

Timed annotations with UML ^{*}

Susanne Graf, Ileana Ober and Iulian Ober

VERIMAG** – Centre Equation – 2, avenue de Vignate – F-38610 Gières – France
{Susanne.Graf, Ileana.Ober, Iulian.Ober}@imag.fr
<http://www-verimag.imag.fr/{graf,iober,ober}>

Abstract. In this paper we describe an approach for real-time modeling in UML focusing on analysis and verification of time and scheduling related properties. We show that the use of timed events, representing instant of state changes, provides the right level of abstraction for reasoning about timed computations. This is also, at notation level, the choice of the OMG UML Real-Time Profile. We complete this profile by identifying important events and duration expressions. One originality of the approach presented here, is that it provides a formal semantics of the time related primitives in terms of timed automata with urgency. An interesting point is that this time extension is independent of the dynamic semantics of the functional part.

1 Introduction

Modeling plays a central role in software and systems engineering. The use of models can replace experimentation on actual systems with important advantages such as:

- Enhanced modifiability of the model and of its parameters,
- Ease of construction by integration of models of heterogeneous components,
- Generality by using generalization mechanisms and behavioural non determinism,
- Enhanced observability and controllability, in particular, avoidance of probe effect and of disturbances due to experimentation,
- Possibility of abstraction and application of formal methods.

Building models which faithfully represent complex systems in all their aspects is a non-trivial problem. Mostly, modeling techniques are applied at early phases of system development at a high level of abstraction. Nevertheless, there is a need for a unified view of the various activities in the whole life cycle and of their inter dependencies. In particular, non functional aspects, such as timing, must be integrated into the system model, and this must be done already at an early stage of development.

UML has become a standard by now in the domain of software development and is imposing itself also in the domain of real-time and embedded systems. UML aims at providing an integrated modeling framework encompassing architecture descriptions, and behaviour descriptions, as supported by various behavioral specification languages

^{*} This work has been partially supported by the IST-2002-33522 OMEGA project

^{**} VERIMAG is a research laboratory associated with CNRS, Université Joseph Fourier and Institut Nationale Polytechnique de Grenoble

such as languages based on the concept of communicating state machines. To cover real-time aspects, a proposal for "Schedulability, performance and time" has been submitted to OMG in August 2000 and accepted as standard in March 2002. Nevertheless, this profile provides syntax, and cannot be considered as a complete framework for UML based development of real-time applications. Also, relatively little work on applying timed analysis and verification methods to UML models has been carried out so far.

1.1 The OMEGA project

The work reported here is ongoing in the OMEGA project [30]. The purpose of Omega is to provide a framework for a model based development approach in the domain of real-time and embedded systems. The main problem to be solved is how to build systems with a guaranteed level of quality in a cost effective manner. There is a general agreement that a means to achieve this the existence of a global model, integrating all aspects of the system, as well as the assumptions made on the environment, during the entire life cycle of the system allows the maintenance of a coherent system. In particular, information on non functional aspects, such as timing, should be available at an early development stage.

Such a model-based approach is only useful if it is accompanied by tool support for analysis at all stages of development in order to detect inconsistencies and design errors as early as possible. Extraction of particular analysis models must be tool supported, and in the case of model updates, guidance must be provided, indicating which analysis has to be redone and which ones need not. Finally, the existence of automatic generation of code depending on the target platform is the only means to avoid that bugs are eliminated at code level only, leading to the divergence between model and code and making the model useless.

A means to detect design errors early, especially in the context of real-time systems, where the reactivity of the system is as important as the functionality, is to take into account non functional aspects, and in particular time-related aspects, as early as possible. Indeed, early decisions on the order of activities may later need important redesign when it turns out that time constraints are not met. Not resolving non-determinism before timing constraints are taken into account, may avoid this problem. Nevertheless, models with more non-deterministic interactions are harder to verify.

The long-term vision of the Omega project is to provide an environment for an UML based approach for real-time embedded systems, providing the right notations, a formal semantics and validation tools supporting this semantics and methodology.

In this paper, we discuss the framework for real time defined in the Omega project.

1.2 A framework for Time in UML

In the context of UML, where a system is described as a set of communicating objects, the order in which causally unrelated actions in concurrent objects are executed is a priori non constraint. This is fine for an abstract model with many possible implementations; but on one hand, a high degree of non determinism makes validation extremely hard, and on the other hand some of the possible ordering of system activities may

lead to violation of desired system properties. There are two approaches for getting a restricted set of possible executions:

- use of particular interaction modes, as the one used in the synchronous approach, or run-to-completion execution of groups of objects or, more generally, the use of a priority order between actions (defined by their triggers).
- take into account known (or assumed, required,...) real-time features of the environment and the system at run-time, in order to determine possible ordering.

Very often, in practice, the first approach is used, and only at a later stage assumptions or knowledge on the environment or the execution platform are used to verify if the chosen execution order is a possible one. Taking into account timing constraints early may avoid the risk of necessary redesign by still providing a model simple enough to be verified.

A notation for the expression of the time related aspects of a system must be able to express:

- Time related requirements, mainly constraints on end-to-end delays of the system or its components
- Time related assumptions on the external environment of the system, mainly response times and inter occurrence times of requests. In UML, the external environment being modeled explicitly by means of actors, assumptions on the external environment can be modeled in the same way as constraints on the system.
- Time related assumptions on the underlying execution platform, such as execution times of tasks and actions as well as the dependencies between concurrent tasks via shared resources. Particular notations are needed for that, as we do not want to ask the user to build an explicit platform model.
- Time dependent system behaviour, which is often modeled best by means of timers or explicit access to a system clock.

We define a notation based on the UML profile for Performance, Scheduling and Time, including:

- A constraint-based formalism allowing to restrict the duration between occurrences of events (where our notion of event allows to capture any event or state transition at the semantic level) by means of convenient patterns (such as `ResponseTime` associated with calls, `InteroccurrenceTime` associated with events, `ExecutionTime` associated with actions or “sequences of actions”, ...), as well as an expressive set of notations for the identification of durations between any occurrences of any events.
- a number of operational concepts, as they exist in most modelling languages for real-times systems: a notion of system time, which can be explicitly accessed in the action language by a construct `now`, stored in variables and used in guards, as well as a timer concept.

Notice that, from the semantic point of view, the introduction of a global clock, time stamps and time dependent guards in the action language is sufficient to provide all the expressive power, but it is not sufficient from the modelling point of view. It leads to an important modelling overhead and implies that assumptions on execution time must be

expressed in the form of explicit waiting, thus hindering appropriate code generation. Moreover, it is easy to generate operational timing annotations from patterns (expressing the intuition), whereas the other direction is not possible. Notice that many of the existing modelling formalisms for real-time (such as timed automata like extensions of state charts) are semantic level formalisms.

The semantics of our timing extensions defines a mapping from derived notations into primitive ones, whereas the semantics of primitive notations is defined in terms of extended timed automata which synchronize on occurrences of events and restrict the set of possible sequences of timed events representing the semantics of the model.

An interesting point is that this timed semantics is orthogonal to the semantics of the functional model, and can therefore be combined with *any* functional semantics for UML which is expressive enough to identify all events referred to in the time constraints.

The paper is organized as follows. Section 2 gives an overview on related effort on the introduction of time in UML and related modelling formalisms. Section 3 presents our real-time profile of UML, and Section 4 sketches its semantics. Section 5, provides a short discussion on the implementation of the time extensions in the IF verification tool.

2 Related work on UML and time

UML offers a variety of notations for capturing many aspects of software development, mostly focused on functional aspects, but also covering requirement analysis, implementation, verification, and testing. In this paper we address an aspect neglected in the initial definitions of UML: the specification of the model behavior with respect to time progress. In this section, we discuss related work dealing with explicit handling of time and of time related information.

The UML Real-Time profile [28] (called in the sequel UML RT) is a first step in answering OMG's request for a "*UML-based paradigm for modeling time-, schedulability-, and performance-related aspects of real-time system that would be used to (1) enable the construction of models appropriate for quantitative analysis regarding these characteristics and (2) to enable inter-operability between different analysis and design tools.*" UML RT includes features for describing a variety of aspects of models of real-time systems, such as timing, resources, performance, schedulability, etc. For what concerns timing, the profile provides only operational features like *timer* and *clock* objects, and a set of data types (*time*, *duration*). With respect to this, our approach introduces new features that allow a more abstract and descriptive specification of timing, allowing event identification and definition of general timing constraints on events. Moreover, the profile does not enter into deep semantic considerations.

The UML 2.0 [14] proposal pays more attention to time related aspects than the current UML standard [34]. Indeed, operational time-related concepts that are common to our approach and to the UML RT profile (*timers*, time related types), are present in UML 2.0. But there is no syntax for the expression of time constraints.

In [7], Douglass underlines the importance of time-related information in real-time systems. He distinguishes between six kinds of time (absolute, mission, friendly, simulation, interval and duration), but most of these distinctions do not really help to solve the analysis problem. The main intended application is schedulability analysis, supported by a tool based on rate monotonic analysis (RMA) [16] of a set of periodic tasks in order to estimate (statistically) execution times.

Lavazza et al. [24] present an approach to real-time development centered around a concrete case study. The approach consists in translating UML models in which time-related properties are specified (e.g. guarded timeouts, transitions dependent on other transition times, etc.) into first-order temporal logic with time, on which a standard model checking tool [35] is applied. Knapp et al. [23] use timed state machines for describing a model, and collaboration diagrams with time constraints to describe properties. In order to check the consistency of the time constraints of the model and the property, timed state machines as well as timed collaboration diagrams are compiled into timed automata and checked using the model checking tool UPPAAL [22].

[9] presents work on a UML profile for real-time constraints specifications based on OCL 2.0 [27]. The work consists in extending the OCL 2.0 meta-model with concepts needed to express state chart configurations, execution paths, past and future execution paths, etc. The semantics of the temporal expressions is given in terms of a mapping to “Clocked” version of temporal logic [31]. Nevertheless, the paper only shows some very simple properties, and it is not described how/if they can be verified.

Amongst the commercial UML tools, we mention Rhapsody [19], which is based on a synchronous approach. Globally, the system is seen as a sequential system advancing in well defined global steps (cycles) where time is measured in number of cycles. Within a cycle, concurrency boils down to priority based preemptive scheduling, where timing analysis is based on RMA. ARTISAN Real-Time Studio [2] extends UML to model the system’s reaction to events, time constraints, concurrent tasks and partitioning applications across multiple processors. ROOM[32] and Tau Generation-2 [33] are based on communication via asynchronous events. A global notion of time is defined which can be used to define time dependent behaviours using timers, time stamps and time guards. Both Rhapsody and Artisan-RtS use UML as a graphical programming language, whereas ROOM and Tau have more features of modeling tools.

Although not UML-based, we mention here the related efforts done in the context of SDL [20] because of the similarities of the aims and the formalisms involved. Like ROOM, SDL has basic concepts for time, time-related data types (time and duration), and timers which can be used for the definition of time dependent behaviours. [4] describes the concepts necessary to better address real-time needs, consisting mainly in the ability to define time and scheduling *constraints*. [11] provides a framework for specifying real-time constraints leading to verifiable systems. The QSDL [6] defines an extension of SDL with probabilistic execution time constraints attached with tasks and a notation for some minimal deployment information. The so defined extended models are then fed into a simulator for performance evaluation. An adaptation of the here presented time model to SDL, is presented in [12].

Message sequence charts (MSC) defined by ITU [21] are a formalism related with UML sequence diagrams. The two main directions of the work on timed MSCs are those

based on *event distance* - the ITU efforts on adding time to MSC as part of the standard and those using *time guards* - the efforts of Harel and al. [17] apply temporal logic on LSC [5], an extension of sequence diagrams with mandatory/optional behaviour. The latter embeds a subset of LSC into temporal logic. The result is an operational framework for timed specifications.

Metropolis [29] comprises a framework for pure scheduling and allows to express deadlines, execution times, scheduling policies, etc.

At the semantic level, different variants of timed automata [1] are used with success to model time related features in verification tools for an expressive set of real time properties. KRONOS [35], UPPAAL [22] or Hytech [18] are the most significant tools of this family.

Many other approaches dealing with *temporal* aspects on UML, deal in fact with (partial) *orders* of events. This is the case for instance of the OCL extensions for specifying timing constraints [8]. Here, OCL is extended with *temporal* operators for reasoning on possible ordering of events.

3 Framework for the definition of timed models

This section gives a brief overview of the constructs and notations introduced in our profile. The main ingredients are: time related types *time* and *duration* and a notion of *timer* as it exists in real-time programming languages. We refine the notion of *timed event* as it is defined in UML RT and provide a syntax for defining events associated with any state change in the semantic model, we define also an expressive set of duration expressions and introduce concepts for scheduling and schedulability analysis.

3.1 Time related types, time monitoring mechanisms and time progress model

The proposed profile uses a small set of time related primitives, which are defined with the concern of being compatible with UML RT. The time model is based on two data types: *Time* - corresponds to time instances, and *Duration* - corresponds to the time elapsed between two instances of time. A particular expression of type *time* is *now*, which always holds the *current time*. *Now* can be used in action specifications, guards, as actual parameters of signals and operations, in constraints, etc.

Nevertheless, the model cannot explicitly alter the value of *now* which is provided by some external mechanism, representing “physical time” and satisfying the constraint that values of *now* increase. Moreover, in any infinite computation the value of *now* must grow over all bounds (absence of Zeno behaviours). Finally, we suppose that time progress is fair with respect to system progress, that means that whenever, a system transition is enabled “forever” (without time-limit), it will be taken at some point of time. Notice, that an important feature of our model is that enabled transitions are *not* necessarily taken at the *earliest* point of time at which they are enabled, as this is the case in many existing modeling frameworks based on a “synchrony hypothesis”. Our choice leads on one hand to more realistic time models, but it makes mandatory the addition of hypotheses on maximal time progress in system states.

As in UML RT and in many formalisms for real-time software development, we define two related timing mechanisms with the same set of actions and attributes: *timer* and *clock*. *Timer* is a mechanism that generates a *timeout* when a specified duration time elapsed. There are two versions of a timer, one where *timeout* corresponds to sending an asynchronous signal, which can immediately or later be used for triggering an action, and one, in which *timeout* corresponds to a *call*, which can be “canceled” when a *reset* action is issued after the timeout time has been reached, but before the timeout has been consumed.

This time model has a unique global reference time which is external. *Local time* can be handled by means of local timers and clocks, for which a maximal *drift* and/or *offset* with respect to global time can be defined, as in UML RT.

3.2 Timed events

The aim of our approach is to express constraints on the duration between occurrences of *events*. Thus, the notion of *event*, defining a point of time - the point of time at which the event *occurs*, plays a crucial role in our approach. We have identified a number of “event kinds” allowing to identify all changes of the system state. For instance, in a signal exchange, three events can be identified:

- the *send* event, defining the moment at which the signal is sent by the sender,
- the *receivesignal* event, defining the moment at which it is received in the input queue of its target,
- and the *acceptsignal* event, defining the moment at which the signal is processed (this corresponds to the implicit discarding of the signal or to the instant at which a transition is triggered by the signal)

Notice that in some cases, some of the events may coincide. E.g. in the case of a run-to-completion semantics, the events *receivesignal* and *acceptsignal* of a signal exchange between two objects of the same activity group, may coincide, and when there is no transmission delay between the sender and the receiver, then the events *send* and *receivesignal* may coincide.

List of event kinds We have identified for each syntactic construct some associated event kinds. These event kinds should allow to capture all meaningful state changes in the underlying semantic model (which is a transition system defining a set of state/event sequences):

1. With a *signal transmission*, are associated three events: *send*, *receivesignal*, and *acceptsignal* explained above.
2. With an *operation call* are associated six events:
 - invoke* - the emission of the call request by the caller,
 - receive* - the reception of the call by the callee,
 - accept* - the start of the actual processing of the call by the callee,
 - invokereturn* - the emission of the “return” response by the callee,
 - receivereturn* - the reception of the “return” by the caller, and
 - acceptreturn* - the consumption of the return.

3. With an *action specification* are associated two events:
start - the moment at which the action execution starts,
end - the moment at which the action execution ends, *startend* - the moment at which the action is executed - shorthand defined in the case of instantaneous actions.
4. with a *timer* are associated two events:
occur - the moment at which the expiration time is reached, and
timeout - the moment at which the timeout triggers a transition
 Other important moments are the events corresponding to the execution of the instantaneous actions *set* and *reset*.
5. With a *state machine transition* are associated two events:
starttransition - the moment at which the transition execution starts, and
endtransition - the moment at which the transition execution ends. Again, in the case of instantaneous transitions, we define a shorthand *startendtransition* - the moment at which the transition is executed.
6. With a *state machine state* are associated two events:
enter - the moment at which the state is entered, and
exit - the moment at which the state is left.
7. With an *object* are associated two events:
create - the moment at which the object is created, and
delete - the moment at which the object is deleted from the system.
8. With any *Boolean expression* may be associated a *change event* - corresponding to the moment at which its value changes.

Notice that even a fully specified event can have several occurrences throughout the lifetime of a system. Just as any attribute has a “current value” at any time, an event has a current *occurrence*, which we choose to be the *most recent* occurrence of this event. That means that at any point of time the “current” *occurrence time* of an event is the occurrence time of the most recent occurrence of this event, whose value is undefined before its first occurrence.

In order to reason about histories of occurrences of events, we define event expressions of the form *pre(event)* - naming the second last occurrence of *event* - *pre(pre(event))*, etc. This allows to reason on the occurrence times of any finite history of occurrences of events, which is sufficient in most cases.

Event definition and identification The event types listed above are used in time annotations. Notice that not all events can be identified syntactically in the UML specification. For example, amongst the events associated with a signal exchange, only the *send* event and the *acceptsignal* event, under the condition that it corresponds effectively to a transition trigger (and not to an implicit discard), can be syntactically identified, whereas the *receivesignal* event corresponding to the instant at which the signal reaches the receivers input queue can not be syntactically captured. Therefore, we need a syntactic mechanism to specify events. In our setting, events are defined as attributes of some *event type* which is a UML class stereotyped with *«TimedEvent»* Events are either *local* to a class, or *global* to the model. *Local* events are defined as attributes of some class present in the model. This class defines the context for the event instance. *Global* events

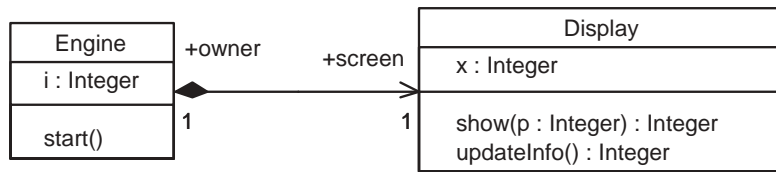


Fig. 1. UML class diagram for the example

are defined as attributes of a special class, stereotyped $\ll TimeAnnotations \gg$. This class is never instantiated and is not part of the functional specification of the model. We only add it to gather global model information needed for the definition of time constraints.

An *event type* defines a pattern for recognizing event occurrences. An *event type* definition may include local *attributes* storing information on the system state at event occurrence time. It contains a *matching clause* describing the *kind of event* (as provided by the list in the previous section) and some *event parameters* - specific to its kind (e.g. for a *send* event the signal that is sent, the target, etc.). A *filter condition* over the event parameters allows to refine the specification of the event type, while an *action statement* specifies a simple action to be executed when the matching clause and its filter condition are satisfied. The most common example of action statement in an event definition is the assignment of the event type attributes with some values extracted from the global system state.

In any execution, each event defines a possibly empty sequence of actual *event occurrences*. Each event occurrence satisfies the pattern and conditions imposed by the event type definition. As the system evolves, the values of the attributes - including the occurrence time - of an event instance may vary with every occurrence of the event.

Event syntax We give an example of an UML model containing an event specification, using the syntax of our setting. For the complete syntax for all event kinds, the interested reader is referred to [13]. Our goal is here just to illustrate how event types and event declarations are integrated within a UML model.

Consider a very simple UML model for an engine showing some information on some display device. After performing the modeling we obtained the UML class diagram described in Figure 1. The blueprints for this model contain the following time constraint:

Between the moment the engine starts to show data on its display and the moment the same engine updates the information on the display less than 10 time units pass, if the data on the engine (attribute i) increases exactly with 7 units.

This constraint cannot be captured by the regular UML model, instead it will be expressed as a time annotation. For this, we have to define the event types corresponding to the moments described in the time constraint. Figure 2 contains the definition of the event types obtained from the time constraint. *EvTIR* captures the moment the Engine starts to show information on its display. The match statement specifies the kind of event we have (invoke) and its parameters (caller, callee, operation), ensures that we are talking about the right display and stores the local information of the engine. The event

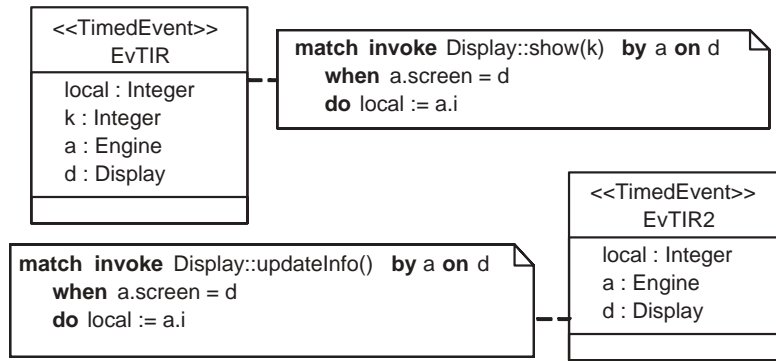


Fig. 2. Event type definitions

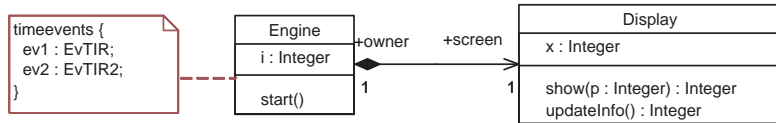


Fig. 3. Updated UML class diagram for the example

type *EvTIR2* is defined similarly. As the evaluations described in the time constraint are done in the context of the engine, we add event instances, corresponding to the event types we just defined, in the context of the engine. This leads us to the updated class diagram described by Figure 3.

In the next section we describe how the durations are evaluated, which allow us to finalize the modeling of this example.

3.3 Durations

The main objective of time annotations are the definition of constraints on *durations* between occurrences of events. In this section, we describe the concepts for the definition of durations offered by our framework.

One of the main challenges in this context is to find a suitable mechanism for identifying event occurrences corresponding to a “matching event occurrence pair”, defining a duration (or time distance). Intuitively speaking, in almost all cases, the matching pairs consist in “causally related” events. Nevertheless, this causal relationship can not always be identified syntactically.

A possible mechanism (proposed in some frameworks) consists in indexing all occurrences of events and to define the durations between occurrences of events with indexes satisfying some relationship. E.g., the “duration between the i^{th} invoke and the i^{th} acceptreturn of some operation invoked by a particular object” (defining a response time for the operation under consideration). While this kind of duration definition may be useful in some situations, it is clearly not appropriate in many cases, where no relationship exists between the causal relationship of events occurrences and their index.

We propose two different mechanisms for the identification of matching event occurrence pair defining a duration.

- (1) The first one consists in a duration expression of the form **duration**(*event1*, *event2*), in which always the *most recent* occurrences of the two involved event instances are matched. The powerful filtering mechanism possible in event definitions makes this mechanism quite expressive. Moreover, we increase the expressivity by allowing an additional *filter condition* on the event pair (similar to the event type filter conditions attached to the matching clauses described in the previous section) in terms of the event attributes (representing the history variables necessary for timed verification).

Some interesting types of matching event occurrence pairs can not be identified by the mechanism defined above. A typical example are causally related events in different objects which communicate through fifo buffers. Still, in many cases, the functional model itself includes an appropriate identification mechanism. For example, in a sliding window protocol, an explicit “sequence number” allows to identify causally related “send” and “acceptsignal” events. In this case, the mechanism above can be used by requiring a matching pair of event occurrences to carry the same sequence number.

Nevertheless, in cases where the functional model relies on the fifo communication mechanism, such an explicit identification of matching events might not be possible. In order to avoid the “pollution” of the software model with information only needed for timed validation, we need a more powerful concept for defining durations.

- (2) In the present proposal, we just propose two more temporal patterns which we found useful in practice:
 - in order to deal with events matching in a pipelined fashion as in the example above, we introduce an explicit notation for a *pipelined duration* (**p-duration**).
 - another useful pattern is the duration between the **first** of a sequence of occurrences of an *event1* and the **first consecutive** occurrence of an *event2*, for which we introduce a notation **f-duration**. A typical example is when event *event1* corresponds to a request - which might be repeated a number of times due to message loss - and *event2* corresponds to the occurrence of the desired effect, which should happen within a limited time starting from the first time a request has issued.

Example.

In the previous section example, we started the modelling of a time constraint that comes on the top of a non-time aware UML model. We added all the ingredients needed in order to capture the time constraint: event types (Figure 2), the event instance (present in the updated class diagram from Figure 3). The next thing to do is to define the time constraint corresponding to the specification given in plain text. The constraint is a simple duration expression on the event instances added, filtered by the specified condition. In our framework, the concrete syntax for this example is:

```
timeconstraints{ C1: duration(ev1, ev2)<=10
                    when ev2.local = 7+ ev1.local;
}
```

3.4 Predefined durations

The duration concepts defined in the previous section based on the explicit identification of events are powerful enough for the expression of most time constraints and allow a strict separation of functional design and time related specification. Nevertheless the explicit definition of all the events occurring in some constraint as well as the corresponding event type, can quickly become cumbersome. Moreover, typical time constraints correspond to relatively small set of patterns.

In order to simplify as much as possible the task of defining time constraints, we add explicitly notations for a number of duration patterns frequently used in practice. These patterns can be expressed in terms of basic events and durations defined in the previous section. Presently, we have identified only a small number of such patterns, listed hereafter, but more of them will be added depending on the needs identified by users:

- A *ClientResponseTime* associated with some *operation* in a given context (which might be restricted to subset of actual operation calls by a *matching condition*) corresponds the overall duration of a call, that is to the time elapsed between the occurrence of the *invoke* event associated with an operation call satisfying the matching condition and the corresponding *acceptreturn* event.
- A *ServerResponseTime* associated with an *operation* (and possible restrictions as above) corresponds to the time needed by the callee to handle the call, that is time elapsed between the occurrence of the *accept* event and the corresponding *invokereturn* event, that is the time.
- The *ExecutionDelay* associated with an *action* or a *transition* corresponds to the time between corresponding *start* and *end* events,
- Whereas the *ExecutionTime* associated with an *action* or a *transition* is useful in the context of scheduling analysis and defines only the accumulated duration during which the corresponding object (or activity group) is not suspended (due to execution of tasks of objects with higher priority)
- The *Period* associated with any *event* corresponds to the inter-occurrence time between any two consecutive occurrences of this event. In case that the event corresponds to a *receive* event, this duration corresponds to an inter-arrival time.
- A *Reactivity* can be associated with an *operation call* or a *signal* (possibly restricted to a class or object), and it defines the time elapsed between a *receive* event and the corresponding *accept* event.
- A *TransmissionDelay* is associated with a *communication path*, and defines the transmission delay of signals and/or calls, that is the time elapsed between a *send* and a corresponding *receive* event.

An interesting point of these patterns is that the constrained events need not to be explicitly defined by the user¹, but can be implicitly defined by the verification tool.

For instance, when the user defines a constraint of the form

$$ExecutionDelay(\text{some-action}) \leq 5$$

¹ the only exception is period, which has an event as argument

the system behaves as if there would be two event type definitions: one for the start and one for the end of the action, each one with the filter condition identifying “some-action”, two event instance definitions (one for each event type), and a duration expression between the two event instances.

3.5 Time constraints

As already mentioned, the definition of time constraints is the main objective of our framework. Time constraints are boolean combinations of basic time constraints which are comparisons between 2 expressions of type duration. These time constraints can be used in the model as predicates in transition guards or in decisions. And they can also be used to express invariance properties of the system.

Predicates defining *invariants* of the system can play two different roles:

- they may represent properties characterizing the model (that is they are assumptions on the environment or the underlying execution platform),
- or properties which can be derived from the model and which should be verified.

We distinguish these two kinds of constraints by means of explicit keywords.

3.6 Scheduling issues

In order to enable scheduling and schedulability analysis, some additional concepts are needed, but we haven’t worked out yet a concrete syntax for them.

As specified in in Section 3.4, we distinguish already between *execution delay* and *execution time* of actions or transitions, where the first one refers to the time elapsing between the start time end the end time of an action and the second one refers to pure execution time, which is obtained by integrating over all the durations in which the object was in the “executing” state.

In order this distinction to make sense we need to distinguish distributed and resource sharing parallelism between active objects, as this determines which execution times add up and which ones don’t. For this purpose, we allow the identification of *resources* (which might be dynamically created during the lifetime of the system, and where we distinguish between preemptible and non preemptible ones). The definition of the mapping from actions to resources, defining the set of resources needed for each action, is defined by means of particular actions which allow to dynamically add and eliminate resources from the set of needed resources.

Schedulers and scheduling policies are defined by dynamic priority orders between trigger events or objects. An important point for mastering the complexity is that priority orders are defined hierarchically, following the architectural structure of the specification.

4 Semantics of the timed annotations

In this Section we provide a sketch of the principle of the semantics of the time related concepts introduced in Section 3. This semantics is defined in terms of a dynamically

evolving set of timed automata synchronizing on the occurrence of events, which define a set of possible occurrence times of all events defined by the functional model and which constrain the occurrence times of events according to the set of duration constraints defined in the model.

4.1 Timed automata with urgency

We use the model of timed automata with urgency [3]. For each timed automata is defined a set of *clocks*, which are in fact time counters. In states, the values of time counters increase at the same rate with some externally defined abstract notion of time and transitions are taken at time instances as allowed by their *guard*, defined by a predicate on the values of clocks. In transitions, some clocks may be reset to zero. Each transition has a *guard* (which may be *true*) and an *urgency* which may take one of the values *lazy*, *delayable* or *eager*. The guard defines the set of time points at which the transition *can be taken*, whereas its urgency defines when the transition *must be taken*:

- a *lazy* transition may be taken within its enabledness interval, but time may progress and disable the transition
- an *eager* transition must be taken at the earliest point of time at which it is enabled; that means eager transition ensure maximal progress
- a delayable transition which is enabled at some point of time may not be disabled just by the fact that time progresses

Finally, an execution of a timed automata is a sequence of transitions (where in our case, transitions represent *events*) where for each transition an occurrence time is defined in terms of a valuation of its clock. If there exists a clock which is never reset, it can be used to define an occurrence time satisfying the criteria of monotonicity defined earlier.

Thus, timed automata are used to define the occurrence time of *events*, whereas the dynamic semantics of the untimed model defines only the possible ordering of events. The only requirement we impose on the untimed semantics is that it allows to identify all the events on which constraints are defined.

We have considered two different formal UML semantics: one based on labeled transition systems [30] and one [25] defined in terms of abstract state machines [15]. In both of them, all the events defined in the previous section can be identified, although they were developed with no timing capabilities in mind. Thus it is possible to define time extensions for both of them, using the approach described here. Although the two semantics differ ², both of them can be timed so that we can reason about e.g. the duration of an operation call or the execution time of an action.

4.2 Semantics in terms of timed automata

In order to define the the semantics of a time extended model, we need to define how *events*, *timers*, *time constraints* and resource mappings are expressed in terms of timed automata.

² For example, the concurrency model is different: while [30] considers that an active class has a single execution thread, in [25] an active class may do several things at a time, a new thread being created for each accepted operation call.

The timed automaton associated with each timer is relatively straightforward.

Each event is represented by a timed automaton synchronizing with all occurrences of the corresponding event in the functional model and updating the event memory. The amount of history that need to be stored depends on the kind of durations for which constraints are defined.

The semantics of a duration constraint is defined either by a single timed automaton or by a dynamically evolving set of timed automata, depending on the type of duration which is constraint:

- in the case of an expression $duration(e1,e2)$ or $f-duration(e1,e2)$ without additional filtering condition on the event pair - where at any time there exists at most one $e1$ event which occurred in the past and might match some future $e2$ event - a unique timed automaton is needed for each constraint.
- in the case of a constraint on $p-duration(e1,e2)$ or durations with matching conditions on pairs, at any point of time there may be several $e1$ events, which have occurred in the past and for which a matching $e2$ event may occur in the future. Here, a timed automaton needs to be created for *each occurrence* of an $e1$ event and which can be deleted at the occurrence of the corresponding $e2$ event.

The exact nature of the timed automata with constraints on the different types of durations can be found in a technical document [13]. An interesting point is the distinction of constraints expressing properties and constraints expressing assumptions. In timed automata expressing properties all the transitions are lazy as the occurrence time of the constrained events should be guaranteed by the rest of the model. On the other hand, timed automata expressing assumptions must impose time constraints on the model; for this they use delayable transitions forcing the constrained event to occur in a specified interval.

4.3 Some remarks concerning implementation

The semantics as sketched above is constructive, that means it can be directly implemented and used for simulation purposes. Obviously, such a naive implementation will be very inefficient and produce an infinite model, even if the underlying functional model is finite state. In the tool described in [26], we define a more efficient implementation, exploiting knowledge on the structure of the system. In this tool, the functional semantics of the model is defined in the IF format [10] as a dynamically evolving set of communicating state machines. For example,

- whenever a duration constraint is local to some state machine - meaning that both events correspond to transitions in the same state machine - they are sequential, and a more efficient implementation can be achieved, by just extending the state machine implementing the functional behaviour with a clock measuring the duration which is set at the occurrence of the first event and which is reset at the corresponding occurrence of the second event.

- static analysis can be used to detect “dead” duration constraints, corresponding to constraints where the first event has occurred, but the corresponding second event will never occur. In this case the corresponding clock (the local case) can be eliminated.
- the fact that all constraints express constraints on duration only, there is no need to consider the global time (represented in the model by *now*) in the model which eliminates one source of infiniteness of the model.

5 Conclusions

In this paper we describe an approach for adding time information to UML models, based on the definition of a framework for timed annotations. This framework is compatible with the *UML Real-Time profile for Performance Scheduling and Real-Time* in the sense that it is based on some of the concepts defined by this profile.

The main added value of our work is that we clearly identify the set of time concepts to be used together, and we give a formal semantics of a UML model containing such time concepts. Our set of primitives is intended to be used in the context of time and scheduling analysis.

Our framework is based on the intensive use of events. We identify the set of events that are needed in time annotations and that can be identified in most operational untimed semantics of UML (we have considered [30] and [25]). Through a concrete syntax, these events can be used in time annotations, on the form of constraints on duration expressions.

We define the semantics of our framework in terms of timed automata with urgency that use the identified events. The timed semantics of a UML model is built on the top of the untimed semantics. We consider important while defining a semantics for timed UML models not only to treat the timed part, but also to integrate it in properly with the rest of UML concepts: inheritance, association, data structure etc. Nevertheless, the current practice is rather oriented towards focusing on small parts of UML, and less on their integration with the whole UML definition.

Using the toolset based on this approach [26], we have successfully applied our profile on small UML models, and we intend to apply it on more realistic case studies in the framework of the OMEGA IST project.

References

1. R. Alur and D.L. Dill. A theory of timed automata. In *TCS94*, 1994.
2. *Artisan Real Time Studio*, 2001.
3. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *International Symposium: Compositionality - The Significant Difference*, LNCS 1536, 1998.
4. Marius Bozga, Susanne Graf Alain Kerbrat, Laurent Mounier, Iulian Ober, and Daniel Vincent. Timed extensions for sdl. In *SDL Forum 2001*. LNCS, June 2001.
5. W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*. Kluwer Academic Publishers, 1999. Journal Version to appear in *Journal on Formal Methods in System Design*, July 2001.

6. M. Diefenbruch, E. Heck, J. Hintelmann, and B. Müller-Clostermann. Performance evaluation of sdl systems adjunct by queuing models. In *Proc. of SDL-Forum*, 1995.
7. Bruce Powel Douglass. *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Object Technology Series. Addison-Wesley, 1999.
8. Stephan Flake and Wolfgang Mueller. Specification of real-time properties for uml models. In *Proceedings of 35th HICSS*. IEEE, 2002.
9. Stephan Flake and Wolfgang Mueller. A UML Profile for Real-Time Constraints with the OCL. In S. Cook J. M. Jézéquel, H. Hussmann, editor, *UML'2002, Dresden, Germany*, number 2460 in LNCS. Springer Verlag, 2002.
10. M. Bozga S. Graf and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen*, LNCS. Springer Verlag, June 2002.
11. Susanne Graf. Expression of time and duration constraints in sdl. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, LNCS, June 2002.
12. Susanne Graf and Ileana Ober. A real-time profile for uml and how to adapt it to sdl. In *SDL Forum 2003, July 1-4, Stuttgart*, number to be announced in LNCS, July 2003.
13. Susanne Graf, Ileana Ober, and Iulian Ober. Definition and translation of time annotations. Technical report, VERIMAG, 2003.
14. U2P Group. *UML 2.0 Superstructure proposal v.2.0.*, January 2003.
15. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
16. M.G. Harbour, M.H. Klein, R. Obenza, B. Pollak, and T. Ralya. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer, 1993.
17. D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002), Fort Worth, Texas*, 2002.
18. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.
19. Ilogix. Rhapsody development environment.
20. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 2000.
21. ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.
22. H. Jensen, K.G. Larsen, and A. Skou. Scaling up UPPAAL: Automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT 2000*, 2000.
23. Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking - Timed UML State Machines and Collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Oldenburg, Germany, September 9-12, 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.
24. Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining uml and formal notions for modelling real-time systems. In *Joint 8th European Software Engineering Conference, 9th ACM SIGSOFT*. ACM SIGSOFT, 2001.
25. Ileana Ober. An ASM semantics for UML derived from the meta-model and incorporating actions. In *Abstract State Machines - Advances in Theory and Applications.*, volume 2589 of LNCS. Proceedings 10th International Workshop, ASM 2003, 2003.
26. Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. Submitted to SVERTS 2003, 2003.

27. *OMG Unified Modeling Language Specification - Object Constraint Language* Version 2.0, 2003.
28. OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.
29. Metropolis project. <http://www.eecs.berkeley.edu/polis/metro>.
30. OMEGA IST project. <http://www-omega.imag.fr/>.
31. J. Ruf and T. Kropf. Symbolic Model and Checking for a Discrete Clocked Temporal Logic with Intervals. In *CHARME'97, Montreal, Canada*, pages 146–166, 1997.
32. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
33. Telelogic. *TAU Generation 2 Reference Manual*, 2002.
34. *OMG Unified Modeling Language Specification*, Version 1.4, June 2001.
35. S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.