Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A real-time profile for UML*

Susanne Graf, Ileana Ober and Iulian Ober

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

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper describes an approach for real-time modelling in UML, focusing on analysis and verification of time and scheduling related properties. To this aim, a concrete UML profile, called the OMEGA-RT profile, is defined, dedicated to real-time modelling by identifying a set of relevant concepts for real-time modelling which can be considered as a refinement of the standard SPT profile. The profile is based on a rich concept of event representing an instant of state change, and allows the expression of duration constraints between occurrences of events. These constraints can be provided in the form of OCL-like expressions annotating the specification or by means of state machines, stereotyped as "*observers*". A framework for modelling scheduling issues is obtained by adding a notion of resource and a notion of execution time.

For proving the relevance of these choices, the profile has been implemented in a validation tool and applied to case studies.

It has a formal semantics and is sufficiently general and expressive to define a semantic underpinning for other real-time profiles of UML which in general define more restricted frameworks. In particular, most existing profiles handling real-time issues define a number of predefined attributes representing particular durations or constraints on them and their semantic interpretation can be expressed in the OMEGA-RT profile.

1 Introduction

Building models which faithfully represent complex systems is a non trivial problem and a prerequisite to the application of formal analysis techniques. Usually, modelling techniques are applied at early phases of system development and at high abstraction level. Nevertheless, the need of a unified view of the various development activities and of their interdependencies motivated the so called model-based development [52] which heavily relies on the use of modelling methods and tools to provide support and guidance for system design and validation.

UML has become by now a standard in the domain of software development and starts to be adopted also in the domain of real-time and embedded systems. UML aims at providing an integrated modelling framework encompassing software structure and architecture, as well as behaviour descriptions. UML includes various behavioural notations, such as communicating state machines and action language. To cover real-time aspects, a profile for "Schedulability, performance and time" (SPT) [51] has been standardised. It defines most useful concepts, at least in an abstract manner. Nevertheless, it is incomplete, in the sense that it is biased versus a use in sequence diagrams. Notice also that in most existing frameworks based on or inspired by this profile, timing constraints are expressed without a well defined relationship to a functional model (e.g., in the form of a task model with periodicity constraints on tasks or by timed sequence diagrams) with the consequence that only the internal consistency of the task model can be analysed, not its conformance with the functional model.

1.1 Modelling time and timed behaviours

Typical categories of real-time properties that need to be expressed are

- *Time dependent behaviour*, which in practise is often modelled by means of timers or explicit read access to a system clock.
- Time related *assumptions* on the external environment of the system and the underlying execution platform, such as response times to requests, inter-occurrence times of events from the environment, execution times of actions, etc.
- Time related *requirements*, such as deadlines of actions (tasks), constraints on end-to-end delays, and more generally, constraints on durations between any two events.

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

** VERIMAG is an academic research laboratory associated with CNRS, Université Joseph Fourier and Institut Nationale Polytechnique de Grenoble

There exists a variety of semantic level modelling formalisms extended with time, such as timed Petri nets [60], timed process algebras [46] and timed automata [4]. In process algebras, timing is added by a "delay" construct similar to timeout or a wait construct, as they exist in real-time programming languages. In Petri nets, minimal and maximal waiting times are associated with states or transitions, and in timed automata, variables called "clocks" increase with time progress and can be set (to 0) so that they always measure the time passed since they have been last set. All these formalisms represent machines that can perform two kinds of state changes, *time progress steps* and *actions*, where actions correspond to *events*, that is instantaneous changes of the system state which may depend on time, whereas time steps do not alter the system state, but only time and clock values.

Contrary to real-time programming languages where time progress is considered as external - it can only be measured and decisions may be taken depending on the current time - modelling formalisms use a notion of *simulation time*. Contrary to external time, time progress may depend on system progress and can block, especially as a result of inconsistency of timing constraints. A typical example of use of simulation time is the assumption of "maximal system progress", as used for example in the synchronous approach [8], where time progresses only when no system step is enabled

For the expression of global real-time properties, there exist extensions of logic based formalisms for expressing real-time properties. Examples are TCTL [30] and TPTL [5], where temporal logics are extended similar to timed automata with clocks and constraints on their values.

Finally, there exist timed versions of scenario description languages, such as Message Sequence Charts (MSC) [33], event occurrences can be syntactically identified and their occurrence time or the duration between the occurrence times of two event occurrences can be constrained.

1.2 Defining a framework for Time in UML

The UML Real-Time profile SPT is a first step in answering OMG's request for a "*UML-based paradigm for modelling 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 interoperability between different analysis and design tools.*"[51]. It includes features for describing a variety of aspects of real-time systems, such as timing, resources, performance, etc.

The profile provides time related data types (*Time* and *Duration*), as well as features to express both local and global time constraints. The SPT profile defines

- a notion of *timed events* that has been mainly intended for defining constraints in timed scenarios,
- attributes of type duration, such as WCET (worst case execution time), defining particular constraints
- local constraints that can be expressed using concepts *timer* and *clock*.

SPT offers also some primitives for modelling resources (*Resource, ResourceUsage, ResourceManager*, etc), scheduling (*Schedule, Scheduling Policy, Schedulable Resource*, etc). However these concepts are abstract, and in order to be used in real projects, they need to be specialised into a concrete real-time framework.

UML 2.0 [53] includes, contrary to UML 1.4, some time related aspects, such as time related types and operational time-related concepts (*timer*, *clock*). But currently, there is no syntax for the expression of time constraints beyond time dependent conditions in actions (which include transition guards of state machines).

The aim of the profile defined in this paper is to make concrete and to generalise the ideas present in the SPT profile. Our profile represents a specialisation of the SPT, in that it fixes a precise semantics for a set of concepts defined by the SPT, and it gives them a concrete syntax. As SPT, it offers several means for expressing time constraints (operational and constraint based ones). Additionally, it introduces expressive event based time constraints and extends their use from timed scenarios to other behavioural formalisms, in particular timed state machines. Concretely, the profile includes the following features:

- Operational concepts, as they exist in most modelling languages for real-times systems: a notion of system time, which can be explicitly accessed in actions by an operator *now*, as well as *timers*, which can be armed and provoke a *timeout* event after a specified time, and *clocks* which can be set and then read to measure the time passed since the clock has been set (as in timed automata).
- A rich notion of *timed event* defining patterns of state changes occurring during execution, as well as a data structure defining a notion of observable state associated with the identified state change which includes its occurrence time. This set of events defines the *observability* with respect to time progress, and thus the set of expressible properties. In our profile, we make accessible some events which are implicit in many other approaches.
- A constraint-based formalism allowing to restrict the duration between occurrences of events, expressive enough to define all duration and constraint patterns as defined in SPT (such as ResponseTime associated with calls, WCET associated with actions, InterOccurrenceTime associated with events, etc.).
- Distinction between assumption and requirements by explicit qualification of constraints. In addition, a notion of observer is introduced. An observer is a state machine synchronising on events with two kinds of acceptance states, "*invalid*" and "*error*" states. An observer defines a (deterministic) automaton accepting sequences of occurrences of timed events. A sequence of events with an execution avoiding *invalid* states is *valid*. A valid event sequence whose execution passes through an *error* state violates a required property and represents an *error trace*.

The paper is organised as follows. Section 2 gives an overview on related work on the introduction of time in UML and related modelling formalisms. Section 3 is the main part and defines the real-time profile and its semantics. In section 4, a short overview on a tool implementing this profile is given as well as some perspectives for the future.

2 Related work

UML offers a variety of notations for capturing different aspects of software development. Real time issues have been addressed more recently in UML. An early attempt of adding time to UML is [19] which underlines the importance of time-related information in real-time systems and distinguishes between six kinds of time (absolute, mission, friendly, simulation, interval and duration), but does not define a concrete framework defining and using these distinctions. Also some UML-based CASE tools integrate timing aspects and several frameworks have been proposed defining real-time versions of a relatively small subset of UML: most of them consider state machines for describing behaviours, but timed extensions of OCL and entity-relationships are also considered. Some frameworks for scheduling and QoS have been defined as well.

Notice that several UML-based frameworks deal with *temporal* aspects, meaning by this that they address dynamic aspects and properties of (partial) *orders* of event occurrences. There are also UML-based frameworks for real-time systems which mainly address the need for particular communication and execution modes (e.g. [17] defines a UML profile for system design based on the synchronous approach) which is outside the scope of this paper.

2.1 Adding time in other modelling frameworks

The question of how to add real-time features to a modelling framework has been addressed also in other contexts. Formalisms for modelling of asynchronous systems, like ROOM [58] and the ITU[1] standard SDL [32] include basic concepts for time, time-related data types (*time* and *duration*), and *timers* for the specification of time dependent behaviour.

An interesting case is the framework defined at ITU for the joint use of SDL and Message Sequence Charts (MSC) [33] which are related to UML sequence diagrams. An operational model is specified in SDL, which includes features for expressing the information provided in class diagrams, architecture diagrams and state machines, whereas MSC are used to express requirement in the form of scenarios which should exist or represent undesirable behaviours. MSC include since the 2000 version occurrence time and time distance constraints in the standard (see for example [37,7,61, 39,20,66]). QSDL [18,45] defines an extension of SDL for performance analysis with probabilistic execution time constraints attached with tasks and a notation for some minimal deployment information. The proposal for extending SDL with time defined in [11,23,25] has strongly influenced the UML profile presented in this paper.

[1] International Telecommunication Union

2.2 Timing in commercial UML tools

Commercial UML-based CASE tools taking into account real-time are, for example, Rhapsody from I-Logix [31], ArTisan Real-Time Studio [6], Tau Generation-2 [63] and Rose-RT [55].

Rhapsody from I-Logix includes a notion of timer associated with state machines, measuring the time passed since the current state has been entered, for time dependent programming. A system consists of a set of activity groups representing a thread, where within each activity group concurrency is resolved deterministically. For individual activity groups sequential code is generated, whereas different activity groups are either distributed or scheduled assuming periodic tasks. This profile has strongly influenced the UML profile defined in the Omega project (see section 2.4).

ArTisan Real-Time Studio extends UML to model the system's reaction to events, time constraints, concurrent tasks and partitioning applications across multiple processors.

Rose-RT, an evolution of Room, and Tau Generation-2, an evolution of an SDL-based tool, are based on state machines communicating 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. Notice that the last two tools have more features of *modelling* tools (they allow for example, explicit non determinism).

2.3 Time extensions of subsets of UML

Several proposals of time extended versions of subsets of UML have been made for providing validation support by means of an existing validation method or tool.

State machine based approaches: Many approaches consider systems where behaviours are defined by state machines. The approaches described in [41,36,16] use UML state machines as a graphical representation of timed automata (and use class diagrams to provide the necessary type information). In particular, like in timed automata, time passes in states, whereas transitions are interpreted as instantaneous state changes (events) which can be constrained by time dependent guards. As a consequence, any event on which a time constraint is defined must correspond to an explicit transition in some state machine. In [36], the technique offered by standard UML is used that allows to specify that a transition is fired after a certain amount of time has passed since its source state has been entered. Communication may take time, but maximal communication time is a parameter defined uniformly for the entire model. Sequence diagrams are used to express properties. Both are translated into timed automata and verified using the model checking tool UPPAAL [40]. The approach in [41] uses the UML *after* statement and an extension of conditions with time to

express general guarded timeouts to represent timed state machines. This model is translated into first order temporal logic with real time. A subset of specifications are translated into timed automata and checked with the KRONOS model-checker [65]. In [16], hierarchical state machines using the action language of UPPAAL are translated first into hierarchical timed automata and then flattened to be verified with the Uppaal model-checker.

[3] addresses the verification of real-time behavioural patterns of embedded controllers by modelling with UML state machines both the controller and the system to be controlled. These state machines are compiled to synchronous active Java objects. Using the execution times of actions observed for this implementation on the target platform, timed automata are constructed in which actions are represented as pairs of transitions enforcing time progress by the observed execution time. On these timed automata, longest execution paths for reactions of the object to environment requests are computed and fed back into to original UML specification together with deadlines and other global timing requirements, from which than a more abstract time annotated global model is constructed and analysed. This is similar to an approach applied successfully to Esterel models in [12]. Such an approach should be supported by a real-time profile for UML.

In [59], events associated with communications (signal transmissions and operation calls) can be referred to by predefined names. Similar as in our profile, also implicit events, such as "*signal arrives in the destinator's event queue*" can be constrained, and not only events corresponding to some state machine transition. A timed semantics of a system is given in terms of a temporal logic with two independent next operators, one for system and one for time progress. It is not clear, in how far the use of such a polymodal logic leads to more interesting results than the use of timed automata, especially, as for verification only one of the two modalities can be used at a time.

OCL based approaches: [22] presents work on a UML profile for real-time constraints specifications based on OCL 2.0 [50]. It consists in extending the OCL 2.0 meta-model with concepts needed to express state chart configurations and sequences of them, so as to allow the expression of real-time temporal properties in OCL. The semantics of the temporal expressions is given in terms of a mapping to a real-time version of temporal logic [57].

Approaches addressing scheduling and QoS in UML: [44] specialises the SPT profile to better address RMA for distributed, critical and embedded applications domains, but it offers little opening to other real-time analysis techniques.

Metropolis [54] has been defined outside UML, but proposes also a UML profile. It defines a tool supported framework for scheduling of tasks that allows to express deadlines, execution times, scheduling policies, etc.

[34] addresses QoS prediction of systems defined by UML state machines extended so as to model stochastic decision processes. This is done by giving a stochastic interpretation to the standard *after* operator.

2.4 *Context of the* OMEGA-RT *profile*

The UML real-time profile presented in this paper has been developed in the context of the Omega project [13,24]. In Omega, we have defined a UML profile for real-time and embedded systems and supported it by several validation tools. This profile generalises the Rhapsody UML profile by increasing the potential of non determinism: set of objects can be grouped into *activity groups* representing a mono-threaded behaviour executing reactions to requests from the environment in run-to-completion steps. A reference semantics for the operational part of this profile is given in [15] and implemented in several tools.

This article presents the real-time part of the Omega profile, called in the sequel OMEGA-RT profile. Notice that it is mostly independent of the choices made for the operational semantics. Nevertheless, it relies on the existence of activity groups for defining the notion of concurrency needed for scheduling.

In Omega, several specialisations of this profile have been considered and tool support has been provided for them:

- [64] defines a sub-profile in which behaviours of objects are described by state machines extended with clocks and time guards. Its semantics conforming to the Omega operational and real-time reference semantics is expressed in terms of the typed logic of the interactive theorem prover PVS which is used for verification of real-time properties expressed in temporal logic or OCL. In this profile, with an operation call are associated two events, the moment at which the call triggers a transition in the callee and the end of the call, which are represented as synchronisations between caller and callee.
- [38] defines a sub-profile of the OMEGA-RT profile for OCL. The extension of OCL with a notion of *event history* can be used for defining arbitrary constraints on such histories. The subset of the events defined in section 3.2, associated with communications are considered there and identified using OCL concepts.
- [29] adapts Live Sequence Charts (LSC [14]) - a sort of sequence diagrams extended with mandatory/optional behaviour - to the OMEGA-RT profile by extending them with time guards and clocks progressing on an explicit *tick* event. Also the extension from instant level to type level is done, by quantifications on live lines [28].
  In this sub-profile, only communications via signal exchange is considered, and a signal transmission defines a single event (all the events defined in the OMEGA-RT profile define the same instant).
- [47,10,48] implements in the IF language and tool-set [9] the operational part of the Omega profile and a large part of the time extensions introduced here: in particular, timers and clocks, the event definition mechanism introduced in section 3.2, some of the duration expressions introduced in section 3.3 and simple constraints, as well as UML observers. This tool-set offers several possibilities for exploring a time extended system specification and for

formally verifying that the operational part of the model, enforced by the assumptions implies the requirements[2].

3 Framework for the definition of timed models

This section gives an overview of the constructs and notations introduced in the OMEGA-RT profile. This profile intends to make concrete the concepts defined in an abstract manner in the SPT profile and to increase their expressive power in order to address more general analysis techniques and in order to define a framework for the definition of a notion of consistency between different view points of a system.

The profile is based on the existence of two basic types, *time* representing time points or instants and *duration* representing distances between time points. Sets of instants and durations are generally expressed by means of predicates on attributes.

Operational concepts, as they exist also in UML 2.0 are not discussed here in detail. There is a notion of *clock* which can be set and then read to measure the time elapsed since the clock has been set, and a notion of *timer*, which can be armed and produce a *timeout* event after a specified duration.

A priori, we suppose the existence of a global reference time. *Local time* as proposed in SPT, may be defined by means of local clocks, for which a maximal *drift* and/or *offset* with respect to global time may be defined.

Section 3.2 introduces a mechanism for identifying events, section 3.3 introduces duration expressions and section 3.4 defines an OCL-based semantics for these notations. Section 3.5 defines a set of constraints and section 3.6 introduces observers as a more general framework for the definition of constraints and section 3.7 proposes additional notations for taking into account scheduling related constraints.

3.1 Running example

In order to illustrate both, missing features in the current version of the SPT profile and the features of the OMEGA-RT profile, we use a small example.

Consider an *Engine* displaying some information (temperature, rotating speed, etc.) provided by sensors on a *Display* device. A part of the structure of a model for such a system is shown in the class diagram in Figure 1. The requirements include the following time constraints:

(1) Between two consecutive calls made by an *Engine* to the operation *update* of its *Display*, less than 100ms pass if the *Engine* rotation speed (attribute *rpm*) exceeds 7000 at the moment the first call to *update* is made.
(2) Between the moment the engine temperature becomes critical (reception of signal *criticalTemperature* by the *Engine* from the *TemperatureSensor*) and the moment the engine reacts by decreasing its speed (invocation of the operation *accelerate* with a negative parameter by the *Engine* on itself) less than 50ms pass. Moreover, the *Display* shall receive an *update* from the *Engine* less than 20ms after the call to *accelerate*.

[2] formal verification is restricted to models with a finite state space.

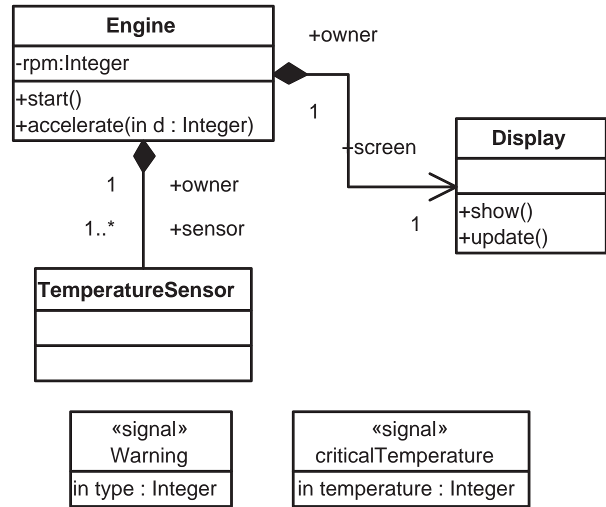

Fig. 1. UML class diagram for the example

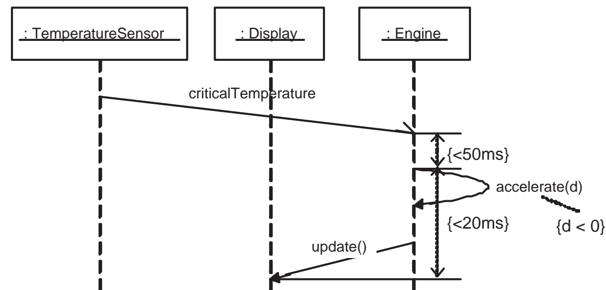

Fig. 2. Sequence diagram with a time constraint

These constraints cannot be captured by a standard UML model. In current practice, constraint (2) is likely to be represented by a sequence diagram, such as the one in Figure 2. But there are two problems with the sequence diagram representation of the constraint:

- In the sequence diagram, it is not clear whether the first event involved in the constraint is the reception of the *criticalTemperature* signal or the consumption of the signal by the *Engine*. Sequence diagrams offer no means to distinguish between these two events.
- There is no means in a sequence diagram to distinguish optional events and mandatory events: property (2) does not require a *criticalTemperature* signal to be sent, but if such a signal is received by the Engine, then a number of reaction events *must* happen within some limited amount of time.

3.2 Timed events

A purpose of the OMEGA-RT profile is the definition of time constraints not only at the instance level, as this is presently the case for timed sequence diagrams, but also at type level. A *TimedEvent* is an instant of state change. It is defined as a type level concepts and the corresponding instance level counterpart is an event instance defined as an attribute of a class or component of the system.

Event kinds define a syntactic classification of events. Each event kind is associated to a syntactic entity and it defines the relevant *parameters* of an event. For instance, in a signal exchange, three event kinds can be identified which make reference to the sender, the receiver, the concerned signal and its parameters:

- 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)

Only a *send* event and an *acceptsignal* event can be syntactically identified on a state machine defining the behaviour of the sender and the receiver, and the second one only under the condition that the signal effectively triggers an explicit transition (and is not implicitly discarded). A *receivesignal* event, representing the instant at which the signal reaches the receiver's input queue, cannot be syntactically captured in a state machine because the semantic level state change to which it is attached is not visible in the state machine[3].

An operation call defines six events (three corresponding to the invocation, and three corresponding to the return) which have the same kinds of parameters as the events associated with signals. With state machine states, *enter* and *exit* events are associated which can make reference to the object to which the state machine belongs and the name of the state as well as any visible attribute. The appendix provides an exhaustive list of the identified event kinds. They define the granularity of the states and the occurrence times of transitions between them that can be observed. When a finer granularity is needed, this has to be modelled explicitly (e.g., a transition with several actions needs to be cut into several transitions if an intermediate state needs to be observed.

When a coarser grained observation is sufficient, some events may be considered equivalent. In some contexts, it is not useful to distinguish between the *send* and the *receivesignal* event.

An *event type* represents a pattern of event occurrences and is defined by a UML class stereotyped with ≪*TimedEvent*≫. An *event instance* is either *local* to a class or component, or *global* to the model. A *Local* event instance of a class or component, can be defined as an attribute. *Global* events are defined as attributes of a special class, stereotyped ≪*TimeAnnotations*≫ which is never instantiated and is not part of the functional specification of the model, it only collects global time constraints and their events.

[3] This problem had also be identified in the context of SDL, and long, finally non conclusive discussions have taken place if and how to make it accessible to the user

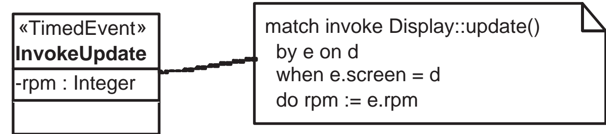

Fig. 3. Event types extracted from the constraints

An *event type* can have its own local *attributes* storing event parameters and information on the system state at event occurrence time (event memory). It is defined by an expression which may include

- a mandatory *matching clause* describing the *kind of event* and possibly names for some *event parameters* - specific to its kind (e.g. for a *send* event the signal that is sent, the sender, the target, etc.). The form of the matching clause depends on the kind of the event (Example 1 below shows the matching condition for events associated with operation calls, and an exhaustive list can be found in the appendix 5).
- an optional *filter condition* of the form

  when `b-expr`

  where the boolean expression *b-expr* can depend on the names introduced in the matching condition and on any attribute visible in the context of the event. The filter condition allows refining the specification of the event type defined by the matching condition. This way, event occurrences corresponding to the filtered event are a subset of those of the original event type. Therefore, the event filtering allows sub-typing.
- an optional *action statement* of the form

  do `action`

  where only allowed actions are those assigning a value to local event attributes with no other side effect. Notice that the value of a local attribute corresponding to a name used in the matching clause is implicitly defined by this use.

Example 1. Now we can define event types corresponding to the events referred to in the example of section 3.1. Property (1) refers to a single event, the *moment an engine calls the operation Update on its associated Display*. Figure 3 shows the definition of such an event type, called **InvokeUpdate**. The matching clause

match invoke *Display::Update* **by** *e* **on** *d*

expresses the fact that an event of this type matches any occurrence of an invocation of *Update* of any object *e* of type *Engine* to an object *d* of type *Display*. The filter condition

when *e.screen* = *d*

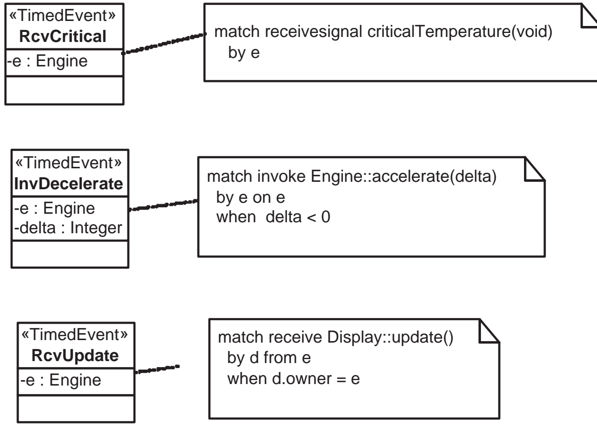

Fig. 4. Event types extracted from the constraints

restricts it however to those occurrences in which the call is made to the display defined by the engine's attribute *screen*. The action statement

do *rpm:=e.rpm*

has the effect that at occurrence of a matching event, the value of the attribute *rpm* of the event is defined as the value of *rpm* of the engine at this point of time (in the state before the occurrence of the event). This is needed as property (1) restricts the time constraint to those consecutive occurrences of events of this type in which at the time of the earlier occurrence the rotation speed exceeds 7000. Notice that the filter clause **when** *e.screen* = *d* is not necessary if it is impossible that an engine calls *Update* on an object different from "its *screen*".

Figure 4 shows the definition of the event types needed for property (2). The event type *RcvCritical* corresponds to the *moment at which the temperature gets critical* and is defined here precisely as the moment in which the signal *criticalTemperature* is *received* by the engine and not as the moment at which the engine reacts to it, which is ambiguous in the sequence diagram of Figure 2.

All events store also the identifier of the engine *e* corresponding to the sender or receiver identified by the matching clause. Section 3.3 shows why this attribute is needed. □

At any point of time of an execution, each event instance has like any object a "current value" corresponding the values defined by its most recent *occurrence*. In order to reason explicitly about older occurrences, we define event expressions which are either an event instance or of the form *E.pre* for some is an event expression *E*. The interpretation is that the "current occurrence" of *E.pre* is the second last occurrence of event *E*.

3.3 Duration expressions

The main aim of the OMEGA-RT profile is the definition of constraints on *durations* between occurrences of events. A difficulty is to provide a suitable mechanism for identifying appropriate pairs of event occurrences defining a *duration* to be constrained.

A possible mechanism consists in indexing events occurrences and defining the durations between occurrences of events with indexes in some relationship. E.g., the "duration between the ith *exit.state* and the $i+1^{th}$ *enter.state*" (defining the duration between two consecutive visits of state *state*). While this is useful, it is clearly not sufficient in situations where no relationship exists between the causal relationship of events occurrences and their index.

Example 2. The duration between the moment at which a signal is sent by a sender over an unreliable channel, and the moment at which it is received by a receiver can not be expressed by means of a constraint on indexes of events *SndSig* and *RcvSig* as there may be much more occurrences of the first event than of the second one. □

3.3.1 Basic durations

Our framework proposes several mechanisms for the identification of matching event occurrences. The simplest one defines the duration between the most recent occurrences of two event instances *E1,E2*:

Duration(*E1*, *E2*)

defines at any time, the time distance between the most recent occurrence of *E2* and the just preceding occurrence of *E1*.

Example 3. In the case that the events *exit.state* and *enter.state* concern a single object, one can express the duration between two consecutive visits of state *state* by **Duration**(*exit.state*, *enter.state*), and similarly, the duration between the moment that a signal is sent and the moment at which it is received, if not lost, by **Duration**(*SndSig*,*RcvSig*). □

It is useful to increase the expressiveness in two ways. On one hand, one would like to consider a subset of such durations. e.g. one may only be interested in the duration between *exit.state* and *enter.state* if the value of variable *x* has increased at least by 100. Such restrictions will be defined by constraints.

On the other hand, instead of the duration between the most recent occurrences of two events, one is interested in the duration between the most recent occurrences satisfying some condition. For this purpose, we extend duration expressions to

Duration(`event-expr`, `event-expr`)[`b-expr`]

The expression **Duration**(*E1*,*E2*)[*match-cond*] defines the duration between the most recent pair of occurrences ($\underline{E1}$, $\underline{E2}$) of the event expressions *E1*,*E2* satisfying *match-cond*($\underline{E1}$,$\underline{E2}$). Notice that **Duration**(*E1*,*E2*) is equivalent to **Duration**(*E1*, *E2*)[true] and we also use an alternative syntax:

Duration(*E1*,*E2*) **match** *match-cond*

In a pipelined computation, where each occurrence of *E1* is followed by exactly one corresponding occurrence of *E2* by preserving the order of the *E1* occurrences,

DurationIndexed(*E1*, *E2*))

measuring the time spent between the ith occurrence of an event *E1* and the ith occurrence of *E2*, defines the useful durations to be constrained.

Example 4. Let us explain the use of these duration expressions on hand of the examples.

Property (1) of the example of section 3.1 defines a constraint which is local to the class *Engine*, and therefore the instances of the events needed for its expression are defined as *local* attributes of *Engine* (see Figure 5). Such a local event instance does only match occurrences concerning the object to which it is attached, and it can only be used in local properties. The duration occurring in property (1) is then defined by

Duration(*ev1.pre*,*ev1*)

where *ev1* is a local attribute of type *InvokeUpdate* of *Engine* It represents the duration between two consecutive occurrences of event *InvokeUpdate* local to each object *Engine*.

Property (2) of the same example involves events of several object, meaning that it needs to be expressed by means of global events with instances attached with class ≪*TimeAnnotations*≫. When there is more than one instance of *Engine* in the system, it is necessary to restrict the durations between events to those associated with the same engine. The duration between the starting point of the deceleration of an engine and the update of the associated display can be defined by

Duration(*evDec*,*evUpd*)
match *evDec.e*=*evUpd.e*

For *evDec* a global event instance of type *InvDecelerate* and *evUpd* an attribute of type *RcvUpdate*. The **match** clause says that we are not interested in the closest occurrences of *InvDecelerate* and *RcvUpdate*, but in the closest ones concerning the same engine *e*.

In this example, a notion of *component* containing a single engine, its screen, and possibly other objects, and the association of events and timed annotations with components allows avoiding the use of the **match** clause. In a sliding window protocol, however, an interesting duration is the one between those events *sendM* and *rcvAck* carrying the same sequence number. Using a duration of the form **Duration**(*E1*,*E2*) requires the introduction of a different event type for each sequence number, whereas the use of a **match** clause allows to simply write

Duration(*sendM*,*rcvAck*)
match *sendM.sn*=*rcvAck.sn* □

3.3.2 Predefined durations

SPT associates durations and constraint of them by associating them to features such as operations, signals, objects, etc. Often, additional features, in particular operations need to be introduced for representing the relevant durations. We propose the explicit definition of events for more expressiveness and flexibility for definition of durations. Nevertheless, such duration patterns are useful shorthands. Examples are

– execution time, execution delay, client response time, server response time, transmission delay which are associated with actions (a call action for the last two)
– reactivity and period which are associated with a trigger
– transmission delay associated with a communication channel
– lifetime associated with an object, and many more.

Our profile does not yield completeness with respect to predefined durations, which is hardly ever possible. The idea is to define the useful ones in any particular context. What our profile provides, is a means for a semantic underpinning of any patterns of this kind. It might even be envisaged to allow the user to define his own patterns for a given application or application domain, and defining their semantics using our profile.

Notice that a particular feature of the above mentioned patterns is that they concern events attached to the same object, which simplifies the definition.

Example 5. For instance, the client response time of an operation *warning* of class *Engine* can be associated with provided interfaces or with a particular call to *warning* (for example a transition triggered by a call to *warning*).

warning.**ResponseTime**

defines the time elapsed between the reception of a call of *warning* and the moment at which the return statement is executed. This duration defines implicitly two event types:

Ev1: **match Receive** *Engine*::*warning*
from *sender*
Ev2: **match InvokeReturn** *Engine*::*warning*
to *sender*

When there is never a call to *warning* before the preceding call has been completely treated, the expression

Duration(*ev1*,*ev2*)

on implicitly defined local attributes *ev1*:*E1* and *ev2*:*E2* of *Engine* defines the required client response time.

In the general case, where there may be several calls which have been received but not yet treated, the expression above is not correct as it defines the duration between an *Ev1* and a directly following *Ev2* event. In the context of Omega, where calls are always considered as blocking, it is enough to store the *sender* of each event occurrence and to define the response time by means of the expression

Duration(*ev1*,*ev2*) **match** *ev1.sender*=*ev2.sender* □

3.4 Semantics

A large part of the profile is implemented by a mapping to the extended timed automata of the IF formalism [9] which has a formal semantics. But, in order to have the profile better integrated with the UML and SPT definitions, we provide a formalisation in UML itself by means of OCL. For increasing the readability, however, we rather use an *OCL-like* notation which can be transformed into OCL in a straightforward

manner. Time constraints express constraints on occurrences of events, whereas OCL allows the expression of properties of configurations, which are interpreted as invariants, that is as a property of all configurations traversed by any execution.

A purpose of the introduction of events is to make time constraints dependent on the defined events, but not directly on the entire state of the system. That means that events define the set of useful *observations* for reasoning about timing constraints.

When OCL is used for defining semantics, event instances have to be interpreted as *objects* which have a value in every semantic level state. Where only duration expressions making reference exclusively to the *most recent occurrence* of an event expression, it is sufficient to consider any attribute *E* of type event as an object changing its value (its local attributes and its occurrence time, denoted *E.t*), at each *occurrence*. The event expression *E.pre* represents an object holding at any time the "previous value" of *E*. In order to be able to define the semantics of duration expressions containing a **match** clause as an invariant, this is not sufficient.

3.4.1 Events

We consider that every attribute *E* of type event, represents at each instant of execution a list of *event occurrences* with an accessor *E.latest* representing the most recent occurrence and *e.prev* defining the predecessor of every given event occurrence *e* in the list. Each list is empty at system start. *E.pre.latest* is defined by *E.latest.prev* pointing to the predecessor of *E.latest*.

We also introduce a derived expression *E.preN(n)* defining the *n* before last element of the list associated with an event expression *E*; it is not accessible at the user level, but needed for the definition of the semantics of duration expressions. It is expressible in OCL by means of the defined primitives:

E.preN(n) = **if** n=0 **then** *E.latest* **else** *E.pre.preN(n-1)*

This mapping is close to the extension of OCL with a *history variable*, representing in every instant of execution the current history of all event occurrences [38]

3.4.2 Duration expressions

Duration expressions are evaluated on a configuration of events, that is the before mentioned lists of event occurrences defining the semantics of a set of events.

The semantics of an expression of the form **Duration**(*E1*, *E2*)[*match-cond*], is defined by:

Duration(*E1*,*E2*)[*match-cond*] =
  if $\exists i \in \mathbb{N}$: *match-cond(E1.preN(i),E2)*
  then let $k = min\{i \in \mathbb{N} \mid$ *match-cond(E1.preN(i),E2.latest)*$\}$
    in $E2.latest.t - E1.preN(k).t$
else Duration(*E1*,*E2.pre*)[*match-cond*]

where the minus represents the distance operator on type *Time* having as result a *duration*.

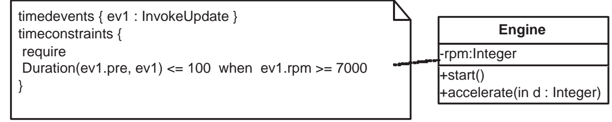

Fig. 5. Expression of property 1

In order to define the semantics of the index-wise duration, we need a function *index* that defines the index of each element in the occurrence list associated with an event expression *E*. Its definition needs an additional primitive *E.first*[4] defining the first element of the list associated with an event.

$E.index = \{i \in \mathbb{N}^+ \mid E.preN(i-1) = E.first\}$

The index-wise duration can now be defined as:

DurationIndexed(*E1*,*E2*) =
  Duration(*E1*,*E2*)[*E1.index* = *E2.index*]

3.5 Time constraints

Time constraints are boolean expressions involving durations. In explicitly time dependent models with timers and clocks, boolean expressions involving timers and clocks can be used in the action language, in particular in guards or in decisions. In principle, any boolean OCL expression can be used in timed annotations. [47], the tool providing the most complete tool support for this profile, handles only a subset of simple constraints corresponding to conditional constraints involving two events. They extend constraints as they are used in SPT. Constraints involving more than two events are expressed by observers[5] in section 3.6. The considered expressions are of the form

`duration-expr` $\approx$ `duration-constant`
  when `b-expr`

where $\approx$ is any comparison operator[6] and `b-expr` may depend on the attributes of the events occurring in `duration-expr`. It expresses the following invariant: in any state, the value of `duration-expr` must satisfy the constraint expressed by `duration-expr` $\approx$ `duration-constant` under the condition that `b-expr` holds for the attributes of the event occurrences identified by the evaluation of `duration-expr`. As constraints depend only on the implied events, it is enough to evaluate them at occurrences of the second event in `duration-expr`.

Example 6. Property (1) of the example defined in section 3.1 is shown in Figure 5. It constrains the duration between the two most recent occurrences of the event *InvokeUpdate* to be smaller than 100, but only when attribute *rpm* of the older occurrence is greater than 7000. Property (2)

[4] which is the element *E.preN(k)* such that *E.preN(k+1)* is undefined, but such a definition is not allowed in OCL. Notice that we could have defined a semantics including a pointer to the initial state, but for exploration based validation it is important to show that this information can be derived

[5] SPT proposes sequence diagrams for the same purpose

[6] $\approx \in \{<, \leq, =\geq . >\}$

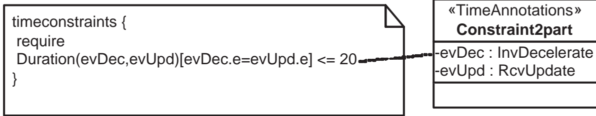

Fig. 6. Second part of property (2)

involves three events. The constraint shown in Figure 6 expresses the second part of this property, but it is only required if beforehand a signal *criticalTemperature* has been received and deceleration has started within the required reaction interval. □

Constraints in time annotations define *invariants* of the model and can play two different roles:

- they may represent assumptions on the environment or the underlying execution platform,
- or requirements, that is properties which should be derivable from the model

We distinguish these two kinds of constraints by means of explicit keywords **assume** and **require**. The constraints shown in the example are all requirements.

SPT offers tagged values that represent attributes of the execution (deadline, WCET, etc) to express time constraints. Our profile does not define such attributes, but it provides the semantic basis allowing their formal definition. For instance, referring to the example in section 3.3.2, the deadline attribute for an operation may in our setting define a constraint of the form

warning.**ResponseTime** $\leq$ deadline

where deadline is an attribute of type duration of the class engine. We allow more fine grained constraints.

3.6 Observers

In order to express assumptions and requirements involving conditions which are more complex than the distance between two events, we define an operational formalism called *observers*.

3.6.1 The *observer* concept

Observers represent dynamic properties by acceptors of languages representing event sequences. An observer is an object which executes synchronously with a system and monitors its state and the events that are occurring. It may have a local memory (*attributes*) and its behaviour is described by a *state machine*. Contrary to objects of a system, the observer's state machine does not react to local conditions or to signals exchanged with other objects, but to events - as they have been defined earlier - occurring in the system's execution and change of conditions satisfied by the system's state.

Observers have unlimited visibility over the objects composing the system (they may observe state machine states, attribute values, etc of any object) and mechanisms for capturing occurrences of events of the types defined in section 3.2.

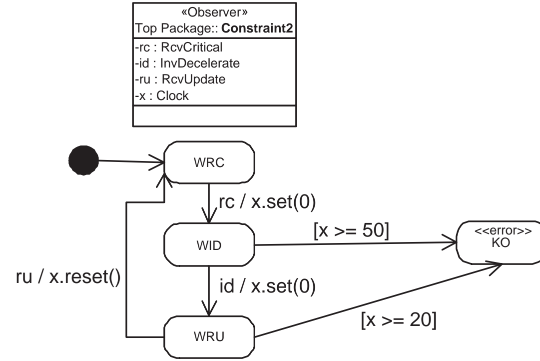

Fig. 7. Example of observer for a safety property.

In order to express assumptions and requirements, some of the states of the observer's state machine be qualified as *invalid* or *error* states:

- *invalid* states express the violation of *assumptions* on system executions. A system execution which may drive[7] an observer through an *invalid* state is not considered as a part of the semantics of the system, only complete executions avoiding *invalid* states are *valid* executions.
- *error* states express requirements (properties) that have to be satisfied by the system, i.e. that have to be implied by the functional semantics of the system under the specified assumptions. A *valid* execution which may drive an observer through an *error* state represents a violation of the requirement.

3.6.2 Syntax

Syntactically, observers are defined in a UML model by stereotyping classes with ≪*observer*≫. The states of the state machine are classified using two stereotypes: ≪*invalid*≫ and ≪*error*≫. Finally, reactions to events are specified in transition triggers using the same syntax as defined in section 3.2. A trigger may be either just a *matching clause* (as used in the definition of event types) or an *event expression* where a corresponding instance must be declared locally as an attribute of the observer class of some *event type*).

Example 7. The observer in Figure 7 specifies property (2) from section 3.1 in a system with a unique object of type engine. It involves three events as defined in Figure 4 and has a local attribute for matching each of them.

After the occurrence of an event of type *RevCritical*, it expects to observe the occurrence of an event *InvokeDecelerate*. If no observable event occurs before 50 ms have passed, the transition leading to the *error* state is executed. The second part of the observer (the behaviour in state *WRU*) represents the constraint of Figure 6. The observer defines precisely the condition on previous observations under which the constraint must hold. We show here the alternative of using observers with attributes of type *Clock* as an alternative of

[7] In general, we consider deterministic observers, so that the notions *may drive* and *must drive* coincide

the use of the occurrence time attribute of events: to express the constraints on durations, this observer uses a unique attribute *x* of type *Clock*. This is enough, as at any time only the duration since the previous observation is needed. Whenever an observable event occurs triggering no transition — e.g. an event *rc* in state *WID* — it is simply ignored.

When several engines, possibly created and deleted dynamically during the lifetime of the system, should have the same property, then the observer class needs an additional attribute of type engine, and only events of "its engine" will trigger transition (by means of an additional guard on transitions). An additional observer creates/deletes an instance of such an observer, whenever it detects the creation/deletion of an engine. □

Notice also that the use of observers is not restricted to timing properties. They represent also an interesting means for the expression of more complex, data- and coordination-oriented properties.

3.7 Scheduling-related concepts

Constraints on durations between events, as they have been defined so far, do not allow to distinguish between distributed and scheduled execution. Duration constraints of causally independent parts are independent. Going towards an implementation means to consider scheduling constraints, that is, taking into account sequentialization of executions of independent actions due to a restricted number of processors executing them or the need to share other resources. In order to add this information to the model, a notion of *resource* is introduced, as well as primitives for binding activities to resources. When the envisaged scheduling policy of independent actions on a (set of) processors is preemptive, the so far introduced notion of "duration" of the execution of an action is no more sufficient. It is necessary to distinguish for each *basic* action its *duration* and its *execution time* When only non-preemptive scheduling is considered, *execution time* and *duration*[8] of actions coincide, but it is necessary to define the granularity of scheduled *tasks* or *threads*, that is the points of control at which the scheduler takes decisions.

3.7.1 Task

We propose to use the notion of activity groups as they exist in the Omega profile to define the set of concurrently active thread to be scheduled. In the Omega profile, an *activity group* is a set of objects grouped around an instance of an active class which represents a single threaded behaviour in which the reaction to an external trigger is computed in a run-to-completion step, without further acceptance of requests from the environment. Activity groups do not share variables, and therefore all tasks are executed on the same processor can be scheduled with or without preemption. At semantic level, an activity group is in an *inactive* state, when all its objects are stable (have no enabled transition). The presence of a signal or an operation call in the input queue and/or the satisfaction of a time dependent condition may activate one of its tasks and bring the activity group into an *active* state.

When preemptive scheduling is used, activity groups are sufficient to define the scheduling problem: at every point of time, the scheduler decides which activity group(s) get their resource(s).

In the context of non preemptive scheduling, we may either use the notion of run-to-completion step for implicitly defining the granularity of *atomic tasks* or use an explicit break-down of the control flow of a step into user defined tasks using primitives *starttask* and *endtask*.

Notice that each run-to-completion step may itself consist of concurrent sub tasks and must be scheduled, possibly by a local scheduler[9] (see Section 3.7.4).

In the context of UML 2.0, architecture diagrams can be used to define a hierarchical notion of *activity group*, *active state*, *task*, ...

3.7.2 Resource

Resources represent mutual exclusion constraints which are not defined in the functional model, in particular shared resources (like processors) related to the computation platform which have no functional meaning but determine timing properties. As in SPT, for such resources an explicit notion of resource is introduced as classes stereotyped ≪*resource*≫. Resources have attributes defining the kind of scheduling that they allow (no scheduling, preemptive of non-preemptive). Instances of resources have no explicit behaviour associated.

Due to dynamic nature of UML specifications, we propose a dynamic means for specifying the resources that a task needs to execute: as for timing we propose

– an operational way by means of the actions *acquire(r)* - from that point on resource *r* is needed to execute - and *release(r)*.
– in the form of annotations associated with events. This has the advantage of providing a better separation between functional and non functional specification where nevertheless the relationship is precisely defined by means of events.

At semantic level, resources are only required in the *active* state, which has two sub-states: in the state *executing* state, it owns all required resources in mutual exclusion, whereas in the *suspended* state, it is waiting to obtain some resource.

All objects of the system which can be accessed concurrently by more than one object to execute requests represent resources. In principle, accesses to such a resource are specified in the functional model, and it is not necessary to explicitly introduce a ≪*resource*≫ for them. But when several

[8] duration refers here to the duration between the events *start action* and *end action* not the duration of an entire transition, starting from the reception of the trigger

[9] In current practise, each run-to completion step is scheduled off-line, and sequential code is generated

actions need to be executed atomically, then it is important to be able to specify this. For this kind of mutual exclusion, the profile introduces a notion of *atomic* action, blocking the access from other *activity groups* to any object in between to accesses by the atomic action.

3.7.3 Execution time

The *execution time* of an action is defined relative to an activity group as the cumulated time in the *executing state* between the start and the end event associated with the action. Execution times can be specified by an expression of the form

ExecTime(*E1*, *E2*)

where *E1* and *E2* concern the same object. Alternatively, the execution time of a *task* associated with an activity group is associated with a trigger and is implicitly defined as the time between the acceptance of the trigger and the moment in which the activity group enters again an *inactive* state.

3.7.4 Scheduling policies

Scheduling is the sequential ordering of concurrently enabled activities, which share resources. A well defined theory for solving the scheduling problem is defined only in restricted settings [42, 56, 62, 43], but there exist recent results on more general frameworks [21, 1, 35, 2]. A general framework for expressing all kinds of scheduling constraints as well as some results on schedulability are given in [26, 27].

Any scheduling policy (including RMS, EDF, ELF, ...) can be expressed by means of dynamic priority rules [27]. In order to provide expressiveness, our framework includes *dynamic priority rules*, that we view as complement to scheduling policies defined by keywords such as proposed by the SPT. A priority rule is of the form

$[c1::]p1 < [c2::]p2$ **if** *b-expr*

where *p1, p2* represent any object of class *c1, c2* and *b-expr* is a condition on the current state, where the typing information is optional when *b-expr* can be evaluated on any object.

Example 8. Dynamic priority rules are useful for both, the specification of scheduling policies and of execution modes. For example, the run-to-completion execution mode for activity groups can be specified by the following rule:

$p1 < p2$ **if** $p2 = p1.manager$

This rule says that the manager object[10] of an activity group has less priority than other objects of the group. This is due to the fact that the task of the manager object consists in accepting new requests which is only allowed when the previous request has been handled. □

As the non determinism to be resolved by a scheduler might appear at any level of the model hierarchy, also priorities are organised in a hierarchical manner: rules for eliminating non determinism amongst concurrent entities are attached with an entity at higher level of hierarchy. For example,

- the priority rules between activity groups executing on the same *resource* are associated with the *resource*;
- the priority rules defining the choice between the enabled triggers within an activity group are associated with the *active object*. This way, an active object is a kind of local scheduler.

4 Conclusions

We described an approach for enriching UML models with time and scheduling related information. The defined framework has a semantic underpinning and is compatible with the *UML Real-Time profile for Performance Scheduling and Real-Time* in the sense that it uses the concepts identified there to define a concrete and expressive framework.

A key point is the introduction of a concrete UML syntax for a set of *events* identifying the instants during the lifetime of the system under some time constraint. Duration expressions represent the time elapsed between certain occurrences of events. Our profile goes much further than SPT concerning the set of identifiable event occurrence pairs that can be constrained.

In fact, the profile defined here can be used to provide a precise semantics for durations and constraints defined in SPT and some of its extensions in the form of attributes. SPT does not fix an interpretation, but leaves this to the tools. The semantics used by different tools can be expressed using our profile. Our profile provides a means for defining such a semantics by means of a set of events and modalities for expressing durations.

The introduced notion of event is rich enough to define an observation criterion for verifying timed properties of the system: the semantics of timing properties depends only on the set of defined events, meaning that any abstraction of the system preserving the observations of events is sufficient to verify the time related properties.

In fact, events are the basis for the definition of the consistency relation between different views of the system. In [47], we use them to define the relationship between a global view provided by observers and an operational view provided by state machines defining object behaviours, but in the same way, they can also been used to relate Sequence Diagrams to an operational view. This means also that profile is not bound to a particular interpretation of operational specifications. Time constraints and observers can be evaluated on any set of behaviours defined by sequences of occurrences of observable events.

An important contribution of our profile, are the introduction of UML *observers*. They are more powerful than sequence diagrams for the expression of global properties and have been used successfully in other contexts, for example together with SDL. Here, we have shown their use for the expression of timed properties, but observers - together with events - can also be used for the expression of complex behavioural properties of a system. For example, an observer may use the event *InvokeUpdate* not only to define a time

[10] in the Omega profile each object has an attribute *manager* which has value *self* for active objects and the identity of an active object for instances of passive classes

constraint between the occurrence time of two consecutive events, but also to limit the difference of the rotation speed between two occurences of this event.

An important difference with the SPT profile, in particular sequence diagrams, is that time extensions are defined at class level and not at object level. This increases the expressiveness, but explains also some increase in complexity.

As already mentioned in section 2.4, parts of this profile have been adapted and supported by verification tools. Tool support for most of the profile is provided by the tool described in [47]. This tool represents UML specifications respecting the restrictions of the Omega profile by means of a dynamic set of communicating timed automata extended with data and actions, as they are defined by the IF format [9]. All events correspond to a set of control transitions in the corresponding IF specification; taking such a transition generates the corresponding event, but all the information attached with an event is stored in the constraints (observers) depending on this event, rather than in a event occurrence list defined in section 3.4. UML observers are represented by IF observers. Timed annotations which are local to an object are represented by additional local attributes and timed annotations of the extended automaton representing the the behaviour of the object. Global time constraints are represented by IF observers. The naive representation of constraints involving duration expressions with matching conditions by timed automata is impractical for validation as it results in the dynamic generation of a large number of instances of timed observers. The main challenge consists in finding criteria for the deletion of observers, when either the expected event can not be observed anymore in the future, or its observation beyond some point of time doesn't change the validation result. Such criteria can be found by means of static analysis of the system.

Using this tool-set, we have successfully used our profile for the case studies considered in the Omega project. In particular, we have applied it successfully to a model of the Ariane-5 flight program for the verification of time depending properties and for schedulability analysis. A more detailed description of this case study can be found in [48], and the scheduling issues will be presented in a forthcoming paper.

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. On optimal scheduling under uncertainty. In *Proceedings of TACAS 2003, Warsaw*, LNCS, 2003.
2. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. In *accepted to TCS*, 2004.
3. J. Aghav and C. Petitpierre. Validating real-time behavioral patterns of embedded controllers. In *SVERTS - Specification and Validation of UML models for Real Time and Embedded Systems, workshop at UML 2003, CA, USA, October 2003, Proceedings*, 2003.
4. R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
5. R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41:181-204, 1994. (a preliminary version appeared in the Proc. 30th FOCS 1989).
6. *Artisan Real Time Studio*, 2001.
7. H. Ben-Abdalla and S. Leue. Expressing and analysing timing constraints in message sequence chart specifications. Technical report, U. of Waterloo, 1997.
8. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
9. Marius Bozga, Susanne 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*, number 2404 in LNCS. Springer Verlag, June 2002.
10. Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM-04:RT 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, number 3185 in LNCS, June 2004.
11. Marius Bozga, Susanne Graf Alain Kerbrat, Laurent Mounier, Iulian Ober, and Daniel Vincent. Timed extensions for SDL. In *SDL Forum 2001*. LNCS, June 2001.
12. E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the developpment and verification real-time embedded systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. CAV'01, LNCS 2102*. Springer, 2001.
13. OMEGA Consortium. Webpage of the OMEGA IST project.
14. 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.
15. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.
16. Alexandre David, Oliver Möller, and Wang Yi. Formal verification UML statecharts with real time extensions. In *Proceedings of FASE 2002 (ETAPS 2002)*, volume 2306 of *LNCS*. Springer-Verlag, April 2002.
17. Robert de Simone and Charles André. Towards a "Synchronous Reactive" UML profile. *STTT, Int. Journal on Software Tools for Technology Transfer*, 2005. In this volume.
18. 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.
19. Bruce Powel Douglass. *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Object Technology Series. Addison-Wesley, 1999.
20. N. Faltin, L. Lambert, A. Mitschele-Thiel, and F. Slomka. An annotational extension of message sequence charts to support performance engineering. In *8th SDL Forum*. North-Holland, 1997.
21. Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability anaysis using two clocks. In *ETAPS 2003*, 2003.

22. 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.
23. Susanne Graf. Expression of time and duration constraints in SDL. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, number 2599 in LNCS, June 2002.
24. Susanne Graf and Jozef Hooman. Correct development of embedded systems. In *European Workshop on Software Architecture: Languages, Styles, Models, Tools, and Applications (EWSA 2004), co-located with ICSE 2004, St Andrews, Scotland*, LNCS 3047, pages 241–249. Springer-Verlag, May 2004.
25. 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*, volume 2708 of *LNCS*, July 2003.
26. Gregor Gössler and Joseph Sifakis. Component-based construction of deadlock-free systems. In *proceedings of FSTTCS 2003, Mumbai, India*, LNCS 2914, pages 420–433, 2003. downloadable through http://www-verimag.imag.fr/ sifakis/.
27. Gregor Gössler and Joseph Sifakis. Priority systems. In *proceedings of FMCO'03*, LNCS 3188, 2004.
28. D. Harel, H. Kugler, and A. Pnueli. Smart Play-Out Extended: Time and Forbidden Elements . In *International Conference on Quality Software (QSIC04)*, pages 2–10. IEEE Press, 2004.
29. 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.
30. Eyal Harel, Orna Lichtenstein, and Amir Pnueli. Explicit clock temporal logic. In *In Proceedings, 5th IEEE Symposium on Logic in Computer Science, LICS 90, Philadelphia, Pennsylvania*, pages 402–413. IEEE Computer Society Press, 1990.
31. Ilogix. Rhapsody development environment.
32. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, November 2000.
33. ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.
34. David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A QoS-oriented extension of UML statecharts. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 76–91. Springer, 2003.
35. Christos Kloukinas and Sergio Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, ISBN 0-7695-1936-9, 2003.
36. 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.
37. P. Kosiuczenko. Formalizing time aspects in message sequence charts. Technical report nr. 9703, Ludwig-Maximilians-Universität München Institut für Informatik, 1997.
38. Marcel Kyas and Frank S. de Boer. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Compositional Verification in UML*, volume 101 of *ENTCS*, pages 73–93. Elsevier, 2004.
39. J.-L. Lambert. PMSC for performance evaluation. In *1st Workshop on Performance and Time in SDL and MSC*, Technical Report 1/98, IMMD VII, University of Erlangen-Nürnberg, 1998.
40. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.
41. 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.
42. J.W. Layland and C.L. Liu. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
43. Giuseppe Lipari and Giorgio Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of System Architecture, Special Isssue on Real Time Systems*, 2000.
44. Jean-Noël Meunier, Frank Lippert, and Ravi Jadhav. RT modelling with UML for safety critical applications: the HIDOORS project example. In *SVERTS - Specification and Validation of UML models for Real Time and Embedded Systems, workshop at UML 2003, CA, USA, October 2003, Proceedings*, 2003.
45. A. Mitschele-Thiehl and B. Müller-Clostermann. Performance engineering of SDL/MSC systems. *Computer Networks 31(17): 1801-1815*, 1999.
46. X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In *Proc. CAV'91*, volume 575 of *LNCS*. Springer-Verlag, July 1991.
47. Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. In *11th International SPIN Workshop on Model Checking of Software, 2004*, volume LNCS 2989, 2004.
48. Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *STTT, Int. Journal on Software Tools for Technology Transfer, 2004*, 2005. In this volume.
49. Iulian Ober, Susanne Graf, Ileana Ober, and David Lesens. Un profil UML et un outil pour la modélisation et la validation de systèmes temps-réel. *numéro spécial du journal Génie Logiciel (ISSN 0295-6322) consacré à la Journée NEPTUNE 05 : Ingénierie des Modèles - vérification de modèles.*, 2005. accepted for publication.
50. *OMG Unified Modeling Language Specification - Object Constraint Language* Version 2.0, 2003.
51. OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.
52. OMG. Model Driven Architecture. http://www.omg.org/mda, 2003.
53. OMG. *UML 2.0 Superstructure proposal v.2.0.*, January 2003.
54. Metropolis project. http://www.eecs.berkeley.edu/ polis/metro.
55. Rational/IBM. Rose real-time development environment.
56. Ragunthan Rjakumar, Liu Sha, John Lehoczky, and Krithi Ramamritham. *Advances in Real Time Systems*, chapter An optimal priority inheritance policy for synchronization in real-time systems. Prentice-Hall, 1995.
57. 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.

58. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
59. Subash Shankar and Sinan Asa. Formal semantics of UML with real-time constructs. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 60–75. Springer, 2003.
60. J. Sifakis. Use of Petri Nets for Performance Evaluation. In *Proc. 3rd Intl. Symposium on Modeling and Evaluation*, pages 75–93. IFIP, North Holland, 1977.
61. F. Slomka, J. Zant, and L. Lambert. Msc-based schedulability analysis. In *1st Workshop on Performance and Time in SDL and MSC*, Technical Report 1/98, IMMD VII, University of Erlangen-Nuremberg, 1998.
62. Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real Time Systems*, 1996.
63. Telelogic. *TAU Generation 2 Reference Manual*, 2002.
64. M. van der Zwaag and J. Hooman. A semantics of communicating reactive objects with timing. *STTT, Int. Journal on Software Tools for Technology Transfer*, 2005. In this volume.
65. S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.
66. Tong Zheng and Ferhat Khendek. Time consistency of MSC-2000 specifications. *Computer Networks*, 42(3):303–322, 2003.

5 Annex: List of event kinds

This annex contains the list of identified event kinds. as well as the corresponding matching statement, which defines its parameters and implicit **do** statements.
Event kinds associated with a *signal transmission*:

- *send* - instant of signal emission:
  match send <signal>(<ident-list>)
  by <ident> **to** <ident>
  where the optional **by** identifier represents the sender, the **to** identifier the receiver, <signal>(<ident-list>) designates a signal and its parameters.
- *receivesignal* - instant of signal reception:
  match receivesignal <signal>(<ident-list>)
  by <ident> **from** <ident>
  where the **by** identifier represents the receiver, the **from** identifier the sender
- *acceptsignal* - instant of start of signal processing by the receiver:
  match acceptsignal <signal>(<ident-list>)
  by <ident> **from** <ident>
  parameters as for **receivesignal**.

Event kinds associated with an *operation call*:

- *invoke* - instant of call by the caller:
  match invoke <class>::<operation>(<ident-list>)
  by <ident> **on** <ident>
  where the optional **by** identifier represents the caller, the **on** identifier the callee and <class> the optional callee class and <operation>(<ident-list>) a method of the callee with its parameters.
- *receive* - instant of call reception by the callee:
  match receive <class>::<operation>(<ident-list>)
  by <ident> **from** <ident>
  where the optional **by** identifier represents the callee, and the **from** identifier the caller
- *accept* - instant of start of operation execution:
  match accept <class>::<operation>(<ident-list>)
  by <ident> **from** <ident>
- *invokereturn* - instant of emission of "return":
  match invokereturn <class>::<operation>(<ident-list>)
  by <ident> **to** <ident>
  where the optional **by** identifier represents the callee, the **to** identifier the caller of the operation.
- *receivereturn* - instant of reception of "return" by the caller:
  match receivereturn <class>::<operation>(<ident-list>)
  by <ident> **from** <ident>
  where the optional **by** identifier represents the caller, the **from** identifier the callee
- *acceptreturn* - instant of call termination:
  match acceptreturn <class>::<operation>(<ident-list>)
  by <ident> **from** <ident>

Event kinds associated with an *action specification*:

- *start* - the instant of start of action execution
- *end* - the instant of end of action execution
- *startend* - instant of execution, if instantaneous action:
  match {**start** | **end** | **startend**} <class>@<label>
  where <class> designates the class of the action and <label> its label.

Event kinds associated with a *state machine transition*:

- *starttrans* - the instant of start of transition execution (trigger consumption)
- *endtrans* - the instant of end of transition execution (enter target state)
- *startendtrans* - the instant of the execution of an instantaneous transition
  match {**starttrans** | **endtrans** | **startendtrans**} <class>@<transitionname>
  where <class> designates the class of the transition and <transitionname> its name.

Event kinds associated with a *state machine state*:

- *enter* - the instant at which a state is entered
- *exit* - the instant at which a state is left
  match {**enter** | **exit**} <class>@<statename>
  where <class> designates the class of the state and <statename> its name

Event kinds associated with an *object*:

- *create* - the instant at which the object is created
- *delete* - the instant at which the object is deleted
  match {**create** | **delete**} <class>
  where <class> designates the class of the created or deleted object.

Event kinds associated with a *timer*:

- *occur* - the instant of timer expiration
- *timeout* - the instant of timer consumption

match {**occur** | **timeout**} <class>::<timer>

where <class> designates the class and <timer> a timer attribute of this class. Other important instants associated with timers are those associated with the instantaneous actions *set* and *reset*.

Event kinds associated with a *resource*:

- *startresource* - the instant at which an object starts to use a resource in mutual exclusion
- *endresource* - the end of such a resource usage phase

match {**startresource** | **endresource**}
of <ident> **by** <ident>

where the first identifier designates the object and the second on the resource.