Time in State Machines

Susanne Graf, Andreas Prinz

Fundamenta Informaticae 77 (2007) 143–174
IOS Press

Time in State Machines

Susanne Graf*
Verimag, Grenoble, France
Susanne.Graf@imag.fr

Andreas Prinz
Agder University College, Grimstad, Norway
Andreas.Prinz@hia.no

Abstract. State machines are a very general means to express computations in an implementation-independent way. There are ways to extend the abstract state machine (ASM) framework with distribution aspects, but there is no unifying framework for handling time so far.

We propose event structures extended with time as a natural framework for representing state machines and their true concurrency model at a semantic level and for discussing associated time models. Constraints on these timed event structures and their traces (runs) are then used for characterising different frameworks for timed computations. This characterisation of timed frameworks is independent of ASM and allows to compare time models of different modelling formalisms.

Finally, we propose some specific extensions of ASM for the expressions of time constraints in accordance with the event-based semantic framework and show the applicability of the obtained framework on an example with a standard time model and a set of consistency properties for timed computations.

Keywords: Abstract State Machines, Time Model, Event Structure, Partial Order Semantics

1. Introduction

Abstract State Machines (ASM) [20, 34] have been proven to be an appropriate formalism for expressing algorithms, as well as interactive systems. ASM, initially called evolving algebras, have as their state space a set of algebras and define state transitions, i.e., changes of algebras, by rules written in an

*Address for correspondence: Verimag, 2, avenue de Vignate, Grenoble, France

imperative style. Interactive and distributed systems are usually described by distributed ASM, that is, a set of sequential ASM sharing the same state space but doing their updates independently of each other. The semantics of distributed ASM is defined in terms of a set of countable, partially ordered runs.

ASM have been used to define a formal semantics for programming languages (e.g. C# [18] and Java [67]), as well as the formal semantics of SDL 2000 [47], the most recent version of SDL [46].

ASM do not come with any predefined time model but several proposals for introducing time in ASM have already been made. We mention here the approach by Gurevich and Huggins [35] which defines a timed computation as a mapping from a $Time$ domain to the states of the considered run, which must be monotonic with respect to the causality order on states. This means in particular that, when $Time$ is an ordered domain, a timed run consists of a totally ordered set of states.

1.1. A General Time Model

Do we need a unifying time model for ASM at all? With respect to expressivity, there is no need to add explicit time concepts to ASM; any notion of (discrete) time may be expressed by an explicit ASM model of time and time progress, for example, in the form of a variable "time" and some particular agent defining time progress.

Sometimes, it is argued that fixing a specific time model may stand in the way of an "optimal" modelling in some cases. For example, the already mentioned framework of [35] which requires strict time progress made the definition of the real-time semantics of SDL quite unnatural, as SDL requires explicitly certain sequences of actions to take place without time progress (see [47, 32]).

An obvious argument in favour of predefined time concepts is that for each framework specific validation methods can be developed and proven correct. We aim at a framework that imposes a minimal set of consistency constraints on any timing framework and that then adds more constraints in order to represent the underlying semantic model of the modelling formalisms in question; in particular, we want to be able to capture the time models of *timed specification languages* — such as SDL, UML, hardware description languages like VHDL [8], synchronous languages [15] and related approaches like B [1] or time triggered architecture [49] — and those of *timed modelling formalisms* — like timed Petrinets [71], timed process algebras [54, 16, 45], duration calculi [25], actor languages extended with time [56, 57], all variants of timed and hybrid automata [5, 6, 22, 39], etc. We also seek to represent models of timed computations proposed in the context of ASM, such as the already mentioned model in [35], the model of HASM [64] and the time model in AsmL [36].

The above mentioned languages and their underlying semantic models of timed computations can be classified according to several distinguishing features:

1. their computation model, which may be state-based or event-based
2. where time can progress, in states or in events (state changes)
3. their underlying time model, in particular, if time is finite, infinite, linear, partially ordered, discrete or dense, qualitative or quantitative
4. the way in which time is associated with a computation
5. the constraints imposed on the relationship between the time and the computation models, in particular relations between time progress and system progress, non Zenoness constraints and urgency constraints, which all select the allowed runs from a set of potential runs.

In the following, we discuss the approach taken and the choices made in this paper. We choose a model for timed computation that fixes a model of computation, defines a general notion of time domain, fixes a way to combine a time and a computation model (items (1) to (4) above), and is general enough to represent also all other categories of timed computation models. Particular semantic frameworks — making specific choices for the issues mentioned in item (5) — are then defined by imposing additional constraints expressed in terms of the ingredients of the semantic model.

1.1.1. Computation Models

For representing computations of reactive or interactive systems, there is a distinction between a state-based and an event-based approach.

The state-based view is represented by a state transition system $TS = (S, \rightarrow)$, where S represents a set of states and $\rightarrow$ a transition relation. The event-based view is represented naturally by an event structure $ES = (E, <, \#)$, where E represents a set of events, $<$ the causal order between events and $\#$ represents the notion of alternative (for more detailed definitions see [72, 55] and Section 2). Both views are dual and can be transformed into each other: an event e of ES represents a transition between two states $s \xrightarrow{e} s'$ in TS, and a state s in TS represents a family of sets of events in ES, each set corresponding to the transitions of a path leading to s. All computation frameworks allow to speak about a *precedence relation* between events or states for modelling a causal relationship.

In this article, we choose an event-based view because it allows a more direct expression of concurrency.

1.1.2. Time Models

Most frameworks for timed computations used in practice consider natural or real numbers for representing time and durations. A slightly more abstract notion of time domain has been proposed in the context of timed automata with urgencies [22, 21]. Notions of local time have been proposed and studied in the context of testing of distributed systems [24, 31]. Abstract time domains, defined only by a set of constraints on the precedence relation and on the distance function on the time domain, have mainly been studied in the context of *temporal logics* (see [61, 63] for overviews) which have precisely been introduced for studying and characterising notions of time.

Temporal modalities allow the expression of dynamic properties as shorthands for quantified expressions on the set of time points accessible via the precedence relation between time points. The expression $t \models \Diamond p$, for example, represents the assertion "*There exists a time point t' later than t at which p holds*"

In temporal logics, time domains for expressing quantitative timing properties are obtained in two, equally expressive ways (see e.g. [63] where both frameworks are presented): either by adding some arithmetic properties to the time domain itself, or by defining a function $dist$ which associates with any pair of time points a *distance* in some external domain with arithmetic properties, which we call *duration*. In existing approaches, the first alternative is often used with a time domain representing an additive group, such as $\mathbb{N}$ or $\mathbb{R}$, whereas in the second alternative, $dist$ is only required to be a metric or a pseudo metric.

Precedence relation and metric must satisfy some consistency conditions; e.g., in [63] it is shown how a (pseudo) metric $dist$ defines a precedence relation "$<$" for any given pair of time points representing the "origin" and the "direction" of time. Interestingly, this induced precedence is not necessarily total

but it may contain sets of "quasi synchronous" time points. We do not follow exactly this approach as the induced pair $(<, dist)$ does not enjoy all the properties one would reasonably require: in particular, for a given (large) c and an arbitrary small ϵ one may find pairs $(t_i, t_i'), i = 1, 2$ with $dist(t_1, t_2) \geq c$ and $dist(t_i, t_i') \leq \epsilon$, and nevertheless $t_1 < t_2$ and $t_2' < t_1'$. This means that the induced time domain forbids "approximation", which is important – especially for computations of systems interacting with a physical environment, changing continuously. For this reason, we impose constraints which imply that any time point t interpolating two other time points t_1, t_2 is at a distance from t_i that is not larger than $c * dist(t_1, t_2)$ for some constant c. In particular, this bound is also imposed on the distance between unordered time points.

Many variants of temporal logics with quantitative time have been proposed for expressing timing properties of computations. There are two main groups: those parameterising modalities with duration expressions [28, 37, 50] where $s \models \Diamond_{exp} p$ represents the assertion that "*there exists a state s' reachable from state s which satisfies p and which is at a time distance d from s satisfying $exp(d)$*". The second type of extensions introduces formulas with explicitly quantified *duration variables* or *clocks* and predicates on them; examples are TPTL [7], an extension of linear time temporal logic [60] or TCTL [4], an extension of branching time temporal logic [14, 27].

An axiomatisation of a temporal logic expresses constraints on the precedence relation and the distance function (see [63] or [26] for a comprehensive treatment of this topic): for example, in a context excluding cyclic time, the axiom $\Diamond p \implies next \Diamond p$ expresses the fact that time is dense[1].

The choice of axioms concerning the precedence relation determines the nature of time: whether time is linear or may be branching, whether it is allowed to be cyclic or not, and whether state changes are allowed to be dense, whether it contains its fix points, or whether time has an origin or an end.

We take from temporal logics the idea that a model of time is defined by constraints on the precedence relation and the distance, but we express them directly on the model, without introducing any logic for the expression of dynamic and temporal properties.

1.1.3. Models of Timed Computations

A model of timed computations can be obtained from a model of time and a model of computations in different ways. A first decision to take is if events represent time points or if they may have duration. There exists frameworks considering such durative actions: examples are the duration calculus [25], frameworks of action-based agents in which functionally atomic actions need some time to be computed [56, 57] and extensions of event structures with time (e.g. [9]). Notice that durative actions are well adapted for so-called transactional models where activities cannot be influenced by the environment, once they have started. But it is generally agreed that considering events as *time points* is the more general approach; indeed, it allows to express durative actions by means of a pair of a *start* and an *end* event. For this reason, we choose this second alternative.

Then, in an event-based setting where events represent time points, one may either introduce a mapping from the events of the computation domain to elements of the time domain or consider the set of events itself as a time domain and distinguish a causal and a time precedence relation. This means that only distances between time points represent relevant quantitative timing information. This is sometimes called implicit timing [52] and has been used in the context of temporal logics. It has also been used as a

[1]This property implies that it is always possible to let time progress and reach p later, which again means that infinitely many time points must exist between any to states (alternatively time must be cyclic).

natural way of extending event structures with time [9, 10] and it is used at language level in formalisms such as Petrinets or timed automata, where only time distances are specified, whereas time points are derived. All existing frameworks of this nature, either consider global time or have metrics which are not related to time [10].

For the formulation of our semantic framework in Sections 2 and 3, we choose implicit timing. At the level of ASM, we propose either to use an implicit approach constraining only distances, exactly as in timed automata, or the definition of a mapping from events into some predefined time domain.

1.2. Constraints on Timed Computations

The semantics of any of the mentioned modelling formalisms can be expressed in terms of an event structure extended with time as we define them. The time models of different frameworks, however, are characterised by different constraints on the relationship between the ingredients of a timed event structure, in particular between the causal and the time precedence relation. Some constraints, for example, the one guaranteeing that causality order cannot contradict time order, are valid in all useful time frameworks, whereas others are shared only by some frameworks. All frameworks considering timing as a function from $Time$ to $State : Exp$, such as those discussed in [35] or in [68, 42], require strict time progress along causal chains whereas this is not required by many other frameworks. Such constraints and the frameworks in which they are used are discussed in Section 3.

In our setting, we consider moves as instantaneous and attach *time points* to them. In order to consider moves as *activities* which *require some duration* d *to be executed*, as in [19], we introduce an intermediate move and a corresponding state, marking the instant when the *move starts*, whereas the original move is executed later, after the delay d has elapsed. That means, the duration of a move is mapped to a delay in the new intermediate state *in-activity*. This allows also modelling concurrently ongoing activities with non-synchronised start and end points.

1.2.1. Handling an Environment with Continuous State Changes

We aim at capturing the particular constraints imposed by real-time systems that interact with a physical, continuously evolving environment (reactive systems). For being able to validate or simulate such systems, in addition to the system itself, an appropriate abstraction of the behaviour of the environment has to be provided. Such an environment can either be expressed declaratively in the form of a "temporal constraint" in some logic, or as a part of the system, in the form of a specific (set of) component(s) or agent(s) realizing (non deterministically) the possible behaviours of the environment.

The second option guarantees executability of the joint behaviour of the environment and the system and is often used for simulation or exploration-based verification. As the environment is "continuous", it apparently contradicts the requirement that computations are intrinsically constrained to be countable. Nevertheless, as the system specification is discrete, and only the values of the continuous environment relevant for the system need to be explicitly represented, at most a countable set of environment events are relevent in each computation. This means that, for any given computation, the time domain can always be made discrete (by just including the relevant time points for this computation) and ASM need not to be extended.

A dense time domain allows an arbitrary, not a priori fixed variability of the time interval between two observation points. Formalisms like hybrid automata [5, 38] do not fix any discretisation of a hybrid

system a priori. Some existing engineering tools, such as Simulink [53] allow denotating the behaviour of such environments by means of differential equations, and use numerical methods to compute a relevant discretisation for a given time step or a required precision. To know that discrete steps are always sufficient in this case, obviously does not solve all problems. As the environment at time t is only defined by numerical approximations, the meaningful tests in the system are restricted; for example, equality tests should be avoided, as a discretised implementation may miss the exact value. In the literature, several criteria have been developed to verify whether a system specification given by a timed automaton is *realizable* or not. These criteria are expressed using constraints on infinite computations. Examples are non Zenoness [41] and other criteria of realisability which have been intensively studied in the context of timed automata [40, 62, 58, 3].

1.2.2. Urgency and Time as a Property of the Environment

In the context of ASM, the most natural modelling of time is to consider time as a part of the environment and not a part of the system state; thus, the system may read time and act depending on time, but not modify it. A clock synchronisation algorithm may assign a value to the "system clock", but such a "clock" is part of the system and is different from "time".

But time cannot evolve arbitrarily, it must satisfy all the already discussed constraints representing the time framework under consideration. It must also satisfy system specific time constraints which formally are hypotheses on the environment. Such hypotheses include constraints representing assumptions on the execution time of activities, reaction times to requests, as well as the frequency at which the environment changes. Such changes are interpreted as a "request to react".

All real-time specification formalisms allow specifying constraints on the time points at which events *are allowed to* occur. For specifying events that *must* occur at a certain time point or within some interval, different solutions have been proposed. Most real-time programming formalisms do not allow the expression of such environment constraints, but suppose that they will be specified externally for verification and testing activities. Declarative logic-based formalisms, such as temporal logic or TLA [51], enforce them in the form of explicit liveness constraints. Timed automata express a priori safety properties but with respect to time, some progress constraints can be imposed: either by invariants, specifying the time points at which the control flow is allowed to enter or to remain in some control state, or by urgency attributes [22], expressing when a transition must be taken with respect to its enabledness: as soon as enabled (eager), somewhere in its enabledness window (delayable) or never (lazy). Other formalisms, like Petrinets, allow only time constraints local to a transition and they are by definition interpreted as delayable.

We characterise urgency at the level of semantics as an additional constraint for selecting eligible runs in an event structure and choose urgency attributes as a means of expressing urgency in ASM.

1.3. Structure of this Paper

The paper is organised as follows. In Section 2, we choose event structures as a semantic framework for systems with partially ordered computations. We present general concepts of time and duration which have been studied in the literature, and finally define a framework for timed computations by superposing these two concepts. In this paper, we are mainly interested in constraints on the relationships between the superposed models of time and computation. Some fundamental relationships are defined which

should be true in any framework for timed computations. By means of examples, we also show that non-standard time domains can be considered as useful, as the corresponding timed computations can be interpreted back into a standard interpretation of time.

In Section 3, we discuss a set of constraints on the relationships between the superposed models of time and computation. Some of them represent reasonable properties for any framework of timed computations, whereas others correspond rather to choices which have been made in some but not in all frameworks studied in the literature. These properties can then either be used as a kind of sanity check or classification for any proposed notion of time, or in the context of verification, as a restriction on the set of computations to be verified.

In Section 4, we give a very brief overview of the principles of Abstract State Machines and then suggest one possible way to describe time in ASM specifications at the syntax level. This way environment constraints related to time can be expressed within the extended ASM time framework. We discuss a choice of axioms from Section 3 and options for the specification of time progress; in particular, we argue in favour of an explicit notion of urgency as a means for a non-uniform time progress policy.

Finally, in Section 5, we show the flexibility of the proposed framework applying it to a version of a railway gate controller which crystallised out to be a standard "toy example" used in several papers proposing time extensions for ASM [35, 66].

2. An Overview of Computations and Time

In this section, we identify the basic domains to consider for the representation of the *behaviour* of systems and of *time*, as well as some basic laws on *timed behaviour* which are our object of study.

We define a semantic framework for timed computations useful for distributed systems, where computations are partially ordered runs. Consequently, the behaviour of a system – that is the set of all its possible computations – corresponds to an event structure [72, 55].

An alternative option is to define a state structure, which is discussed briefly in Section 2.4. The main advantage of an event-based model is that it preserves the notion of concurrency in the form of a partial order on events at semantic level. The notion of *concurrency* that might be present at the language (syntax) level is lost when the semantics is represented as a global transition system: events may be concurrent, but (global) states represent alternatives[2]. Notice that there is also a notion of *asynchronous transition system* [65, 13] which allows to extract information on the concurrency of events from a state-based transition system.

Notice also that we are not the first ones to propose the use of an event-based model for reasoning about Abstract State Machines, for example, also [12, 66] feel the need to identify events for reasoning.

2.1. Semantic Model: States, Moves and Runs

The semantic model of a distributed ASM is a set of partially ordered runs representing alternative executions. The corresponding event structure [72] is defined by a set of events, also called *moves*, a *partial order* relation defining the causal dependency between events, and a *conflict* relation indicating which events occur in alternative runs. Unrelated moves are *concurrent*.

[2] therefore, state-based approaches preserve often a parallel operator at semantic level

Definition 2.1. (Event structure)
An *Event structure* is a triple $ES = (Move, <_f, \#)$ where

- $Move$ is a domain of *moves*,
- $<_f \subseteq Move \times Move$ is a well-founded irreflexive partial order on moves representing causal order of moves, and
- $\# \subseteq Move \times Move$ is a conflict relation between moves, which is commutative and irreflexive.
- Moreover, events cannot be both, ordered and conflicting, i.e. $<_f$ and $\#$ are disjoint.

In the sequel, we use $\leq_f$ for the weak order derived from $<_f$ containing in addition the elements $(m, m) \in Move \times Move$, and $>_f$ to represent the reversed order. Out of the definition of an event structure, it is straightforward to define states and runs, if we think of states as the result of performing a set of moves.

Definition 2.2. (State)
Let $ES = (Move, <_f, \#)$ be an event structure. Then, a domain $State$ is a *state domain* if it is the largest set having the following properties:

- $State \subseteq \mathbb{P}(Move)$
- $\forall s \in State \,.\, s$ is downwards-closed[3] with respect to $<_f$
- $\forall s \in State \,.\, \neg\exists m, m' \in s$ such that $m \# m'$

This definition matches the usual understanding of an execution being an alternating sequence of states and moves: $s \xrightarrow{m} s'$ iff $s' = s \cup \{m\}$, where m is a minimal move such that $s <_f m$. This transition relation defines the transition system corresponding to the event structure and represents its interleaving semantics (see Section 2.5).

States as defined above have an implied partial order on sets. This gives a first trivial lemma which is also valid for runs (as defined below).

Lemma 2.1. $Move$ is totally ordered $\iff$ $State$ is totally ordered (with respect to set inclusion). □

Definition 2.3. (Run)
A *run* $R = (M, <_f)$ is a *maximal state* where the partial order $<_f$ of ES is restricted to $M \subseteq Move$.

As runs are used to represent "discrete computations", we restrict event structures to those in which each run is a countable set. The set $Move$ containing all moves of all runs need not be countable, meaning that we allow an arbitrary number of alternative runs.

For executions of runs on a single processor, it is often convenient to speak of sequentialisations of runs, that is, runs with totally ordered events.

Definition 2.4. (Sequentialisation)
A run $R' = (M, <'_f)$ is a *sequentialisation* of a run $R = (M, <_f)$, if $<'_f \supseteq <_f$ and $<'_f$ is totally ordered.

[3] A set is downwards-closed if it also contains all predecessors of its elements.

2.2. Time and Duration Domains

The second ingredient is a semantic model for real-time, consisting of the definition of domains for *Time* which represents the set of possible time points, and *Duration* which defines the co-domain of some metric that may be defined on *Time*.

When global time is considered, *Time* is equipped with a weak order relation $\leq_t$, and in case we are interested in quantitative assertions concerning time, we either need an external metric as already discussed in Section 1, or we have to require the existence of an arithmetic on *Time* as in [28, 22]. This motivates the fact that, in general, real or natural numbers equipped at least with addition are chosen. In this case, the *Duration* domain coincides with *Time*, which is likely to be the reason why they are not always well distinguished. Notice also that, for example, timed automata use only the domain *Duration*: the only time related variables are "clocks" of type *Duration* which increase with progress of time. Time itself is kept implicit. Such time models are sometimes called implicit time models [52].

When a partial order semantics is used to represent the semantics of systems, it can be interesting not to require global time, but to allow also time to be partially ordered.

Definition 2.5. (Time)
A pair $(Time, \leq_t)$ is a *time domain* if $Time$ is a set of *time points* and $\leq_t$ is a (weak) partial order over $Time$. We denote by $<_t$ the corresponding strong order not including the $t_1 \leq_t t_2$ which are also $t_2 \leq_t t_1$.

Such a time domain allows us to argue about qualitative time: $t_1 <_t t_2$ means that t_1 is *before* t_2.

In general, time is introduced in specifications in order to express quantitative time constraints, which is done by means of a notion of distance on *Time* representing the duration between time points.

Definition 2.6. (Duration)
A domain *Duration* is a *duration domain*, if it is an Abelian group with an operation $+$ and neutral element 0 which comes equipped with an order $<_d$ such that $0 <_d d$ for all $d \neq 0$, $d \in Duration$. $<_d$ is required to have a least upper bound D_∞ which is the limit of every strictly increasing chain and which needs not to be in $Duration$.

Definition 2.7. (Distance)
A distance $dist : Time \times Time \mapsto Duration$ is a *duration distance*, if it has the following properties:

(t1) $\forall t_1, t_2 \in Time \,.\, t_1 \leq_t t_2 \wedge t_2 \leq_t t_1 \implies dist(t_1, t_2) = 0$ (strictness)

(t2) $\forall t_1, t_2 \in Time \,.\, dist(t_1, t_2) = dist(t_2, t_1)$ (symmetry)

(t3) $\forall t_1, t_2, t_3 \in Time \,.\, dist(t_1, t_3) \leq_d dist(t_1, t_2) + dist(t_2, t_3)$ (triangular inequality)

(t4) $\forall t_1, t_2, t_3 \in Time \,.\, t_1 <_t t_2 <_t t_3 \implies dist(t_1, t_2) <_d dist(t_1, t_3) \wedge dist(t_2, t_3) <_d dist(t_1, t_3)$

(t5) $\forall t_1, t_2 \in Time \,.\, t_1 <_t t_2 \implies dist(t_1, t_2) \neq 0$

If reals are chosen as the duration domain, then the requirements (t1-t3) imply that $dist$ is a pseudo metric rather than a metric, as we allow $dist(t_1, t_2) = 0$ for distinct time points. In particular, we want to allow unordered time points to be at distance 0 (see also Section 2.4).

The order on *Time* and $dist$ cannot be chosen independently, as a distance already defines a preorder on *Time* as shown in [63]. Property (t4) expresses that time progress to a closer point is necessarily shorter than time progress to a time point further away. (t5) says that strictly ordered time points may not have distance 0. Notice that, the construction in [63], which simply orders all time points according to their distance to some arbitrarily chosen origin, does not imply this restriction. In addition, the above constraints allow the order to be undefined almost everywhere, whereas [63] orders all pairs, except those which cannot be ordered by their criterion.

An important consequence of (t4) and (t3) is that unrelated points must be closer to each other than twice the distance of the closest related point.

Lemma 2.2. (maximal distance between unrelated events)
If $dist$ is a duration distance, then
$\forall t_1, t_2, t_b, t_e \,.\, t_b <_t t_1 <_t t_e \wedge t_b <_t t_2 <_t t_e \implies dist(t_1, t_2) \leq_d 2 * dist(t_b, t_e)$.

Proof: From (t2) and (t3), we get $dist(t_1, t_2) \leq_d dist(t_1, t_e) + dist(t_2, t_e)$. Moreover, from (t4) we get $dist(t_1, t_e) <_d dist(t_b, t_e)$ and $dist(t_2, t_e) <_d dist(t_b, t_e)$, completing the proof. □

Definition 2.8. We call a tuple $T = (Time, \leq_t, dist)$ a *time domain with a distance* if $(Time, \leq_t)$ is a time domain and $dist : Time \times Time \mapsto Duration$ is a *duration distance* for some duration domain $Duration$ satisfying the constraints (t1) to (t5).

2.3. Adding Time to Event Structures

For our considerations, we now combine runs with time. We do this by enriching the set $Move$ with a (partial) time order and with a duration distance.

Definition 2.9. (Timed event structure)
A *Timed event structure* is a tuple $TES = (Move, <_f, \#, \leq_t, dist)$ where

1. $ES = (Move, <_f, \#)$ is an event structure,
2. $T = (Move, \leq_t, dist)$ is a time domain with a distance, and
3. $<_f \subseteq \leq_t$ (consistency condition).

This definition allows the time order $\leq_t$ of an event structure to be weaker than the causal order $<_f$, but not to contradict it. This rules out so-called non-causal runs. We use this definition to also define a timed run $R = (M, <_f, \leq_t)$ similar to an untimed one.

Notice that the way of building timed event structures by adding distances, or sets of possible distances between events, is also used in other approaches (as representative examples see [48, 9], where some alternative representation of event structures is chosen). Our approach differs from the cited ones by the fact that we allow the distinction between the causal order and the time order. This explicit distinction allows handling partially ordered time domains and discussing the consistency conditions in Section 3.

Now we define sequentialisations of timed runs.

Definition 2.10. (Correctly timed sequentialisation)
A timed run $R' = (M, <'_f, \leq'_t)$ is a *correctly timed sequentialisation* of $R = (M, <_f, \leq_t)$ over a timed event structure $TES = (Move, <_f, \#, \leq_t, dist)$, if it has the following properties:

- R' is an untimed sequentialisation of R according to Definition 2.4
- $TES' = (M, <'_f, \#, \leq'_t, dist)$ is a timed event structure (with an empty conflict relation)
- $\leq'_t \supseteq \leq_t$

An untimed run may have inconsistent timed sequentialisations. However, it is always possible to sequentialise a timed run.

Proposition 2.1. (Properties of sequentialisations)
(a) An untimed run of (the untimed part of) a timed event structure has at least one correctly timed sequentialisation.

(b) If $<_t \subseteq <_f$ then all sequentialisations $R' = (M, <'_f, \leq'_t)$ of $R = (M, <_f, \leq_t)$ are correctly timed sequentialisations.

Proof sketch: (a) is due to the fact that $<_f \subseteq \leq_t$. This implies that at least one extension of $<_t$ must be consistent with the (total) order imposed by $<'_f$.
(b) is due to the fact that $<_t$ does not introduce more dependencies than $<_f$. □

We do not need any new consistency conditions concerning the distance function. If the considered time domain $T = (Move, <_t, dist)$ is a time domain with a distance, then this is the case for every consistent run or sequentialisation.

2.4. Associating Time with Computations

In the definition of a timed event structure, we interpret the set of moves of an event structure (or similar for a state structure) as a set of time points defining a time domain by means of a (partial) time order and possibly a distance on them.

An equivalent option is to say how, for some (standard) time domain $Time$, to attach time points in $Time$ to every move, as shown below.

Definition 2.11. If $ES = (Move, <_f, \#)$ is an event structure and $T = (Time, <_{time}, dist_{time})$ is a time domain with a distance, then a function $time : Move \mapsto Time$ is a *timing function* for ES, if $TES = (Move, <_f, \#, \leq_t, dist)$ defined by

- $m_1 <_t m_2 \iff time(m_1) <_{time} time(m_2)$ and $m_1 =_t m_2 \iff m_1 = m_2$
- $dist(m_1, m_2) = dist_{time}(time(m_1), time(m_2))$

is a timed event structure.

The approach for introducing time in ASM, proposed in [35], is different. It defines a mapping from a time domain to states (a mapping to events would not be very different) which, by definition, imposes a stronger consistency condition between the causal order and the time order: in particular, the fact that there is a function from $Time$ to $State : Exp$ forbids to have several state changes (that is causally related events) at the same time point. Nevertheless, using the approach of [35] for a partially ordered time domain in which unordered time points are interpreted as being *quasi synchronous*, or alternatively, by using a refined time domain that allows to insert enough time points so as to associate a different time to any sequence of events originally at the same time point, as it is done in [64], leads to modelling frameworks which are equivalent to ours from the point of view of the expressiveness.

When a function $time$ as defined above is used, then there is generally no need to have several time points at distance zero, but one will associate the same time point with events taking place at "the same time". But when the move domain itself is chosen as time domain, moves taking place "at the same time" are to be considered "distinct time points" at distance zero. There is an implied equivalence relation between time points at distance zero which allows to bring us back to a time domain without zero distance between non-equal time points. This is guaranteed by the following lemma.

Lemma 2.3. (zero distance implies equivalence)
$\forall t, t', t'' \in Time \,.\, dist(t, t') = 0 \implies dist(t, t'') = dist(t', t'')$

Proof: From the triangular inequality we get $dist(t, t'') \leq_d dist(t, t') + dist(t', t'') = dist(t', t'')$ and $dist(t', t'') \leq_d dist(t', t) + dist(t, t'') = dist(t, t'')$. □

Our proposed concrete approach for handling time in ASM discussed in Section 4 allows both alternatives for associating time with events.

2.5. State Structures versus Event Structures

In this section, we discuss an alternative to the framework defined above. It is mainly referring to results on event structures used to discuss the relationship between a state-based and an event-based approach.

An alternative to the use of an event structure is the use of a state structure for the representation of the semantics for a state-based formalism such as ASM. But in fact, either of these structures can be transformed into the other one. We prefer the event-based view of a system because it allows a direct representation of concurrency by distinguishing ordered, unordered and conflicting events.

States are either ordered or conflicting (alternatives); there is no notion of partial order between (global) states. Notice that it has been shown in the work on *asynchronous transition systems* [65, 13] how to extract the information on concurrency from a state-based transition system. In the context of model-checking, partial order reduction methods [33, 59] are used, requiring to derive the concurrency of transitions by structural or dynamic analysis.

These results can be used to transform a transition system $TS = (State, Move, \rightarrow)$ such that each label in $Move$ occurs at most once in any maximal path through TS into an equivalent event structure $ES = (Move, <_f, \#)$.

2.6. Examples of Time Domains

So far, we have defined what a timed event structure is and what kind of properties it has to satisfy, but we have not given any evidence that such time domains even exist. Below, we discuss two examples of such partially ordered time domains.

Example 2.1. Let us consider a time domain defined by n real lines, representing the time in n locations, such that each time line is cut into half open intervals $I_{ij} = [t_{ij}, t_{ij+1})$, as suggested in Figure 1, and choose $Duration = \mathbb{R}$. For $n = 2$, we have $Time = Loc \times \mathbb{R}$ with $Loc = \{1, 2\}$.

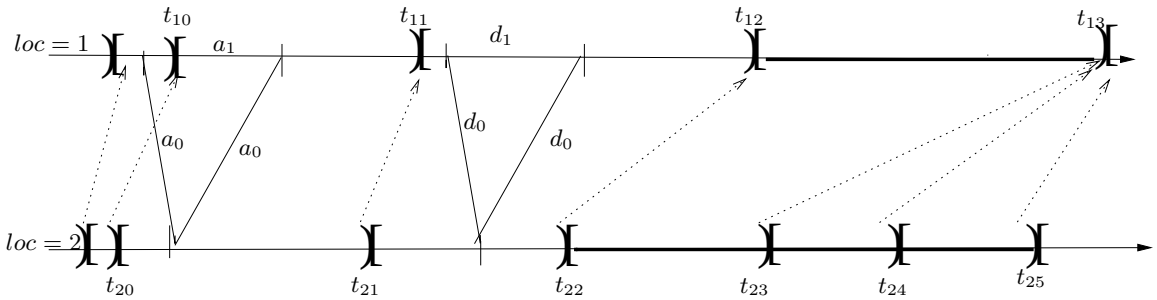

Figure 1. A partially ordered time domain defined by intervals on 2 real lines

$\forall i, l \in Loc \forall t, t' \in \mathbb{R}$, the relation $\leq_t$ is defined as the weakest preorder such that $(i, t) \leq_t (l, t')$ is implied by

- $\exists j, k \,.\, t \in I_{ij} \wedge t' \in I_{lk} \wedge t_{ij+1} \leq t_{lk}$, that is, points are ordered according to their real value if their respective intervals have a non-overlapping real part, otherwise they are unordered.

Thus, we have $I <_t J$ if all real values of I are before all real values of J, other intervals (and all their time points) are unordered, which may be interpreted as "approximately at the same time". Notice that the unordered relation is generally not transitive.

Similarly, the distance $dist$ between time points depends only of the bounds of the intervals they are in. We consider several cases.

1. Choosing the distance between points in distinct intervals I_{ij} and I_{lk} uniformly as the difference between the real part of their offset $dist(I_{ij}, I_{lk}) = |t_{ij} - t_{lk}|$ and the distance between points in the same interval as 0 leads to a natural extension of the synchronous approach to quantitative time, that is, the distance between points is that of the starting points of their time slices and all points in the same time slice have distance 0.

   Notice that choosing the distance between any to timely unordered points as 0 is not a solution, as it may violate the triangular inequality as indicated in Figure 1 by the points with distances $a_0 = 0$ and $a_1 > 0$.

2. A distance in which all points within an interval are equidistant for a strictly positive value, can be defined by taking as distance of any two points the maximal difference of the real values of their respective intervals. This distance represents an over approximation of the (usual) distance between concrete points. Therefore, it can be easily refined in a classical manner at a later point of time.

Example 2.2. The second example shows a time domain used in the context of testing of distributed systems, where transmission times and the relationships between clocks in different locations are supposed to be unknown (see [24, 31] for seminal papers on this issue): time can be measured and compared with arbitrary precision only on individual locations, whereas between events on distinct locations, time precedence can only be established for causally ordered events, where causally related events are determined exclusively through message flows, as illustrated in Figure 2.

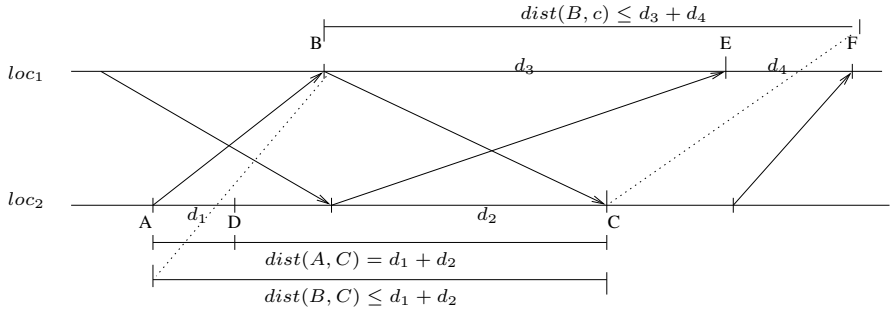

Figure 2. Precedence and distances in test of distributed systems

That means, the time domain is as in the first example of the form $Time = Loc \times \mathbb{R}$, that is, in each location time is represented, as usually, by reals[4]. Then, given the causal order $<_f$ transposed to $Time$, the preorder on time and distances are defined as follows:

- the partial order on time is the least partial order satisfying
  - $(l_1, t_1) <_t (l_2, t_2)$ if $l_1 = l_2 \wedge t_1 < t_2$ or if $l_1 \neq l_2 \wedge t_1 <_f t_2$
- the distance on $Time$ is for each pair of time points the best approximation that can be obtained by the following constraints, that is the minimal value that satisfies all inequations that can be obtained for the pair of time points by means of constraints constructed as below:
  - $\forall e_i = (l, t_i) \in Time \,.\, dist(e_1, e_2) = |t_1 - t_2|$
  - $(l_1, t_1) <_t (l_2, t_2) <_t (l_1, t_3)$ implies $dist((l_1, t_1), (l_2, t_2)) \leq_d dist((l_1, t_1), (l_1, t_3))$

It is easy to see that $<_t$ is an order on time that is compatible with the causal order. The function $dist$ is a metric that satisfies requirements (t1) to (t4), but is satisfies even two stronger constraints.

(a) the distance between unordered time points (in different locations) is smaller than the minimal distance to any common related time point, that is

$$\forall e_1, e_2, e \in Time \,.\, (e_1, e_2 \text{ are unordered } \wedge e_1 * e \wedge e_2 * e \implies dist(e_1, e_2) \leq_d dist(e_i, e))$$

where $*$ stands uniformly either for $<_t$ or for $<_t^{-1}$.

(b) all time points that are not ordered with respect to a third one, have the same distance to it, that is,

$$\forall e_1, e_2, e \in Time \,.\, (e_1, e), (e_2, e) \text{ are unordered } \implies dist(e_1, e) = dist(e_2, e)$$

[4]Taking natural numbers wouldn't change anything for our purpose.

3. Timed Frameworks defined by a Set of Properties

So far, we have introduced a notion of timed event structure, including a notion of distance. The consistency conditions for a timed model that we have defined so far, are the weak consistency between time order and causal order ($<_f \subseteq \leq_t$) from Definition 2.9 and the properties inherited either from properties of runs (e.g. that they are a maximal state) or from the fact that a timed event structure is a time domain (e.g. that the distance from the start point must (weakly) increase).

In this section, we discuss more properties specific to timed computations and define compatibility conditions between time and computations. This is similar as in the framework of temporal logic where some axioms are used in (almost) all frameworks and others may or may not be included in an axiom system.

We focus on three classes of properties which play an important role in existing frameworks. As for temporal logic axiomatisations, any given framework can always be refined to a consistent framework by means of new axioms. This section discusses properties that are underlying either in the basic framework or the associated verification algorithms of many frameworks. This should also demonstrate the adequacy of our semantic model for expressing such constraints conveniently.

In Section 3.1, properties concerning the relationship between time and causal order are discussed which distinguish global and distributed frameworks. Section 3.2 contains properties of the distance function; the properties presented refine the notion of *correctly timed computation* defining criteria used in the work on implementability of specifications. Sections 3.3 discusses immediate reaction at the semantic level and defines criteria for move selection which may be used to define urgency. The properties of Sections 3.2 and 3.3 do not necessarily restrict the event structure under study, but rather the set of legal runs based on the relation $\#$ representing (conflicting) alternatives.

In the following, if not otherwise stated, we suppose $TES = (Move, <_f, \#, \leq_t, dist)$ to represent an event structure with a distance, $R = (M, <_f, <_t,)$ an arbitrary timed run of it, and we consider that TES has a given property if all its runs satisfy it.

3.1. Additional Time Ordering Constraints

The properties in this subsection provide additional constraints between the causal order and the time order, strengthening those defined in Section 2.3.

Property 1.1. (Global Time)
In every run, the weak timing relation $\leq_t$ is a total order.

This strengthening of the relation $<_t$ expresses the fact that all events are ordered with respect to some global notion of time, but several events may occur at the same time point. Also, when global time is considered, distances are in general considered to be additive.

Property 1.2. (Additive Time)
(t3') $\forall m_1, m_2, m_3 \in Move \,.\, m_1 \leq_t m_2 \leq_t m_3 \implies dist(m_1, m_3) = dist(m_1, m_2) + dist(m_2, m_3)$

The above two additional properties characterise the basic time progress model of the timing frameworks underlying timed automata, Petrinets, SDL and most real-time programming languages. In the second time domain of Section 2.6, distances between comparable time points are additive.

Notice that additivity may also be stated for unrelated time points, then implying that the distance between time equivalent points must be zero, which is the reverse direction of Lemma 2.3.

A different strengthening of the $<_t$ relation concerns the connection between the causal order and the time order, forbidding reaction chains in zero time.

Property 1.3. (Strict time progress)
Strict time progress along causal chains: whenever two moves are causally ordered, their occurrence times are strictly ordered, i.e.,

$$<_f \subseteq <_t \ .$$

This property, combined with the previous one, is underlying languages for the description of synchronous circuits, such as VHDL [8] or in the framework defined in [64] for ASM. The same is true for the framework proposed in [35], except that there two (but not more) consecutive events can take place at the same time (see also Section 2.4).

An even stricter property that has already been seen in Proposition 2.1 and in the second example of Section 2.6, is the following one.

Property 1.4. (Timed order is not stronger than causal order)
In every run, causally non related events are not comparable in the timed order, i.e.,

$$<_t \subseteq <_f \ .$$

Together with strict progress (Property 1.3), this implies $<_t = <_f$ which means that one of the two order relations is redundant.

As explained earlier, Property 1.4 is used in the context of testing of distributed systems where time can be measured and compared only on individual locations, whereas between distinct locations time precedence can only be established for causally ordered events, and it is used there together with Property 1.3. Distances are constrained by Property 1.2 (additivity) and the constraints on distances mentioned in Example 2.2, refining the general constraints on distances.

A last restriction on the relationship between the different ingredients that we consider states that, conflicting moves must be comparable in time.

Property 1.5. (Conflicting moves must have comparable time points)
Any two moves that are in conflict, that is exclude each other, must be ordered in time, i.e.

$$\# \subseteq \leq_t \cup \leq_t^{-1}$$

This property trivially holds in all frameworks with global time. When time is not global, it states that only independent moves are allowed to have an undefined precedence in time. This property is satisfied by the second time domain in Section 2.6. If "localities" are threads, and timely uncomparable events correspond to events in different threads, then this property guarantees that timely uncomparable events can be scheduled in any order without breaking the atomicity of their "enclosing task" (otherwise, the events would be conflicting).

In the following section, we consider constraints on distances in infinite computations or minimal distances between events. While the properties in the Sections 2 and 3.1 restrict the set of allowed event structures, all the subsequent properties restrict, for a given event structure, the set of allowed runs.

3.2. Restricting the Number of Moves per Time Unit

In the case that we allow time staggering steps, that is, runs in which causal progress does not necessarily imply time progress, we have to add a requirement guaranteeing that a run with an infinite amount of steps runs until the *end of the time*. Similarly, if we consider a dense time domain $Time$, by the definition of a run, the set of moves in a run represent a countable subset of $Time$, but we still have to guarantee that time progresses "over all bounds".

This is sometimes done by means of a constraint of finite variability (see for example [42]), but finite variability cannot be used for all time domains satisfying the constraints (t1) to (t5) of Definition 2.7[5]. The following definition can be applied to any time domain satisfying the constraints of Definition 2.7.

The first property, generally referred to as non Zenoness property, describes that in an infinite run time values always converge to an upper bound of the values in the time domain which has the intuitive meaning that it is impossible to do an infinite number of computation steps in a finite amount of time.

As our computation model includes the notion of concurrency and local time, it is not necessary to require that a run has an accumulation point of time values. We only need to require that any ordered chain has a limit which is an upper bound of the time domain. As we do not require time domains with arithmetic properties, we express the same property in terms of distances, as follows.

Property 2.1. *Absence of Zeno computations*: Any maximal chain $C = \{m_j | j \in \mathbb{N} \wedge m_j <_f m_{j+1}\}$, subset of a run R, has a well defined limit $lim_{n\to\infty} dist(m_0, m_n) = D_u$ (the upper bound of $Duration$):

$$D_C = \{d_j = dist(m_0, m_j) | j \in \mathbb{N}\} \text{ has the limit } D_u$$

Notice that due to the point (3) of Definition 2.9 and the triangular inequality, D_C is an (at least weakly) increasing chain with respect to the total order $<_d$ and has therefore a limit. Non Zenoness means that this limit must be the least upper bound of the Duration domain (∞ when $Duration$ is taken as $\mathbb{N}$ or $\mathbb{R}$).

When we also consider time points to be numbers (reals or integers) with their standard distance, then it means that time and distances must be diverging for any causally ordered set of moves.

In formalisms forcing time progress, non Zenoness is satisfied by construction. But in practise, only formalisms with discrete time progress do that. In formalisms like VHDL [8], or in formalisms allowing zero time steps, the usual method consists in showing that reachability properties can be satisfied on non Zeno executions. Alternative characterisations of non Zenoness have been proposed, adapted for implicit timing, e.g. one reducing non Zenoness to the requirement that time progress by any fixed δ must always be possible [41]. In the context of timed automata, sufficient syntactic conditions on rules have been given which guarantee the absence of Zeno computations [69]; moreover, verification methods exist for checking that all finite prefixes can be extended to non Zeno computations [70].

If the underlying time domain allows for diverging distances, we can refine the above requirement to exclude that time progress may slow down arbitrarily. For example, consider reals as the time and

[5]For instance, the real domain can be projected onto the interval $(0, 1)$ by preserving (t1) to (t5), but not finite variability.

duration domain and the run defined by a set of moves occurring at time points t_n, with $t_{n+1} = t_n + \frac{1}{n}$. Then, the timepoints diverge but the time progress in n computation steps tends towards 0.

The requirement of absence of such sequences expresses a necessary condition for the implementability of a system, at least if the ratio of moves representing "environment steps" is bounded. In the formulation below, we make no explicit distinction between system and environment steps, which can be done easily by introducing an appropriate classification of events.

Property 2.2. (Bounded variability)
Bounded number of computation steps in bounded time: for every move, the number of moves that are at a time distance that is smaller than some arbitrary finite d is uniformly bounded in every run $R = (M, <_f, \leq_t)$.

$$\forall d \in Duration\,.\,\forall m \in M\ \exists N \in \mathbb{N}\ |\{m' \in M \mid dist(m, m') \leq_d d\}| \leq N.$$

Proposition 3.1. Property 2.2 implies Property 2.1, but not conversly.

Proof sketch: The first implication is trivial, and above we give a counter example of the opposit implication, that is, a run in which any finite time interval contains only a finite number of moves, but with time progress, this number grows over all bounds. □

This is the qualitative property underlying the implementability guarantee in the context of the synchronous approach. In the context of synchronous hardware [8] or synchronous programs [15, 23], each basic cycle is required to have, not just a finite, but a bounded number of steps.

Notice, however, that Property 2.2 does not exclude the existence of moves that are arbitrarily close to each other which is excluded in the following property.

A frequently used time model is discrete time. From the qualitative point of view any run is already such a discrete time domain by definition. For an appropriate notion of distance, adding quantitative time allows to cut runs into time slices, such that all steps in the same slice have a distance 0 and moves n slices apart have distance n as given below.

Property 2.3. *Events at discrete steps*: Any two moves have a distance that is the multiple of some fixed value, which is the same for all runs.

$$\exists \delta \in Duration\ \forall m_1, m_2 \in Move \exists k \in N.\neg(m_2 \# m_1) \implies dist(m_1, m_2) = k * \delta.$$

Proposition 3.2. Property 2.3 and Property 2.2 are uncomparable. □

Synchronous models and related approaches, such as B [1], as well as engineering approaches used in the context of testing [30] or real-time embedded systems [44, 43, 29] use both of the last two properties for models representing a global system level view. This property is trivially satisfied when the time domain are natural numbers.

Finally, notice that some properties related to real-time computations mentioned in the literature make little sense in a discrete event-based setting. As an example take a property called *strong retrospection* in [68] requiring that any two runs coinciding on a time interval of the form $[0, t)$ must also coincide on $[0, t]$. In an event-based model, any time point t may be the first instant for which two discrete event sequences are different. Such a property may be useful for constraining continuous environments or in the approach of [35] when the time intervals defining a state are supposed right closed.

A state machine or a set of concurrent state machines specifies an (abstract) transition relation, and represents a safety property according to the characterisation given in [2]. Liveness properties, specifying that some infinite sequences are not allowed, even if all its prefixes can be extended to allowed sequences, cannot be expressed by a transition relation alone. The Temporal Logic of Actions (TLA) describes systems by means of a transition relation (next relation) and a set of fairness constraints enforcing liveness. Absence of Zeno computations is a typical example of a liveness property: time must progress, eventually. Timed automata do not allow to express this property: when timed automata are used to describe timed systems, non Zenoness is used as an external property that is taken into account universally for all specifications in the proposed verification algorithms.

3.3. Maximal Progress and Urgency at Semantic Level

The expression of maximal time distances between events and more generally, taking into account relevant changes in the environment is essential for ensuring correct system progress and for reasoning about real-time computations. In some modelling formalisms, for example all synchronous languages, B and Statemate, a computation is defined as a strict alternation of time and environment steps, where after each environment step (which realizes also time progress) the system must react as defined by the system specification (which is often supposed to be deterministic), but the system cannot "miss its turn".

In other formalisms, for example process algebras or other formalisms based on the asynchronous (interleaving) composition of state machines, a move has a "guard" indicating when it is "enabled", that is, *may happen* as a next step. Fairness constraints can be used to ensure that a component that is permanently enabled, will do its move finally. But a component may be enabled at some point of time, and due to the fact that other components (or the environment, including time) "move first", it may get disabled again before getting the opportunity of being executed.

The use of a partial order semantics clarifies the underlying problems. A permanently enabled move m is part of the considered run and therefore must be executed, whereas a move m' that can get disabled by a move m'' before being executed is represented explicitly by the fact that m' and m'' are conflicting moves ($m' \# m''$). It is then not allowed to have both of them in the same run.

Notice that the framework we have so far, does not allow to require that either m' or m'' *must* be present in any run. This is done with the properties below. First we start with an example for the use of urgencies.

Let us consider two agents A_1 and A_2 as in Figure 3. This event structure allows two runs which contain either m_{11} and m_{22} or m_{21} and m_{12} with time distances to m_{i0} as indicated by the constraints in the figure; therefore, each m_{ij} represents a set of events, one for each possible time point. Requiring the *enforcement of upper bounds* means operationally that there is no execution in which, e.g. m_{11} is enabled but time progresses beyond the upper bound of its enabledness condition, and an alternative move is then taken at a later point of time. In the example, this implies that the run containing m_{21} is only allowed when the occurrence time of m_{20} is less than 2 time units after the one of m_{10}. That means basically that, the decision between the two alternatives has to be taken before one of the alternatives times out. Requiring *maximal progress* means that each event occurs at the earliest possible point in time satisfying all the constraints. Such constraints can be expressed by splitting up the existing event structure until a set of runs with more refined time constraints remains.

We however, want to be able to define such a run selection *without* changing the event structure, and even more importantly without changing the "ASM program level" (see Section 4.2.3).

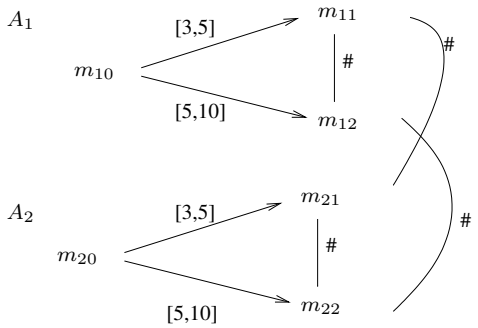

Figure 3. Illustration of urgency

In Petrinets [71], upper bounds of time constraints of all events, that is transition executions, are always enforced. In timed automata, explicit notions of *urgency* have been introduced in the form of invariants [5, 38] or urgency attributes of transitions [22, 21] which allow the definition of other interpretations: if a move m is *eager* then m must occur at the earliest time point at which it is "enabled", from the point of view of the causal and the time constraints. When m is delayable, then an alternative can be chosen if it occurs at a time point at which m is possible as well, and finally, when m is *lazy*, then some alternative may be chosen even if this alternative must occur later than m.

Other formalisms make other implicit choices: e.g., timed versions of process algebras [16, 45] consider all (maximal) synchronisation actions and only those as urgent. The SDL standard semantics does not enforce any urgency but all tools implementing the language use minimal time progress, that is, urgency of all system actions. In declarative formalisms such as TLA, urgency can be enforced by adding constraints enabling time progress — an explicit *tick* action — only when no urgent transition is enabled.

A restricted form of urgency is *immediate reaction* which says that an action must be executed at the time point at which it becomes causally enabled.

Property 3.1. *Immediate reaction of a move in a run*: In any given run, a move $m \in M$ is in immediate reaction if it occurs at the same time as at least one of it's causal predecessors.

$$\exists m' \in M \,.\, m' <_f m \wedge dist(m', m) = 0$$

Property 3.1 is incompatible with Property 1.3. For this reason, HASM [64] introduces a duration δ which is smaller than any standard positive real and defines $dist(t + \delta, t) = 0$. In [35], immediate reaction chains are possible, but at most of length two.

Property 3.1 makes sure that an immediate move m appears at the correct point of time under two conditions:

- *if* reaching the time point enabling m is represented explicitly as a move. This is the case in time triggered systems, but we prefer not to impose this as a general principle;

- *and if it appears at all* in the run. That is, we still have to make sure that immediate moves must be there.

In order to claim that certain moves cannot be missed or must be executed at the earliest possible time point, we must express that an enabled action must be taken, or some allowed alternative must occur. At the semantic level, we define a set of moves representing the choice of alternatives that must be included in the run. In order to do so, we first introduce some notations.

Definition 3.1. (enabled moves, successor moves, alternative moves)
For a timed event structure TES as given before

1. Denote the set of *enabled* moves in a state $s \in States(TES)$ by

$$en(s) = \{m \in Move \mid s \cup \{m\} \in States(TES)\}$$

2. Denote by $succ_R(s)$ the successors of a state s in a timed run $R = (M, <_f, \leq_t)$, that is

$$succ_R(s) = en(s) \cap M$$

that is the minimal moves not already contained in s that are in R.

3. Let MS be a set of moves and $R = (M, <_f, \leq_t)$ a timed run. Denote by $Alt_R(MS)$ the moves in R which are alternatives of some move in MS in some state *instead of* (alternative to) an enabled move m in MS. That is

$$Alt_R(MS) = \{m \in M | \exists s \in States(R)\, \exists m' \in MS \cap en(s)\,.\, m \in succ_R(s) \wedge m' \# m\}$$

That is, $Alt_R(MS)$ is the set of moves of R which "avoid" enabled moves in MS.

Now, we can express the fact that an action, represented by some set of moves MS is

- *delayable*: that is, any run in which a move m in MS is enabled in some state can only contain alternatives of m that occur not later than some enabled move in MS.
- *eager*: that is, any run in which a move m in MS is enabled in some state can only contain alternatives of m that occur not later than the earliest move in MS. In addition, if MS is not avoided then, the chosen move occurs at the earliest time point at which a move in MS is enabled in the considered run.

Property 3.2. A set of moves MS is *delayable* in run R if

$$\forall s \in States(R)\,.\,(m \in succ_R(s) \cap Alt_R(MS) \implies \exists m' \in en(s) \cap MS\,.\,\neg(m >_t m'))$$

Property 3.3. A set of moves MS is *eager* in run R if

$$\forall s \in States(R)\,.\,(m \in succ_R(s) \cap Alt_R(MS) \implies \forall m' \in en(s) \cap MS\,.\,\neg(m >_t m'))$$

Note that we require $\neg(m >_t m')$ rather than $m \leq_t m'$ as it is possible that the m and m' are not comparable when interpreted as time points. When Property 1.5 holds, in particular when time is totally ordered, both conditions coincide.

Requiring all updates to be *eager* excludes time constraints of the form "m takes place at time $t > 2$", because the open enabling interval has no minimum.

The notion of *eagerness* has an advantage over the notion of *immediateness* of Property 3.1; it can also be used when Property 1.3 holds as it only requires that, amongst a choice of possible moves, one with an earliest occurrence time must be chosen. The notion of *eagerness* according to Property 3.3 is based on the assumption that, if the time distance to the previous move must be zero, then there exists at least one move in the event structure that allows to satisfy this constraint. It is easy to see that, whenever all sets $succ_R(s) \cap MS$ are singletons, the Properties 3.3 and 3.2 coincide. On the other hand, when even arbitrary time progress cannot disable a set MS, then MS is delayable in any run.

The difference between the properties just presented and those in the previous section is that they are parameterised by sets of moves. Some modelling frameworks require immediate reaction as a general principle. For example, the existing SDL tools do this as a means to make time progress deterministic. The idea of immediate reaction is also underlying the time-triggered and synchronous approaches and their variants.

4. A Proposal for Representing and Handling Time in ASM

In this section, we propose a way for introducing time to ASM. This is just one possibility, and there have been several other proposals. We think, however, that our proposal allows a very general, yet intuitive handling of time at specification level.

4.1. An Overview of ASM and their Computation Model

Abstract State Machines (ASM) [34, 20] were introduced as a general computation model taking concurrency into account explicitly. The ASM definition provides a sequential version which is then extended to the general, distributed case.

A *sequential* ASM algorithm (see [20], page 72) is described by a set of rules applied to states. Although the notion of state in ASM is very general – a state is an algebra and a rule describes a transformation between algebras – we consider for the purpose of our discussion that a state is defined by the values of a set of locations, which can be understood as (programming) variables, i.e. nullary functions.

The notion of "atomic step" (called a *move*) is defined by means of a (arbitrarily complex) rule for assigning new values to some locations, depending on the old state values. In order to model inputs and non-determinism, some locations are designated to be "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* (see [20], page 34). That means, the behaviour 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 by the set of possible moves. The semantics of the system is given as the set of possible *executions* (see [20], page 75), 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 (see [20], page 208), there exist several sequential 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 given by a controlled function $program$. The semantics of distributed ASMs is given by a set of *partially ordered runs*, where each of these runs consists of a partially ordered set of moves and each

move having a finite number of predecessors. 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). This coherence condition implies that each sequentialisation of the partially ordered run yields the same result, and vice versa. The set of executions is the set of state sequences induced by all possible linearisations of all runs. In order to distinguish between agents, each agent gets a reference to itself using a monitored function $Self$ where the environment assigns the agents that is making the move. In ASM, each agent has "its rule" which can be applied to any state independently of any other agent or the changes of the environment. But, sometimes the application of the rule does not change the state (empty update set). We consider these rules to be *guarded rules*, representing partial functions from global states to global states, and consider only runs without such empty moves.

This means that the set of runs defined by a specification corresponds to an event structure, where the events are the set of all possible updates, and the definition of the relations $<_f$ and $\#$ is straightforward.

4.2. Adding Time to ASM

Introducing time to ASM, we want to be able to handle all possible frameworks discussed in Sections 2 and 3. We want to be able to handle any time domain that has been identified as such (see Paragraph 4.2.1) and enforce a connection between environment and program (see Paragraph 4.2.2), as well as ways to set general time requirements (amongst those described in Paragraph 4.2.1) and application-dependent time requirements (see Paragraph 4.2.3). Moreover, we introduce a few useful abbreviations in Paragraph 4.2.4. This allows expressing timing properties directly in the specifications without explicitly referring to runs.

4.2.1. Time Domains

An ASM specification defines naturally an event structure $ES = (Move, <_f, \#)$, a timed specification has to define $<_t$ and $dist$ on $Move$.

Our proposal includes the possibility to explicitly handle only time distances by specifying constraints on distances between named moves. In this case, only a $Duration$ domain needs to be provided, and $<_t$ becomes a deduced relation from $dist$, $<_f$ and the choice of basic properties. The $Time$ domain is in this case the same as the $Move$ domain.

The second way of handling time we support, consists in using the alternative definition of a timed event structure of Section 2.4, in which a general time concept — with domains $Time$ and $Duration$, together with their associated relation $<_t$ and distance $dist$ — is used to associate time stamps with moves. In this case, we suppose the time concept to be provided by a set of auxiliary definitions including a set of properties for the notion of time. The properties of $T = (Time, <_t, dist)$ can then be checked a priori and independently of a particular specification. These definitions may well map the $Time$ and $Duration$ domains to some standard domain, such as positive $Real$ with their standard order relation and difference as distance.

The conjunction of the properties chosen out of those of Sections 2 and 3 are then used as a part of the constraint on the environment to enforce time progress to be consistent with the framework choices.

4.2.2. Reading Time

When time is provided as a separate concept, one needs to define how the environment and the specification are connected. We do this by introducing a monitored function to access the time of a move.

$$\mathsf{monitored}\ now : Time$$

This function seems to imply that the time is stored in the state, which seems strange since we attach times to moves. However, ASM describe state changes rather than states. This means that, particularly for monitored functions, the values are only available within state changes. Consequently, each access to the function now means an access to the time function of the corresponding state change (move). Thus, the interpretation of now is fixed to be the value of the $time$ function (defined in Section 2.4) of the corresponding state change. A similar approach is already used in ASM when the actor of a state change is determined using the function $Self$ (see [20], page 210).

Using this function now, it is possible to express any constraints on the occurrence time of or the distance between events. It does also allow to read and store the occurrence time of an event in a variable for later comparison.

4.2.3. Urgency

The urgency properties given in Section 3.3 are parametrised by a set of moves. That means, they are denoting application-specific constraints. We propose to use them via a controlled function which can be set by any agent.

$$\mathsf{controlled}\ urgency : \{eager, delayable, lazy\}$$

The new value of this function in a move is used to define two sets *Delayable*, collecting the delayable moves, and *Eager*, collecting the eager moves. These sets are then used as the parameter MS of Property 3.2 (requiring the move set *Delayable* to be delayable) and Property 3.3 (requiring the move set *Eager* to be eager) which are joined to the environment constraints on time progress. For $lazy$ moves no additional constraint is needed.

As an abbreviation, agents can be declared to be eager or delayable, meaning that all their moves have this urgency. Notice that in [22] it is shown that at least in the context of global time, these three urgency types together with guards and framework conditions are sufficient to express any kind of time progress law.

4.2.4. Some Useful Abbreviations

In this section, we introduce some abbreviations solving common specification problems. We introduce local clocks and occurrence times of events, durative updates and occurrence times of state changes.

Local Clocks and Storing Time In order to access the occurrence time of any causally previous event, it is just necessary to store the value of now in the event. Using these stored times, also local clocks as introduced by timed automata, can be implemented. To introduce this in ASM, we first define a domain $TimeStamp$ with a timing function on it. These time stamps can represent local clocks,

where we just have to define an operation for reading the clock (time stamp) and for resetting it.

$$\begin{aligned}&\mathsf{domain}\ TimeStamp\\&\mathsf{controlled}\ time : TimeStamp \mapsto Time\\&reset(ts) =_{def} ts := now\\&\mathsf{derived}\ clockValue(ts) =_{def} dist(ts, now)\end{aligned}$$

For an easy use of this functionality, we propose to introduce labels before statements, meaning a) the declaration of a $TimeStamp$ and b) an assignment of the actual time whenever this statement is "executed", e.g. $time(label) := now$.

Times of Value Changes It is often necessary to refer to the occurrence time of a *value change* of a location. Of course, this can be done by storing time, whenever the location is changed. In order to allow simpler access to changes and also to allow keeping track of environment changes, we propose to use the following function whose result is the time point of the last change of the location provided as parameter.

$$\mathsf{monitored}\ changeTime : Location \mapsto Time$$

This function is used in the example in Section 5.

Updates Taking Time In timed systems, it is often necessary to express that some actions take a certain amount of time. In the semantic model, moves are instantaneous and do not take time. This means durative moves have to be represented by two separate moves with an enclosed intermediate state in which time passes. This way, we think of a durative move meaning that the state is read at some time t_1, and the new state is written at some later time t_2.

At the designer level, we propose to use the following notation: $\mathbf{readyAt} < expr >$ to express this behaviour. This notation implies two state changes:

1. computing the update set using the current state and disabling the current agent, and
2. at time <expr> enabling the agent and applying the precomputed update set. This guarantees the atomicity of the move, that is, other agents or the environment cannot influence the result of the move by changes while the agent is in its wait state.

5. Example

In order to show how the proposal of the previous section works in practice, we apply it to the Generalized Railroad Crossing Problem presented in [35] which has become a kind of standard toy example for ASM. In fact, we rather refer to the refined version presented in [11] from which Figure 4 is taken. This problem handles a railroad crossing with several train tracks and a common gate. There are sensors on every track detecting incoming and departing trains,

Based on the signals from the sensors, an automatic controller signals the gate to open and to close.

This problem should be fairly well known and we just sketch it here. The problem is to model a railroad crossing with a finite number of tracks with trains on them. Initially, there are no trains and the

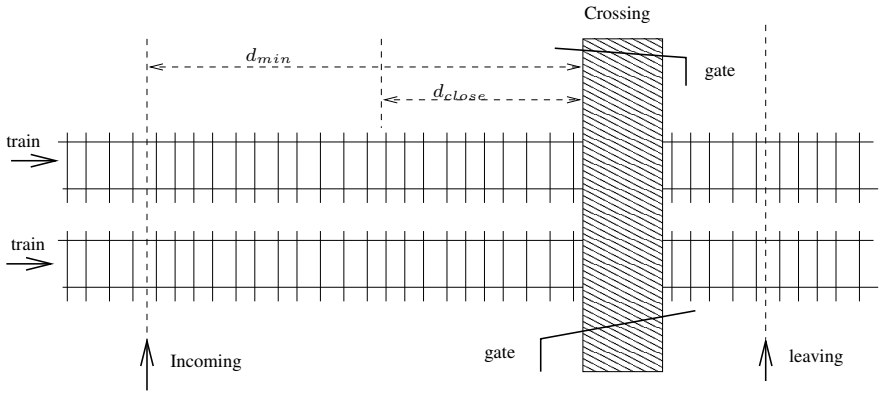

Figure 4. The Gate controller

gate is open. There are several actions that take time: closing the gates takes at most d_{close}, opening the gate takes at most d_{open}, and each train takes between d_{min} and d_{max} from the sensor to the crossing. There are some more conditions that are needed to prove that the given implementation is safe, but these are not needed for our specification.

In this paper, we are not (yet) interested in validating the safety and liveness aspects of this problem, but in specifying timing in algorithms or systems. We start by giving the solution from [11] below, modulo some notational differences, and then show how to add timing constraints.

agent GATE
  if Dir = open **then** GateStatus:= opened **endif**
  if Dir = close **then** GateStatus:= closed **endif**

agent CONTROLLER
  forall x **in** Tracks
    if TrackStatus(x) = coming **and** Deadline(x) = ∞ **then** Deadline(x):= Now+WaitTime **endif**
    if TrackStatus(x) = empty **and** Deadline(x) $< \infty$ **then** Deadline(x):= ∞ **endif**
  endforall
  if Dir = open **and** $\neg$ SafeToOpen **then** Dir:= close **endif**
  if Dir = close **and** SafeToOpen **then** Dir:= open **endif**

Here SafeToOpen stands for $\forall x \in Tracks.TrackStatus(x) = empty \vee now + d_{open} < Deadline(x)$ and WaitTime is an abbreviation for $d_{min} - d_{close}$. This original version is not complete yet. It needs in addition some constraints on the behaviour of the environment, as given below.

1. Tracks change their status in the order $empty \mapsto coming \mapsto inCrossing \mapsto empty$, where the time spent in state $coming$ should be between d_{min} and d_{max}. Moreover, a Track can not stay forever in the state $inCrossing$. The duration of the state $empty$ is not restricted at all.
2. the Controller agent is eager.
3. Gate is a bounded agent, i.e., whenever it is continuously enabled for some time it must finally be activated. The maximal delays are d_{close} for closing and d_{open} for opening.

Notice that these environment conditions are given in the original paper by statements on possible runs and the allowed sequence of states and corresponding time points. Using our proposal, these environment conditions can be stated directly in the specification – without leaving the ASM framework. Moreover, the specification can be slightly simplified as timeouts need not anymore to be set explicitly, but one may use labels (e.g. *enter* and *leave*) and the function $ChangeTime$ (e.g. of Dir) instead.

delayable agent GATE
if Dir = open **and** dist(ChangeTime(Dir),now) $\leq d_{open}$ **then** GateStatus:= opened **endif**
if Dir = close **and** dist(ChangeTime(Dir),now) $\leq d_{close}$ **then** GateStatus:= closed **endif**

The semantics of delayable is exactly as we need it: the agent can choose when to fire, but it must fire before time progress disables an enabled guard; this corresponds to the environment condition 3 (boundedness of Gate agent).

For handling the condition 2 (eager Controller agent), we just have to make the controller eager. In order to make the specification easier, we can refer to the track status change times instead of explicitly setting them.

eager agent CONTROLLER
if Dir = Open **and** ¬ SafeToOpen **then** Dir:= close **endif**
if Dir = Close **and** SafeToOpen **then** Dir:= open **endif**

This gives the desired semantics that the controller has to react as soon as its enabling condition becomes true. The first part of the controller handling the deadline is given implicitly by the labels of agent TRACK below. Then we use $\mathbf{if}\, time(enter(x)) > time(leave(x))\, \mathbf{then}\, time(enter(x)) + WaitTime\, \mathbf{else}\, \infty\, \mathbf{fi}$ as the value of Deadline(x).

For the environment condition 1 (track behaviour), we have to make the tracks an active part of the specification. This is also done quite naturally in ASM.

delayable agent TRACK
enter(Self): **if** TrackStatus(Self) = empty **then** TrackStatus(Self):= coming; urgency:= lazy **endif**
if TrackStatus(Self) = coming **and** $d_{min} \leq$ dist(now,ChangeTime(TrackStatus(Self))) $\leq d_{max}$ **then**
TrackStatus(Self):= inCrossing
endif
leave(Self): **if** TrackStatus(Self) = inCrossing **then** TrackStatus(Self):= empty **endif**

Note that the agent Track is considered delayable, but the first transition (enter) is lazy, i.e., it need not to appear in all runs. This way, we have implemented all timing requirements at the specification level. This specification does not include delaying actions, such that we did not need to use the **readyAt** construct. We could have used it in the GATE which may act immediatly to any change of direction but take some time to actually execute the necessary move.

In order to be in line with the original specification, we choose the *Time* and *Duration* domains, which have not yet been fixed, to be reals and use Property 1.1 (weak time order is total) and Property 1.2 (additivity of distances). The Properties 3.2 and 3.3 are used implicitly, due to the urgency attributes. The properties restricting the relation of system progress to time progress (non Zenoness, bounded variablity,

discreteness,...) may be chosen independently. Notice that the specification itself enforces finite variability. The reason is that the crossing action of the track imposes a minimal duration of a cycle of status changes of the track, and both the Controller and the Gate depend on these status changes to have enabled actions. Model-checking techniques may be applied to verify this property. Property 1.3 requiring strict time progress for causally related moves, however, is inappropriate for this example, because it conflicts the eager controller.

6. Summary and Conclusions

We have proposed a semantic framework based on timed event structures for discussing the incorporation of time in specification languages, in particular those with a partial order semantics, such as ASM. In contrast to other approaches for timed event structures we are aware of, we allow time to be a partial order on events, and we use constraints on the different relationships between events to represent the general constraints defining any specific timing framework.

Choosing event structures, we attach time points to state changes (moves, events), whereas time implicitly progresses in states. For any framework with global time, one can always come up with a model in which time is defined as a function from time to state that is equivalent to a given event-based model of our framework, in the sense that it satisfies the same properties. This can be done by using the corresponding reachability graph instead of the event structure, and by refining the time domain sufficiently, so as to include different time points for events that occur at the same time in the event structure, as it has been proposed for example in HASM [64]. For non-global time, this transformation requires at least the introduction of a notion of local state which permits to map local times to local states.

We have expressed properties of timed executions used in existing frameworks as constraints on timed event structures, in order to show the appropriateness of our semantic timing framework for this purpose. We have shown how to define the corresponding timing frameworks in the context of ASM, that is at the specification level, and how to use the formulated properties. To this aim, we have proposed some minimal syntactic extensions to ASM, which allow us to include the specification of time constraints in an ASM specification without making "time" a part of the state. In the context of ASM, this is usually done by defining timing constraints globally, separate from the description of the system under study, whereas our proposal allows including timing constraints into the specification of the concerned agents.

We have not yet fully addressed the aspects of constructivity. ASM have been introduced as a formalism for the constructive description of systems, allowing to construct the set of possible successor moves and states from any given state, or in terms of event structures, prefix of execution. Imposing additional constraints, in particular the properties we have defined and discussed in Section 3, may reduce the executability of the framework. Nevertheless, some constraints or modelling disciplines may help to avoid this problem, similar as in the context of timed automata.

Another important future work is about an appropriate notion of refinement of timed specifications. In the context of ASM, two natural notions of refinement have been introduced [17]. We have not yet studied in how far they are compatible with our framework for defining timed extensions.

We have not provided an exhaustive treatment of all possible and existing timing frameworks, but we believe that our semantic framework is expressive enough to characterise the dynamic semantics of other frameworks by the introduced properties. When structure related characteristics are considered, however, the semantic domain itself needs to be enriched with the required structural concepts.

Moreover, not only the expressiveness of the semantic framework counts, but also the practical usability of the specification language. Existing frameworks for handling time have put much effort into finding appropriate concepts and notations which can be made available in ASM through the definition of appropriate macros. We have just given a few of them in order to illustrate how this can be done.

We have defined so far a notion of *model* for timed specifications and not yet discussed an appropriate *logic* for the expression of properties of such timed specifications which needs still to be done. For this purpose, the properties given may be lifted to the level of some first order temporal logic (as used in [66]) so as to obtain an adequate logic.

Finally, the computation model defined here is global, in the sense that a timed event structure is associated globally with a system. In general, systems are obtained as a composition of subsystems, and these composition operators should also be reflected at the semantic level. An interesting topic is to study how a compositional semantics can be obtained depending on the choice of a set of properties. For example, in order to achieve a compositional setting in the presence of urgency or maximal progress, it is necessary to add an urgency concept at the semantic level [21], that is in our setting in a timed event structure.

Acknowledgements

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

We thank Egon Börger and Anatol Slissenko for the opportunity to be included in this journal as well as for their patience and support for preparing this paper. We are also very thankful to the anonymous reviewers giving us many valuable hints without which we would not have been able to produce this final version with the same degree of completeness and formalisation.

References

[1] Abrial, J.-R., Lee, M. K. O., Neilson, D., Scharbach, P. N., Sørensen, I. H.: The B-Method., *VDM Europe (2)*, LNCS 552, 1991.

[2] Alpern, B., Schneider, F.: Recognizing Safety and Liveness, *Distributed Computing*, **2**, 1987, 117–126.

[3] Altisen, K., Tripakis, S.: Implementation of Timed Automata: An Issue of Semantics or Modeling?, *Formal Modeling and Analysis of Timed Systems, Third Intl Conf. FORMATS 2005, Uppsala, Sweden*, LNCS 3829, 2005.

[4] Alur, R., Courcoubetis, C., Dill, D.: Model Checking for Real Time Systems, *5th annual IEEE Symposium on Logic in Computer Science, Philadelphia*, June 1990.

[5] Alur, R., Courcoubetis, C., Henzinger, T. A., Ho, P.-H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems, *Hybrid Systems 1992*, LNCS 736, 1993.

[6] Alur, R., Dill, D.: A Theory of Timed Automata, *Theoretical Computer Science*, **126**, 1994, 183–235.

[7] Alur, R., Henzinger, T. A.: A Really Temporal Logic, *30th Symposium on Foundations of Computer Science (FOCS 89)*, IEEE, 1989.

[8] Ashenden, P. J.: *The VHDL Cookbook. First Edition*, Dept. Computer Science University of Adelaide South Australia, 1990.

[9] Baier, C., Katoen, J.-P., Latella, D.: Metric Semantics for True Concurrent Real Time., *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg*, LNCS 1443, 1998.

[10] Baier, C., Majster-Cederbaum, M. E.: Metric Semantics from Partial Order Semantics., *Acta Inf.*, **34**(9), 1997, 701–735.

[11] Beauquier, D., Slissenko, A.: *On Semantics of Algorithms with Continuous Time*, Technical Report 97-15, Department of Informatics, Université Paris-12, 1997.

[12] Beauquier, D., Slissenko, A.: A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class, *Annals of Pure and Applied Logic*, **113**(1–3), 2002, 13–52.

[13] Bednarczyk, M.: *Categories of Asynchronous Systems*, Phd thesis in Computer Science, University of Sussex, 1988.

[14] Ben-Ari, M., Manna, Z., Pnueli, A.: The Temporal Logic of Branching Time, *Acta Informatica*, **20**, 1983.

[15] Benveniste, A., Le Guernic, P., Halbwachs, N.: A Decade of Concurrency, Reflexions and Perspeptives, *REX Symposium*, LNCS 803, 1993.

[16] Bergstra, J. A., Ponse, A., SmolkaS. A., Ed.: *Handbook of Process Algebra*, Elsevier, ISBN: 0-444-82830-3, 2001.

[17] Börger, E.: The ASM Refinement Method, *Formal Asp. of Comput.*, **15**(2-3), 2003, 237–257.

[18] Börger, E., Fruja, G., Gervasi, V., Stärk, R.: A High-Level Modular Definition of the Semantics of C#, *Theoretical Computer Science*, **336**(2-3), 2004, 235–284.

[19] Börger, E., Gurevich, Y., Rosenzweig, D.: The Bakery Algorithm: Yet Another Specification and Verification, in: *Specification and Validation Methods* (E. Börger, Ed.), Oxford University Press, 1995, 231–243.

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

[21] Bornot, S., Sifakis, J.: An Algebraic Framework for Urgency, *Information and Computation*, **163**, 2000.

[22] Bornot, S., Sifakis, J., Tripakis, S.: Modeling Urgency in Timed Systems, *International Symposium: Compositionality - The Significant Difference*, LNCS 1536, 1998.

[23] Caspi, P., Benveniste, A.: Towards an Approximation Theory for Computerised Control, *EMSOFT'02* (A. Sangiovanni-Vincentelli, J. Sifakis, Eds.), LNCS 2491, Grenoble, October 2002.

[24] Chandy, K. M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. Comput. Syst.*, **3**(1), 1985, 63–75.

[25] Chaochen, Z., Hoare, C. A. R., Ravn, A. P.: A Calculus of Durations., *Inf. Process. Lett.*, **40**(5), 1991, 269–276.

[26] Chellas, B. F.: *Modal Logic, an Introduction*, Cambridge University Press, 1980.

[27] Emerson, E. A., Halpern, J. Y.: 'Sometimes' and 'not never' revisited: On Branching versus Linear Time, *10th ACM Symposium on Principles of Programming Languages (POPL 83)*, 1983, Also published in Journal of ACM , 33:151-178.

[28] Emerson, E. A., Mok, A. K., Sistla, A. P., Srinivasan, J.: Quantitative Temporal Reasoning, *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*, LNCS 407, 1989.

[29] Esterel Technologies: SCADE Suite, `http://www.esterel-technologies.com/products/scade-suite/overview.html`.

[30] ETSI: ES 201 873 1, v2.2.1: "The Testing and Test Control Notation TTCN-3: Core Language ", 2002.

[31] Fidge, C. J.: Logical Time in Distributed Computing Systems., *IEEE Computer*, **24**(8), 1991, 28–33.

[32] Glässer, U., Gotzhein, R., Prinz, A.: The Formal Semantics of SDL-2000: Status and Perspectives, *Computer Networks*, **42**(3), 2003, 343–358.

[33] Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods, *Workshop on Computer-Aided Verification (CAV), Rutgers*, LNCS 531, 1990.

[34] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide, in: *Specification and Validation Methods* (E. Börger, Ed.), Oxford University Press, 1995, 9–36.

[35] Gurevich, Y., Huggins, J.: The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions, *Proceedings of CSL'95 (Computer Science Logic)*, LNCS 1092, 1996.

[36] Gurevich, Y., Schulte, W., Campbell, C., Grieskamp, W.: *AsmL: The Abstract State Machine Language*, Technical report, Version 2.0, Microsoft Research, Redmond, 2002.

[37] Harel, E., Lichtenstein, O., Pnueli, A.: Explicit Clock Temporal Logic, *5th Symposium on Logic in Computer Science (LICS 90)*, IEEE, 1990.

[38] Henzinger, T. A.: The Theory of Hybrid Automata., *Conference on Logic in Computer Science, LICS*, 1996.

[39] Henzinger, T. A., Horowitz, B., Kirsch, C. M.: Giotto: A Time-Triggered Language for Embedded Programming., *Embedded Software, 1st International Workshop, EMSOFT 2001, Tahoe*, LNCS 2211, 2001.

[40] Henzinger, T. A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks?, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna*, LNCS 623, 1992.

[41] Henzinger, T. A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems, *Inf. Comput.*, **111**(2), 1994, 193–244.

[42] Hirschfeld, Y., Rabinovich, A. M.: Logics for Real Time: Decidability and Complexity, *Fundam. Inform.*, **62**(1), 2004.

[43] Ilogix: Rhapsody Development Environment, `http://www.ilogix.com/rhapsody/rhapsody.cfm`.

[44] Ilogix: Statemate for Embedded Systems Design Software, `http://www.ilogix.com/`.

[45] ISO/IEC: E-LOTOS (Enhanced LOTOS), ISO/IEC standard, 2001.

[46] ITU-T: *Recommendation Z.100. Specification and Description Language (SDL)*, Technical Report Z-100, International Telecommunication Union – Standardisation Sector, November 2000.

[47] ITU-T: ITU-T Study Group 10: SDL Semantics, 2002, URL: http://rn.informatik.uni-kl.de/projects/sdl/.

[48] Katoen, J.-P., Langerak, R., Latella, D., Brinksma, E.: On Specifying Real-Time Systems in a Causality-Based Setting., *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96, Uppsala*, LNCS 1135, 1996.

[49] Kopetz, H., Grünsteidl, G.: TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems., *Symposium on Fault-Tolerant Computing, FTCS, Toulouse*, IEEE, 1993.

[50] Koymans, R.: Specifying Real-Time Properties with Metric Temporal Logic., *Real-Time Systems*, **2**(4), 1990, 255–299.

[51] Lamport, L.: The Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems*, **16**(3), May 1994.

[52] Lamport, L.: Real-Time Model Checking is Really Simple, *CHARME 2005, Saarbruecken*, LNCS, 2005.

[53] MathWorks: MATLAB and Simulink for Technical Computing, `http://www.mathworks.com/`.

[54] Nicollin, X., Sifakis, J.: An Overview and Synthesis on Timed Process Algebras., *REX Workshop*, LNCS 600, 1991.

[55] Nielsen, M., Plotkin, G. D., Winskel, G.: Petri Nets, Event Structures and Domains, Part I., *Theor. Comput. Sci.*, **13**, 1981, 85–108.

[56] Nunes, I., Fiadeiro, J. L., Turski, W. M.: Coordination Durative Actions., *COORDINATION*, LNCS 1282, 1997.

[57] Nunes, I., Fiadeiro, J. L., Turski, W. M.: A Modal Logic of Durative Actions, *Advances in Temporal Logic*, Kluwer Academic Publishers, 2000.

[58] Ouaknine, J., Worrell, J.: Revisiting Digitization, Robustness, and Decidability for Timed Automata., *18th IEEE Symposium on Logic in Computer Science (LICS 2003), Ottawa*, IEEE Computer Society, 2003.

[59] Peled, D.: All from One, One for All: On Model Checking Using Representatives., *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece*, LNCS 697, 1993.

[60] Pnueli, A.: The Temporal Logic of Programs, *18th Symposium on Foundations of Computer Science (FOCS 77)*, IEEE, 1977, Revised version published in TCS, 13:45–60, 1981.

[61] Prior, A. N.: *Past, Present, Future*, Oxford University Press, 1967.

[62] Puri, A.: Dynamical Properties of Timed Automata., *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby*, LNCS 1486, 1998.

[63] Rescher, N., Urquhart, A.: *Temporal Logic*, Springer Verlag, 1971.

[64] Rust, H.: Hybrid Abstract State Machines: Using the Hyperreals for Describing Continuous Changes in a Discrete Notation, *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland* (Y. Gurevich, P. Kutter, M. Odersky, L. Thiele, Eds.), nr 87 in TIK-Report, Swiss Federal Institute of Technology (ETH) Zurich, March 2000.

[65] Shields, M.: Concurrent Machines, *Theoretical Computer Science*, **28**, 1985, 449–465.

[66] Slissenko, A.: A Logic Framework for Verification of Timed Algorithms., *Fundam. Inform.*, **62**(1), 2004, 29–67.

[67] Stärk, R., Schmid, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.

[68] Trakhtenbrot, B. A.: Understanding Automata Theory in the Continuous Time Setting, *Fundam. Inform.*, **62**(1), 2004.

[69] Tripakis, S.: Verifying Progress in Timed Systems., *ARTS*, LNCS 1601, 1999.

[70] Tripakis, S., Yovine, S., Bouajjani, A.: Checking Timed Büchi Automata Emptiness Efficiently., *Formal Methods in System Design*, **26**(3), 2005, 267–292.

[71] Wang, J.: *Timed Petri Nets: Theory and Application*, Kluwer, 1998.

[72] Winskel, G.: *Events in Computation*, Phd in Computer Science, University of Edinburgh, 1980.