in \text{prefix}(R) \wedge (m,t) \notin \text{R} \Rightarrow (m',t') \in R \wedge m \# m' \wedge t' \le t$$

Computations with maximal progress are intuitively the most interesting ones. Nevertheless, it supposes to assume the controllability by the system over the time points at which a transition is taken. In practice, maximal progress is not the right option for all transitions: the execution time of an action of 2 to 3 time units has to be modeled by a state for which there exist runs with outgoing transitions at any time point between 2 and 3, but for which no run exists in which time progresses beyond 3 without that the state has been left. In this case a different form of urgency is needed, saying that a move is taken in the time interval in which it is enabled (there exists a run including a corresponding move) or it is disabled by a conflicting move

Property 4b: time progress does not disable transitions, i.e.

$$\forall F, F \cup \{(m,t)\} \text{a prefix exactly for } t \in [t_1,t_2] \Rightarrow$$
$$\exists R.F \in \text{prefix}(R) \wedge (m,t) \notin \text{R} \Rightarrow (m',t') \in R \wedge m \# m' \wedge t' \in [t_1,t_2]$$

3.4 Consistent time extension

We have said earlier, that a timed computation is defined by extending partially ordered moves with time points. An interesting constraint is the one saying that time extension does not introduce local deadlocks: that is, the duration in some local state cannot be infinite unless the untimed computation ends in this state.

Property 5a: Time extensions do not introduce local deadlocks, i.e a timed computation can be extended with an action an agent *i* if this is the case for the corresponding untimed computation

$$\forall F \subseteq M_T \text{ a prefix} \wedge m \in \text{agent}_\text{i} \wedge F_M \cup \{\text{m}\} \text{ an untimed prefix } \Rightarrow$$
$$\forall R.F \in \text{prefix}(R) \bullet \exists m' \in \text{agent}_\text{i} \wedge m' \in R \wedge m' \notin F$$

A framework in which this condition is syntactically enforced is when agents are time complete: a typical example is when time extensions are introduced in the form of timers, and whenever time progress disables some move, an alternative *timeout move* is enabled.
A stronger requirement, expressing orthogonality of timed behavior and functional behavior, says that the projection of timed runs to untimed ones is exactly the set of untimed runs up to the equivalence introduced by causal independence between events.

Property 5b: Timed behavior is orthogonal to system behavior, i.e. for any untimed partially ordered run there exists a timed one:

$$\forall F \subseteq M \; F \textit{ conflict free } \exists F_T \subseteq M_T \text{ where } F_T \text{ conflict free}$$

A stricter, but constructive version requires that every timed prefix of an untimed run can be extended in accordance to the untimed run.
This is an interesting property which in frameworks with pure interleaving semantics is even hard to formulate. This property is not satisfied by time dependent systems. A framework for syntactically enforcing it is not very interesting (it can be enforced by

allowing only time constraints imposing minimal occurrence times of moves). Nevertheless, a typical schedulability analysis consists in validating exactly this property, given time constraints in the form of worst case execution times and deadlines.

Notice, that these properties, contrary to the properties 2 and 3, do not restrict the set of runs, but require the existence of certain runs. For this reason, they do not represent axioms, but typical requirements that are often imposed. For example [22] provides a framework for guaranteeing deadlock free systems.

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 of models. A system is represented as a transition graph and a transition is labeled by a name and a constraint on when it can take place. Transitions represent named instants of change (events) and time progresses in the states (property 1). 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, that is non countable executions are not forbidden. 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, time constraints are expressed by means of constraints on *clocks*. The values of all clocks increase at the same pace (at the pace of external time which is not explicitly represented); clocks can be reset to zero in transitions; they represent the duration since this transition or event until they are reset for the next time. A set of timed automata defines a conjunction of constraints where events with the same name are considered as the same event[4].

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 respect to the set of properties of section 3: timed automata impose only the minimal restriction (property 2) on timed behaviors. In particular, causal chains in zero time are possible. Only non Zeno computations are considered as valid (property 3a). Sometimes, it is proposed to enforce non Zenoness by enforcing 3b syntactically (require a minimum delay in each syntactic loop). Timed automata do not require maximal progress, but there exist several means to express urgency:

[4] In [12] more flexible compositions of constraints have been defined close to those of timed Petri nets[20].

- *Invariants,* associated with states are used to limit the time during which the system is allowed to stay in this state; when the invariant becomes false (due to time progress or if the invariant is false on entering the state) the state must be left by some enabled transition[5].
- *Timed automata with urgency* [11] associate a *urgency* with transitions (specifying constraints on when transitions *must* be taken, meaning that time cannot progress beyond such a point without that this transition or an alternative one is taken). They define three different kinds of transitions: *eager* transitions (enforcing maximal progress, that is property 4a), *delayable* transitions (must be taken before the time guard becomes false, that is they satisfy property 4b) and lazy ones (which may be disabled by time progress)..

Conservation properties such as expressed by properties 5 are not enforced by timed automata, but for most systems these properties (or variants of them) are added to the set of requirements. Notice that in the IF language [13], which is close to abstract state machines, timed automata with urgency have been adopted as the underlying timing model.

Hybrid automata

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. This means, state changes are considered to occur "*at*" some instant *t*.

As timed automata are special hybrid automata, where the only continuous variables are *clocks* which increase at the same pace as time and can be reset to *0* in discrete transitions, the above mentioned variants of timed automata can also be defined for hybrid automata.

4.2 Timed Petri nets and time extensions in process algebras

The aim of Process algebras [19] and Petrinets is the description of coordination of activities. The main aim of their time extension is to express constraints on the occurrence times of synchronization events. Thus, they 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 have to be added at analysis time as verification conditions or a hypothesis. They differ with respect to properties 4 and 5.

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 and partial actions (which are able to synchronize with other subsystems) are

[5] Obviously, the non proper usage of invariants can lead to time lock which has to be considered as an inconsistent specification.

delayable – can be taken at any point in their time interval, whereas *invisible* or complete actions –which cannot synchronize anymore with other actions - must be taken as early as possible. This enforces maximal progress for system internal actions (property 4a). There exists also the possibility to *set* time in a transition, and this is not interpreted differently from reading time and constraining its occurrence time to a single time point. As in the previous frameworks, property 5 is not part of the model but has to be added as a property to be verified.

A Petrinet [20] defines a set of partially ordered runs and time extensions express constraints on the occurrence times of synchronization events. Petrinet "transitions" are interpreted as "waiting states" and an event is a token consuming move from states to a transition or a token producing move from a transition to a set of states. There are several variants of timed Petrinets. We mention here two of them. One variant is very close to the model of timed process algebras. Each arrow leading into a transition has a constraint on when this synchronization can happen. If possible, it takes place at the earliest possible time point, but it might also be impossible if the intersection of the constraints is empty. That is, property 4a holds, but not property 5. Another variant of Petrinets, constrains only the *ready times* of the arrows entering a synchronization transition. The actual occurrence time of a synchronization event may be any time point in which all participants are ready before some defined limit time, and at the earliest possible time, if all participants are ready after this time limit. That means that a synchronization event is *delayable* up to some point of time, and then becomes *eager*, in terms of urgency in timed automata. As no synchronization becomes ever impossible through time progress, property 5b is enforced by this framework. This makes it particularly interesting when liveness is more important than hard real time constraints. The approach for timed automata in [21] is close to this model.

4.3 Time extensions of ASM

The general problem to introduce time into ASM is that in the ASM framework the notion of state is central to the whole theory, but in the definition of partially ordered runs the events (moves) play the central role. Most of the time extensions for ASM start by considering time as a part of the state (a priori, it is the only way in which it can be done without extending ASM) and try to make this fit with the rest of the formalism.

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

It presents the idea, not only to associate a time value with every state but also to attach a state with every time value, meaning that consecutive states in a execution have different times and so enforcing property 2b. The main problem in this framework is that time is part of the state, and thus time progress implies state change. The model is hence described using a dense sequence of states for each time point. This is contradictory to the requirement that computations are countable. Therefore, one will

observe time only at points of moves. This, in turn, brings the problem of consecutive state changes and 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. The agents may but cannot be forced to react immediately (property 4 does not hold).

This model is due to the observation that the application of a set of update rules defining the state changes does in many cases not correspond to an event, but to an *activity* which might have *duration*. Depending on the considered computation model, the variables changed during the update might

- either keep the old value during the update time (supposing that their value is copied into some internal non observable variable and updated at the end)
- have an undefined value to express the fact that if the environment tries to touch or read this value during the state change, this is an error
- or have an arbitrary value in the domain; in this case it is often more appropriate to consider the computation at a finer granularity to be able to say more about the possible values of the variables when an interaction with the environment happens.

As already mentioned in section 2, the transition, in the untimed setting considered as a state change, becomes a state, the state "*in transition*" with some duration; there are two state changes associated with an update, each of them occurring at an *instant in time*: "*start transition*" - which might render undefined a set of variables, but has otherwise no effect on the state - and "*end transition*" - which results in the updated state. We have discussed the inconveniences of this approach as a basic feature in section 2.

Extension 3: AsmL

AsmL [8] is an implementation of ASM for execution and testing of ASM specification. The real-time concept of AsmL 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 the system 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 - and waiting conditions. This model enforces property 2c. As finite experiments cannot express properties of infinite sequences, properties of 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 one can suppose the measured times are relevant for the times observed on the target platforms. Some experiments with Rhapsody showed that the observed results are extremely sensitive to scaling, which is unsur-

prising, but disappointing from the point of view of the modeler. These kinds of models are useful, but they do clearly not cover all the needs in modeling of real-time systems.

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. A similar framework has been chosen in VHDL, where any causally related events must be separated by at least some infinitesimal duration, denoted δ and where only an infinite amount of delta steps have a positive duration. This framework thus imposes a weakened form of property 2b which is covered by the non Zenoness requirement 3a: an infinite number of steps cannot be done in zero time.

Also in this model computations must be made countable, and in this case it is not totally clear what the introduction of hyper-reals brings in. In any case it seems more appropriate for modeling systems interacting with a continuous environment without sticking to a discretization a priori, similar as hybrid automata. Moreover, 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. However, its support for real-time behavior is essentially limited to the use of 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 depending on time (like in ASML). Time can basically pass everywhere, e.g. in states, in tasks, in evaluations of guards, and in communications. There is no means in SDL 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 as soon as possible, i.e. *immediately*.

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[6]. 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 ASM frameworks insist on enforcing at least property 2c (non zero delay between causally related events).

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

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
We want to be able to model systems in which the behavior can depend explicitly on time. For this purpose, we introduce an explicit expression allowing to access time in moves

```
monitored now: Time
```

which provides in any move *m* the current instant `when`(*m*), which is also the exit time of the preceding and the entry-time of the next state.

This function is monitored, that is defined by the environment. The occurrence time of an event can be constrained by a guard which may depend on the current state and the occurrence time of any causally preceding event[7].

Time progress and urgency
We are interested in simulated time where the possible evolutions of the values of `now` may depend on system progress. In particular, we need to describe timed behaviors, in which the time distance between two moves is within some interval like required by properties 4. Due to the partial order semantics, in ASM, an explicit notion of urgency is not needed, when each agent has at any point of time only one enabled transition. But there are two situations, where the explicit definition of urgency is useful:

- when an agent has two alternative transitions, t_1 enabled upto 5ms and t_2 enabled after that time, then, one wants to express the fact that if the first one is enabled, then it will be taken, whereas the second is not possible.
- It is also useful to distinguish between transitions that are taken at the point of time at which they become enabled (they do not necessarily have a time dependent guard) and those which may be delayed.

Adopting maximal urgency as a general principle, both situations can be handled by adding transitions and by strengthening guards. In the first case, an additional transition t_0 is taken as soon as t_1 gets enabled, and t_0 disables t_2 and decides some arbitrary waiting time before t_1 is taken.

In order to obtain more readable timed ASM specifications, we propose the introduction as a shorthand of a notion of *urgency* associated with transitions, as defined in the context of timed automata.

[7] Like any value, the value of the occurrence times of events can be stored in variables and made locally or globally accessible

Updates which take time
We consider ASM models where state changes correspond to events and time passes in states. In order to simplify the expression of models where the application of update rules are interpreted as activities taking time, it is possible to define a notion of "*timed update rules*", which associate with an update rule a *time guard*, expressing when it can be started, an urgency, expressing when it *must* be taken, and a *duration* (in fact a constraint on its duration) expressing when it will be terminated. This is a shorthand which can be translated into a sequence of two moves where the first move just starts to count time[8] and the second one checks the constraint on the termination time and does the actual update.

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 state changes, they are instantaneous.
2. Time progresses in states.
3. Time values are attached to state changes.

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 of an update, 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 several unwanted state changes. 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 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. May be 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.

[8] It might also make the local state inaccessible to other agents.

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.