A Real-time Profile for UML and how to adapt it to SDL*

Susanne Graf and Ileana Ober

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

Abstract. This paper presents work of the IST project OMEGA, where we have defined a UML profile for real-time compatible with the "*Profile for Performance, Scheduling and Real-time*" accepted recently at OMG. In contrast to this OMG profile, we put emphasis on semantics and on its use in the context of timed analysis of real-time embedded systems. The defined profile is compatible with the time concepts existing in SDL, and we show how we can adapt also those notations to SDL and MSC which do not yet exist in these ITU languages.

1 Introduction

Today's embedded applications, have often strong constraints with respect to deadlines, response time and other non-functional aspects[1]. They may be distributed and run on different execution platforms and this influences strongly the non functional characteristics. In order to allow early analysis of timing properties, we propose to lift the choices and constraints coming from a given run-time system, as well as assumptions on the environment, to the abstract model level in terms of a set of annotations which can be interpreted by timed validation tools. In order to make this approach feasible, the annotation language must be flexible and orthogonal to the functional description. It must allow the expression of global constraints and their refinement into a set of local constraints.

The aim:

Our aim is not only to provide means for describing systems with time dependent choices, but also for defining timed models of systems, including time related assumptions on the environment, e.g. on the arrival time of inputs of the environment and (platform dependent) durations of tasks. Thus, the idea is to perform validation (simulation, symbolic test or verification) on a timed *model*, built from knowledge or assumptions on the environment and the platform including aspects like resource sharing and scheduling (which have to be confirmed later by testing, run-time monitoring or code analysis). At requirement level, this

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

** VERIMAG is a research laboratory of CNRS, Université Joseph Fourier and Institut National Polytechnique de Grenoble

1 We restrict ourselves here exclusively to timing aspects

approach can be used to validate the consistency of a set of requirements and to build the loosest timed model satisfying all requirements, which than can be further refined compositionally[2].

Overview on existing approaches:

Modelling languages and tools used in the domain of embedded and real-time software, usually do have time related concepts, but in general, they are not sufficient for timed analysis at model level. The existence of analysis tools is crucial in this context, but presently only partially covered.

Languages using an *asynchronous event driven approach*, like SDL [IT00a], ROOM [SGW94], State charts [Har87] and others, use a notion of global and external time. In SDL and ROOM, global time is available within the system through a special expression, and can be stored in variables or sent as parameters of messages. The Unified Modeling Language UML [UML99] does not state its position concerning the semantics of time progress, with the aim of being flexible enough to accommodate all possible approaches.

The development tools existing for SDL, [Tel99a,Tel99b], allow code generation and some restricted form of analysis. Properties are expressed in terms of MSC [IT00b] or by means of a tool internal observer language. These tools define a particular time model and special profiles for limited performance and timing analysis based on simulation of scenarios. Most approaches for enriching SDL with time constraints and providing tools for performance or timing analysis are scenario based [Leu95,BAL97]. Also, an SDL methodology for the development of real-time systems [ADL01$^+$01] and on the extension of SDL with time constraints for timing and performance analysis [MTMC99,BGM$^+$01] have been studied.

Although standard UML does not include any particular time framework, a number of tool supported frameworks have been proposed. For example, Rose RT (a variant of the ROOM framework), the Real-Time Studio of Artisan [Art01], Rhapsody [Ilo] and, more recently, the tool TAU Generation 2 [Tel02] propose UML frameworks including real time aspects. These tools allow automatic or semi-automatic code generation and have facilities for simulation and restricted functional analysis but almost no support for timing analysis. For instance, for Rhapsody exist validation tools [BDW00], but only in the context of a rather deterministic, external time semantics and not for time dependent properties.

Recently, a UML Profile for Schedulability, Performance and Time Specifications [OMG02] has been defined. It integrates the ideas and concepts from most previously named approaches. It is very general, in order to be able to adapt to any possible real-time framework and for all kind of diagrams, and it defines essentially a vocabulary. It is very much tailored towards timed scenarios, and for the moment, it exists essentially on paper.

Timed automata [HNSY92][BST98b] have been used for modelling real-time aspects of systems, for defining semantics of modelling languages and for studying controller synthesis [MPS95,AMP95] and scheduling frameworks [BGS00] [HHK01]. A number of validation and analysis tools, such as Hytech [HHWT97],

[2] the traffic light presented in section 4 is an instance of this type of use

Kronos/IF [Yov97,BGGM00,BGM02] or Uppaal/Time [LPY97,FPY02], exist for the framework of timed automata (extended with data), but they are in general not closely coupled with development environments.

In the OMEGA IST project[3], aiming at *the definition of a development methodology for embedded and real-time systems in UML based on formal techniques*, we have started to refine the UML Profile for Schedulability, Performance and Time in order to make it usable for efficient analysis. In this article, we report on this work and show how it can be adapted for the definition of a real-time profile for SDL which is compatible with the already existing time related features of SDL and MSC.

We start, in Section 2, with an overview of the concepts necessary in a real-time framework. In Section 3.1, we define the *basic time concepts* of the UML real-time profile of the Omega project and and their semantics. They are expressive enough to express all time related elements of a model. An increase in expressiveness can be obtained by an extension with probabilistic features, as used in the context of performance analysis. In Section 3.2, we define the semantics of the basic time concepts as an extension of any untimed formalism which can be interpreted as an event labeled transition system. In particular, we distinguish two different interpretations of Boolean expressions, either as predicates or as constraints. In Section 3.3, we define a set of derived notations, intended to give the user a convenient means for the expression of common constraint patterns. They can be expressed in terms of basic concepts, at least in the context of the expression language OCL [WK98], allowing powerful quantifications. The example in Section 4 illustrates some of the concepts introduced previously. Finally, in Section 5, we propose a way to adapt this UML profile to SDL.

2 Needs for timed specifications

A minimal set of real-time related concepts necessary for a modelling language with the aim to support all the stages of the development of a real-time system, are the following ones:

A global notion of time. If necessary, local time can be defined by means of local clocks which have a well-defined relationship to global time (which might be defined in terms of drift (maximal deviation of speed), offset (max. deviation of value), ...).

Explicit *functional use of time* by means of access to time through *timers*, *clocks* and a construct like *now* (allowing access to global time), allowing to make control- or dataflow explicitly time dependent. It is desirable that the necessary discretization can be decided as late as possible in order to avoid unnecessary complexity.

Means for the specification of time constraints representing requirements or assumptions on the environment; for the underlying execution platform assumptions (or knowledge) on the execution time of activities must be expressible and

[3] see http://www-omega.imag.fr

taken into account at model level. We propose constraints on durations between *events* as the basic means for expressing time constraints, as it is the case in Sequence Charts and in Timed Automata. This is also in line with the UML RT profile.

Timing issues are strongly influenced by the *execution mode* and *the type of concurrency* between different parts of the system. Concerning parallelism one can distinguish *simulated parallelism* (concurrent entities sharing computing resources) and *distributed parallelism.* This distinction is generally not made in modelling languages emphasizing on functional aspects, but it is important when assumptions on execution times and effects of resource sharing should be taken into account.

For entities composed by *simulated parallelism*, one may define restrictions on the level of granularity at which computations of concurrent objects are interleaved: in the case of *functionally independent behaviours*, the choice of this granularity has no influence on the functional properties; at the level of the implementation, this granularity is defined by the *preemptibility* or non-preemptibility of steps of computations. When interleaving a set of *functionally dependent parallel behaviours*, the choice of the granularity influences not only the timed but also the functional behaviour. It defines when stimuli from outside can be taken into account. Extreme choices consist in:

- making no hypothesis on the granularity (e.g. in Java): any parallel behaviour of the set can accept stimuli at any time, independently of all others. This obliges the designer to handle mutual exclusion and access of shared variables explicitly,
- run-to-completion semantics: stimuli from the outside of each entity are only accepted when all the parallel behaviours of the set are stable (waiting for a stimulus) and no stimulus from within the entity is present.

In the context of the second solution, activities are insensitive to their environment before they are terminated, that is *atomic* as seen from outside. In this context, it is important to have an *interrupt* concept, allowing for example to terminate useless activities without cutting all the atomic steps into smaller pieces (and adding corresponding locks, etc.). In this article, we focus only on purely time related aspects, and we consider the execution model to be part of the functional model. In the subset of UML we consider in Omega, a notion of *activity group* (basically an active object and several passive objects, executing like an SDL process in a run-to-completion fashion) defines how activities within a group can interleave.

Scheduling. For schedulability analysis, one needs to distinguish between simulated and distributed parallelism between *activity groups*, as this determines which execution times add up and which ones don't. We introduce the following notations[4]

- a notion of *resource* and a *deployment mapping* from activity groups to resources

[4] partly described in Section 3.3, but still to be worked out in detail

– the distinction between *preemptible* and *non preemptible* resources, actions and objects for the definition of a notion of atomicity.
– the distinction 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[5] which is necessary for an analysis taking into account scheduling issues.
– *priorities* within or between concurrent behaviours, defined in a hierarchical fashion, are a powerful means to eliminate non determinism and can be used at model level to define any scheduling policy. At implementation level, priorities might either be reflected by a run-time scheduler (possibly a hierarchy of explicit event handlers) or by sequential code generated according to the defined priority rules.

3 Ingredients for a UML profile for timed verification

We start by defining a set of basic time related concepts, that give expressiveness to the framework, in Section 3.2 we define their semantics, and finally in Section 3.3, derived concepts are introduced to make the use of the framework easier.

3.1 Basic time related concepts

The only time related concept existing in the standard UML is the *Time* data type. UML defines no mechanisms for representing time progress nor operations on *Time*. *profile for schedulability, performance and time* [OMG02] defines a large vocabulary of time related concepts, which is not completely worked out and which is more or less syntax so far. We consider here only the subset concerned with real-time, add some missing concepts, and propose a semantic framework.

Primitives for modelling time. As in SDL, the time model is based on two data types: *Time*, relating to time *instances*, and *Duration*, relating to the time elapsing between two instances of time. These data types can be used, like any UML data type, in attributes, parameters, etc.

A particular instance of time is *now*. It always holds the *current time* and is visible in all parts of the model. Using the vocabulary of ASM [Gur95], *now* is a *monitored* variable, i.e. some external mechanism changes its value. A general constraint imposes its values to monotonically grow, and model dependent *time constraints* can introduce additional restrictions on time progress as compared with system progress.

Timing mechanisms. In the UML RT Profile, two related timing mechanisms are introduced, informing the system on time progress: *timer* and *clock*. A *timer* is an extension of its homonym in SDL (as it can be *set*, *reset* and sends *timeout* signals). Additionally, timers can be periodic, can be paused or restarted. *Clocks* are similar to periodic timers and emit *Tick*s.

[5] obtained by measurements on the target platform or by static timing analysis as done, for example, in [SRW02]

Events play an important role in the UML RT profile. They are defined in UML as a "specification of a type of observable occurrence", that is as an (observable) state change. There exists no explicit notion of event in SDL, but MSC define behaviours in terms of events. We use *TimedEvents* with a *time stamp* holding their occurrence time. We define a rich set of events allowing to refer to all relevant time points of a behaviour.

For instance, with any transmission of a signal *sig*, are associated three events: *send(sig)* - the instant at which the signal is sent by the sender, *receive(sig)* - the point of time at which the signal is received by the receiver (in its input queue), and *consume(sig)* - the instant at which the signal is consumed and a transition is triggered (or the signal is discarded). Not all three events associated with a signal must necessarily be distinct. For example, if *receive(sig)* and the corresponding *consume(sig)* occur at the same time, they represent a single event, and this corresponds to non buffered communication. With any *action* are associated events *start(action)* and *end(action)*. In case of an *instantaneous* action, the *start* and *end* event are always simultaneous and denoted by the event *start-end(action)*. The complete list of predefined events (associated with operation calls, signal exchange, actions, transitions, timers, etc.) is given in the Appendix.

As an event is an instant of state change, it is defined by a triple of the form *(occurrence time, current state, next state)* or equivalently by a triple of the form *(occurrence time, current state, action)*, where the *next state* is defined by the *current state* and the *action* defines the rules according to which the state after the event can be obtained. We chose the second type of representation, by encoding the relevant part of the current state and the action in an *event name*, thus allowing constraints on an event only to depend on its *occurrence time*, *current state* and the parameters of its *action*.

For instance, a *send event* associated with a call is identified by the relevant part of the state in which the call is issued, that is: the object initiating the call, the target object, the operation that is called its parameters, and possibly also the place in the control flow of the state machine in which the call takes place as well as some condition on the local or global state.

Example: The expression *cond:send(o#tr@lab:target!sig)* identifies all the events in which the signal *sig* is sent by an object identified by *o* to an object *target* as a result of the execution of an action labeled *lab* on a transition named *tr* (of the state machine associated with *o*); moreover the condition *cond* must hold on the global state when the event occurs. Events can be specified partially, for example *send(o:sig)* represents any event in which the signal *sig* is sent by an object *o*, regardless of target, action that generated it, parameters or the global state.

Note that, even if all the parameters are identified, the event may have multiple *occurrences* in a given execution. Thus, for any given execution, an *event specifications* defines a *sequence* of *event occurrences*. In order to be able to distinguish different occurrences of an events and to to talk about the history of a given event, previous occurrences of an event can be referred to by means of expressions of the form *pre(event)*, *pre(pre(ev))*,...

Time expressions evaluate to *Time* instances. A particular time expression, evaluated only in events, is *now*. In any state, the expression $time(ev)$ evaluates to the time point of the last occurrence of an event matching ev, whereas *time(pre(ev))* evaluates to the point of time of the occurrence before the last one, ... Simple arithmetic expressions, as in SDL, are other examples of time expressions.
Example: t :*Time* $\pm$ d :*Duration*, t :*Time* $\pm$ *real* $\times$ d : *Duration* .

Duration expressions evaluate to *Duration* instances. All evaluations of features of type *Duration* are duration expressions. Other duration expressions are, as in SDL, simple arithmetic expressions, such as time instance subtraction.
Examples:
t_1:*Time* - t_2:*Time* (under the condition that $t_1 \geq t_2$)
pos_ real $\times$ *d:Duration* (a scalar product)
now - *time(ev)* (time elapsed since the last occurrence of event *ev*)
time(ev) - *time(pre(ev))* (time elapsed between the very last and the previous occurrence of event *ev*)

Predicates on Time and Duration: any Boolean expression containing duration or time instances is a time dependent predicate. It can be used (just as any Boolean expression) as guard or test within an action, or as a predicate in a property to be checked. Although in principle arbitrary time dependent Boolean expression can be used, in practice only simple forms of predicates are really useful. Indeed, the type of predicates used in properties or guards determine the possibility of analysis (see Section 5).
Examples:
t_1 :*Time* - t_2 :*Time* $\leq$ *d:Duration* (simple duration constraint)
$time(ev_1)$ - $time(ev_2) \leq d$ (simple duration constraint)
now - *time(event)* $\leq$ *time(pre(event))* - *time(pre(pre(event)))* (constraint on difference of durations, as used in auto adaptive algorithms).
Time dependent predicates are evaluated in events, and the fact that an event satisfies some predicate amounts to the evaluation of an assertion of the form $(s,t,a) \models p$[6] which is equivalent to the assertion $(s,a) \models p[t/now]$ or $s \models p[t/now]$ if p is independent of the parameters of the action. The interpretation of time dependent predicates can be extended to states:

$$s \models p \text{ iff } \forall t \in [time(enter(s)), time(exit(s))] \,.\, (s,t) \models p$$

that is if p holds in all time points in which the system stays in state s.

We distinguish between *predicates*, which are evaluated to *true* or *false* in individual events (or states), and *invariants* - of the form *invariant(p)* (where p is a predicate) - which hold if p has been *true* ever since the initial state. Interesting invariants are often of the form $invariant(in(s) \Rightarrow p)$ or $invariant(at(e) \Rightarrow p)$, requiring p to hold at each occurrence of event e, respectively in all time points in which the system is in state s. In order to ease the expression of such invariants, we allow to "attach" *invariant(p)* with states s or events e as a short hand.

[6] where s is a *state*,t a *time* value, and a represents the parameters of the *action* represented by the event

3.2 Semantics

The semantics of time related concepts is defined independently of the semantics of the formalism used for expressing functional behavior. We only suppose that the semantics of the functional behaviour of systems can be viewed as a labelled transition system, where transitions represent *events*. An additional requirement is that all the events referred to, are identifiable in the transition system defining the functional semantics.

We define the semantics of the time related concepts by a set of timed automata with urgency [BST98b][7] constraining the *occurrence time* (and only on the occurrence time) of all the events whose possible occurrence ordering are given by the untimed semantics.

Thus, the *timed behaviour* of a system can be represented by the synchronized product of the possibly infinite event labeled transition system defining its behaviours in terms of state changes and (partial) order of events, and a set of time automata defining allowed occurrence times of events.

How time related concepts can be represented as timed automata is relatively straightforward for most constructs. We present here the timed automaton associated with a timer to demonstrate that this way of defining time extension has the expected effect, and we discuss the interpretation of time dependent Boolean expressions, as for them several interpretations coexist.

The time automaton associated with any timer instance is given in Figure 1a. It does not express any constraint on the occurrence times of the actions *set* and *reset*, but we suppose that they occur immediately after the preceding action (in the same sequential behaviour), which in the timed automaton is expressed by an *eager* transition. Also, given an occurrence of *set*, this automaton constrains the occurrence time of the *timeout* (i.e. the *occur* event) to exactly the defined timeout time *time(set(timer,delay)) + delay* which is also expressed by an *eager* transition.

The point of time of the consumption of the timeout, however, is not restricted by the timer itself, but possibly by some additional constraint, for example *"always the time between the timeout and the corresponding consumption is smaller than d"*[8], which can be expressed by an invariant of the form

$$\textit{invariant}(\ \textit{at}(\textit{consume}(t)) \Rightarrow \textit{now} - \textit{time}(\textit{occur}(t)) \leq \textit{consume-delay}\ )$$

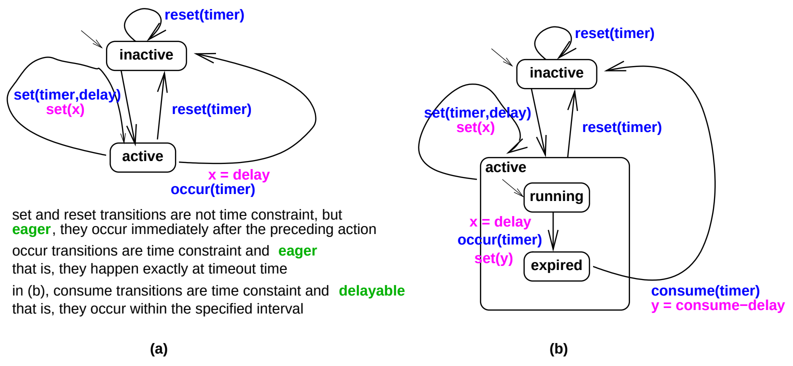
or, as we will see in the next section, by the shorthand

$$\textit{invariant}(\ \textit{duration(occur(t), consume(t))} \leq \textit{consume-delay}\ )$$

Figure 1b shows the timed automaton corresponding to the composition of the timer timed automaton and the timed automata associated with this constraint, where the transition associated with the consume event is *delayable*, meaning that it will occur somewhere within the specified delay.

[7] Alternatively, we could have another formalism, such as ASM, but the advantage of timed automata is that they have as predefined concepts, all the primitives making the expression of the semantics easy

[8] under the condition on the functional model that the timer is only set again after the timeout has been consumed



set and reset transitions are not time constraint, but **eager**, they occur immediately after the preceding action

occur transitions are time constraint and **eager** that is, they happen exactly at timeout time

in (b), consume transitions are time constaint and **delayable** that is, they occur within the specified interval

Fig. 1. timed automaton for a timer (a) with constraint consumption delay (b)

Interpretations of time dependent Boolean expressions

Boolean expressions can be interpreted in different ways, depending on their use:

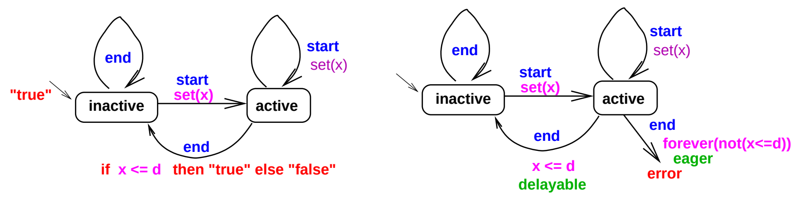
Predicates: Time dependent Boolean expressions used as guards of transition triggers or in decisions are predicates, which take at each occurrence of the transition either a value *true* or *false* depending on the value of time at the instance the *trigger event* occurs. The value of complex predicates can be combined from basic ones and can be used in other event based property formalisms: for example in sequence charts, predicates are used as *conditions*.

A timed automaton representing a time dependent predicate, is an observer which, depending on the point of time at which the concerned events are observed, provides a Boolean value. In such a timed automata all transitions are *lazy*, meaning that no constraint on possible time progress is expressed, but the Boolean value (the value of the predicate) depends on time progress. As an example, see the timed automaton representing the predicate $duration(start, end)$ in Figure 2.

Invariants: The value of a (time dependent) invariant of the form $invariant(p)$ is *true* only if p holds on the entire execution leading to the current state or event. Invariants can be used as *properties*, meaning that they should be derivable from a given system description. Live Sequence Charts [DH99,HM02], as they are used in Omega, provide an intuitive means to express complex invariants as well as liveness properties. The timed automaton associated with $invariant(p)$ is that of p, except that, once it takes the value *false*, its value remains *false* forever.

Constraints: invariants can also be used as *constraints* or *(assumed) facts* on the system under consideration, its environment or the underlying execution platform. An assignment $x := 0$ represents the invariant *whenever this step is executed, in the state just after the execution the variable* x *has the value* 0 without necessarily saying how this is achieved exactly. Analogously, an invariant of the form $duration(action) \leq 5$[9] can be used to restrict the model to the set

[9] see next section for the definition



a) automaton representing a predicate on duration(start,end) b) automaton representing a constraint on duration(start,end)

Fig. 2. timed automata representing a predicate(a) or an invariance constraint (b)

of executions satisfying the functional constraints in which not more than 5 time units pass between the event $start(action)$ and the corresponding event $end(action)$.

Such a constraint on the time elapsing between two events, *forces* time progress or waiting in accordance with the constraint. In timed automata, this is obtained by means of *eager* or *delayable* transitions. Whenever, it is impossible to force time progress in such a way as to satisfy some constraint, this means that the set of time constraints is either inconsistent or incomplete and can be represented in the timed automaton implicitly by a timelock or explicitly by an error condition (see Figure 2b for an example).

Constraints restrict the model on which *properties* are verified. If constraints are *non overlapping*, that is, in every possible execution, any event is under the scope of at most one time constraint, they are consistent. In this case, the set of associated timed automata define indeed an executable model, in the sense that any computation satisfying the constraints up to some point of time, can be continued by satisfying all constraints. When constraints are overlapping, they might be inconsistent or incomplete. Inconsistency must lead to some redesign. Figure 3 shows two incomplete sets of (overlapping) constraints. Both sets of constraints are satisfiable, but must be completed by a constraint on the occurrence time of the event e_3 (relative to the occurrence time of e_1) in order to guarantee that all other constraints remain satisfiable. Most existing tools, such as the simulation tools for SDL or the tool for executing timed live sequence charts of [HM02], avoid this problem by a priori choosing the earliest point of time at which an event is possible. This corresponds often to a possible execution (at least when the set of constraints is consistent), but not always, as shows the second example of Figure 3. Moreover, our aim is to keep constraints as loose as possible, and thus to characterize the largest set of timed behaviours of (sequential) subsystems and to enable compositional analysis.

In [BS00] is described a framework for more flexible composition of duration constraints than conjunction which could be used to further extend the real-time profile presented here.

3.3 Derived duration expressions

Our approach is based on duration constraints between the occurrences of an event *ev1* and a subsequent event *ev2*. In a sequence of occurrences of *ev1* and *ev2*,

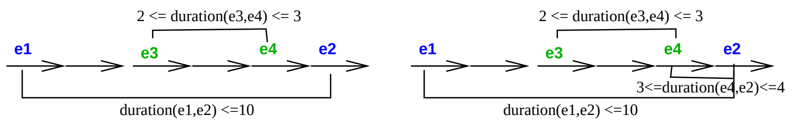


Fig. 3. Incomplete Time constraints

several options exist to identify the "matching" pairs between which to impose the constraint. We propose expressions for three different ways of identifying such matching pairs.

1. *duration*(*ev1, ev2*) represents the duration between an occurrence of *ev1* and the next occurrence of *ev2* such that there is no other occurrence of *ev1* in between. If evaluating *duration*(*ev1, ev2*) only at the occurrence of the first *ev2* after a (series of) *ev1*, the value of *duration(ev1, ev2)* is that of *now - time(ev1)* that is the time passed since the **last** event *ev1*. At other points of time, its value is difficult to express only in terms of occurrence times of events: $time(pre^n(ev2))$ - $time(ev1)$ where n is chosen such that $time(pre^{n-1}(ev2)) \leq time(ev1) \leq time(pre^n(ev2))$. Using additional time stamp variables or a timed automaton makes the expression of this constraint easy (see Figure 2 for the expression of $duration(start,end) \leq d$).
   A particular instance is *duration(ev1,now)* which represents at any point of time the duration since the last occurrence of *ev1*, that is *now - time(ev1)*. This expression corresponds thus to an implicit duration counter associated with event *ev1*, and it can alternatively be expressed by means of a time stamp at each occurrence of *ev1*.
2. When several request events (each corresponding to *ev1*), may all be answered by a single *effect* event (corresponding to *ev2*), and this should happen within a limited time starting from the first request, the time elapsed between the **first** event of a series of occurrences of *ev1* and the first consecutive *ev2* is relevant. The expression *duration-first*(*ev1, ev2*) can be used in this case (the timed automaton associated with $duration\text{-}first(start,end) \leq d$ is obtained by eliminating in the timed automaton of Figure 2 the *set(x)* action in case of repeated *start* events).
3. Finally, the case where several observations on pairs *(ev1,ev2)* are "active" simultaneously, must be considered. In most cases, one can find a parameter x (or combination of parameters) identifying the matching pairs, and express the required constraint by a constraint of the form

   $$for\ any\ x{:}\ duration(ev1(*,x,*),ev2(*,x,*)) \leq d.$$

   Nevertheless, due to the implicit use of FIFO buffers for storing signals in objects, it might be impossible to always find the matching parameters, and for this purpose *pipelineDuration*(*ev1, ev2*) is introduced, where the matching pairs may be overlapping, observed in a pipelined fashion. In any case, such a set of constraints cannot be be expressed by a single timed automaton, but by multiple instances of a parameterized timed automaton, such that at every occurrence of *ev1*, a new instance (with appropriate values of parameters) is instantiated, constraining only its corresponding *ev2* event; each instance is killed when the constrained event *ev2* occurs.

Moreover, we propose some convenient shorthands

1. Instead of *duration(enter(state),exit(state))* or *duration(trued(cond),falsed(cond))*, the shorthands *duration(in(state))*, respectively *duration(in(cond))* can be used, and analogously for any type of *"duration"*.
2. We allow also expressions of the form *duration(SE1,SE2)*, where *SE1, SE2* are lists of events[10], representing the distance between any occurrence of an event in *SE* and the next instance of any event in *SE2*.

Additionally, duration expressions are defined which correspond to frequently used duration patterns. We define their semantics by a mapping to basic duration expressions:

Client response time attached with operation calls and signals with some specified response, is defined as *duration(invoke(op),consume-return(op))*, that is, the time elapsed between the moment at which the operation call (respectively the initiating signal) is emitted and the moment at which the response (return or defined response signal) is consumed by the client.

Example: ClientresponseTime(sourceObj:sig1, sig2)≤ 7, means that each time the object *o* sends *sig1*, then, **if** it receives a response *sig2* some time later, this will happen within less than 7 time units.

Similarly, a *Server response time* attached with operation calls or signals with specified response, is defined by *duration(receive(op),invoke-return(op))*, that is, the time between the reception of a request (in the input queue) and the moment at which the response (or return) is sent back (from the server point of view).

A *Duration* can be associated with some behaviour (action, transition, operation, or signal with specified response), as a shorthand for the duration between the start and the termination time of the behaviour. That is, one can simply write *duration(act)* for *duration(start(act),end(act))*. It does not include waiting time in the event queue when the execution refers the treatment of a signal, but it does include potential preemption time during the execution of the behaviour. *Execution time* associated with a behaviour, is similar to its *duration*, except that it also takes deployment issues into account and accounts only for the time during which the object is executing and neither preempted nor waiting for some response from external objects.

A *Period*, attached with an event, is defined as the duration between successive occurrences of the event, that is *duration(pre(event),event)*.

Reactivity is attached with objects or groups of objects and is defined as the maximum delay between any event of the form *Receive(req)* and the corresponding event *Consume(Req)*, that is the maximal delay which may elapse between the moment at which a request reaches the object and the moment at which it starts to treat it. This feature is useful, when the size of input queue can be statically bounded.

A *Transmission delay* attached with associations stereotyped as communication paths (similar to SDL channels), defines a communication delay.

[10] where a list is used to represent a set of events

The list of derived duration expressions has the status of an initial proposal. It contains a number of concepts likely to be interesting and serves to illustrate the general idea. Nevertheless, a larger discussion and more feedback from users is necessary. Some of the concepts may need adaptation; for instance, the more restricted notion of *Worst case execution time* might be more useful than *Execution time*, it might be interesting to distinguish different kinds of transmission delays, and predefined notions of *jitter* might be introduced explicitly, instead of asking the user to write down his preferred formula relating the occurrence times of n consecutive events *ev, pre(ev), ..., pre*^{n}*(ev)*.

4 Example

We illustrate a possible use of time annotations by modeling a part of the traffic light system of the SDL design contest. Given the level of description of this example, execution times, response times, etc play no role. At control level, only expressions of type *duration()* and the distinction of different types of invariants are relevant. The main idea is to illustrate the possibility of complete separation of functional and time dependent specifications and of the expression of time constraints by means of a set of relative simple constraints, each one depending on not more than a few events.

A traffic light controller consists of a set of traffic light controller proxies (TLCP)[11]. Each TLCP controls the light switches of all its associated traffic lights and gets information about presence or absence of traffic on its controlled lanes from a number of sensors (at different distances from the lights on all the lanes) by means of signals *traffic* and *notraffic* from a *traffic sensor handler* associated with each TLCP, not described here. The information about the status is stored in a Boolean variable *traffic*. The state machine of TLCP described using SDL (see Figure 4) has only two states, *red* and *green*, and it can pass from *green* to *red* without any condition, and from *red* to *green* when traffic has been detected and all other traffic lights are *red* (this requires a global variable *AllRed* updated by all the TCLPs at every transition). Each traffic light has an additional *orange* phase of some fixed length, but this is handled in the traffic lights themselves (not described here).

The state machine of Figure 4 describing the functional behaviour is totally time independent. In addition, a number of time constraints need to be satisfied. We show that also for this type of "functional" time constraints, it can be advantageous to use a constraint based approach, instead of starting immediately with an *implementation* in terms of timers or some cyclic environment observation.

The first constraint prevents livelocks: whenever there is traffic, the light will be turned *green* within some delay max_r. This is expressed by an invariant of the following form, where we omit the context *TLCP(i)* to simplify the expressions:

[11] In the contest only two TLCP corresponding to two "directions" (that is sets of coupled traffic lights) are considered, but in a more general setting a larger number of directions may exist (the approach could also be extended to dynamically formed sets of lights with common green phases).

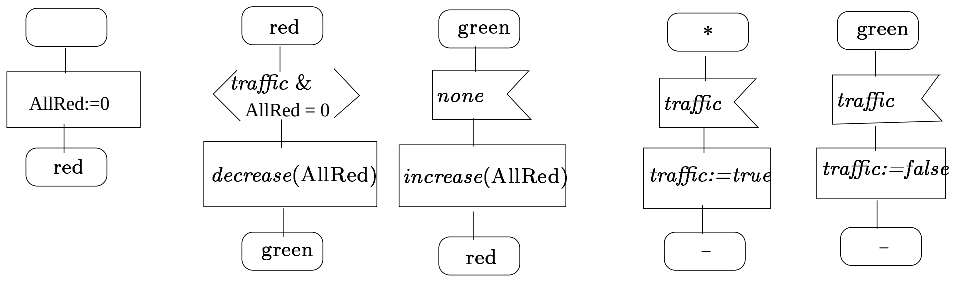


Fig. 4. TLCP functional behaviour

$$\textit{duration(\{enter(red),trued(traffic)\}, enter(red))} \leq max_r \quad (1)$$

Due to the existence of an *orange* phase, and in order to avoid unstable behaviour, a minimal duration of both the *green* and the *red* phase must be fixed:

$$\textit{duration(in(green))} \geq min_g \ (2)$$
$$\textit{duration(in(red))} \geq min_r \ (3)$$

Constraint (1) implies a constraint on the "nominal" length of the *green* phase, depending on the number N of "directions". Moreover, once the minimal delay passed, as soon as there is no traffic anymore on the lanes of *TLCP(i)*, the *green* phase should be terminated within a delay ϵ (except if within ϵ new traffic arrives). When there is traffic all the time (with the exception of some durations not longer than ϵ), the *green* phase takes its nominal length $max_g \leq max_r/(N-1)$. We express this as an invariant[12] attached with the *green-to-red* transition:

$$\begin{array}{l}
(\textit{duration(in(green))} = min_g \ \Rightarrow \ \textit{duration(trued(notraffic),now)} \geq \epsilon) \\
\wedge \ (min_g \leq \textit{duration(in(green))} \leq max_g \ \Rightarrow \ \textit{duration(trued(notraffic),now)} = \epsilon) \\
\wedge \ (\textit{duration(in(green))} \geq max_g \ \Rightarrow \ \textit{duration(trued(notraffic),now)} \leq \epsilon \ ) \\
\wedge \ (\textit{duration(in(green))} \leq max_g) \quad (4)
\end{array}$$

The composition of the timed automata associated with all constraints[13] is given in Figure 5. The so defined traffic light never turns *green* without traffic and stays *green* a limited amount of time. In order to make sure that even with permanent traffic on all directions each light turns *green* after at most max_g time, the order in which the lights turn *green* cannot be chosen non deterministically[14]. Instead of choosing some fixed order, we propose a more flexible solution based on a dynamic *priority*, initialized arbitrarily to

$$\textit{init(priority) = forall j (priority(j):=j)} \quad (5)$$

and updated each time $TLCP(i)$ changes to *green* by

$$\textit{update(i,priority) = forall j (priority(j):= (if j=i then 0 else priority(j)+1))} \ (6)$$

[12] using the "min-synchronization" of the framework of [BS00] instead of conjunction with constraint (2), would allow to simplify a lot the expression of constraint (4)

[13] taking into account that the green and the red phase alternate, and simplified by the assumption that traffic cannot go away in the *red* phase

[14] in order to allow a non deterministic choice, the condition for *turn green* must also depend on the waiting time of all the TLCP(i)

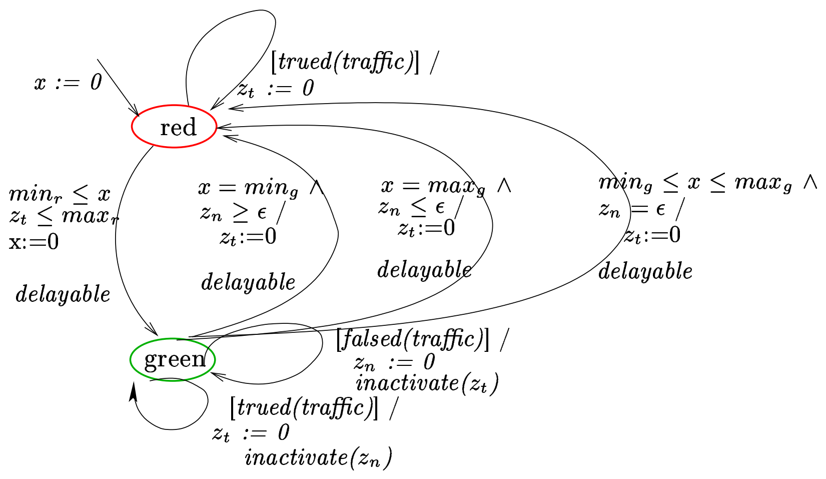


Fig. 5. TLCP behaviour: time constraints

that is, the longer a light remains *red*, even in absence of traffic on its lanes, the higher gets its priority, allowing occasional traffic to pass quickly without disturbing lanes with heavy traffic.

Figure 6 shows a possible run for 3 directions, depending on traffic. If permanent traffic is present on all directions, each traffic light spends the maximum time in *red* and in *green* (the first period in Figure 6). If there is no traffic on direction 2 before the end of the nominal green phase, then its light turns *red* earlier. If no traffic is present on some direction (here 2), even if it has the highest priority, it passes its turn keeping its priority.

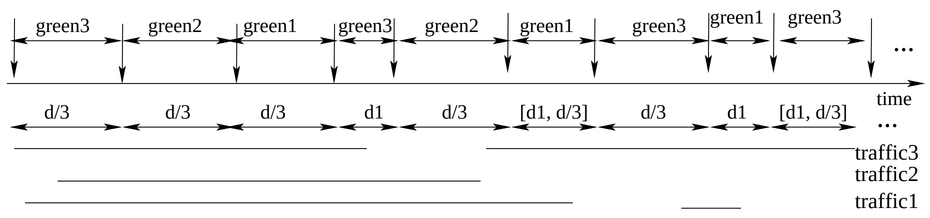


Fig. 6. A possible evolution of traffic lights over time

The timed traffic light behaviour is defined by the conjunction of the behaviours defined by Fig. 4 and Fig. 5. It can not be directly translated into SDL by replacing *clocks* by time stamps, for two reasons:

- the use of *urgency* is essential to obtain the expected behaviour
- SDL transitions represent actions or activities which might take time, whereas timed automata transitions represent *events* that is instantaneous state changes.

An actual traffic light satisfying all the constraints can finally be implemented in different ways. One consists in introducing signals *trued/falsed(traffic)* and timers for the ϵ waiting period. A second solution consists in a process scanning every ϵ for changes of the *traffic* variable, where the length ϵ of the cycle is induced by the required minimal reaction time. Notice that the cycle length is not necessarily constant: during the *red phase*, it depends on the precision with which one wants to determine the time of traffic occurrence, and, as long as the minimal waiting time (min_g, min_r) has not been reached, there is no need to scrutinize for changes of *traffic* at all.

Choosing $max_g = max_r/(N-1)$ allows to respect constraint (1). Nevertheless, in case of high traffic on all lanes, no flexibility is possible: the only solution satisfying all the constraints consists in turning the lights *green* in a round robin fashion. A smaller value of max_g, would allow to modify the priority rules and to provide additional green phases to those lanes with the highest traffic.

Another flexibility could be obtained by not a priori fixing the maximal length of the *green* phase: there is no need to quit the *green* phase if there is no traffic on any other line. This can be done by relaxing constraint (4) when there is no traffic on any other line.

Notice also that the proposed solution making use of a global variable *priority* would be made easy by the introduction of a flexible notion of global priority in SDL.

5 Discussion: Adapt this profile to SDL

SDL already contains most primitives for dealing with time. Nevertheless, SDL suffers from some deficiencies which are discussed in this section.

SDL does not offer the possibility to define local time, timers are less powerful as they cannot be periodic and they cannot express clocks which "tick" periodically, where ticks are consumed instantaneously or lost. The extension of SDL with more powerful timers and UML clocks has already been proposed previously [BGM+01,MSDH01]. Also the introduction of local time is straightforward by having a different *now* attached with each "locality". The values of different local times need not to evolve strictly synchronously, but by respecting some specified drift, offset,... Nevertheless, explicit modelling of local clocks should be avoided whenever possible, and the uncertainty on the relative speed or values of local clocks expressed in terms of uncertainties on the occurrence times or durations.

In SDL, it is left open how concurrent activities are sequentialized (when they do not run in a distributed fashion). It is however specified that activities within a process agent are executed in a run-to-completion fashion. This makes the definition of "interrupts" in principle impossible. It has been proposed (e.g. in [BGM+01]) to use the exception mechanism of SDL, which can be considered as a kind of exception to the run-to-completion principle, for the definition of interrupts.

Our main concern is the expression of time constraints on execution times and durations between events. The SDL semantics says that time passes in actions and not in states, but no means is provided to specify constraints on the time passed in actions. Waiting can be forced by means of timers and time guards. Existing tools implement a different interpretation: Like in MSC or timed automata, "atomic" parts of transitions are viewed as *instantaneous events*. Moreover, time passes only where explicitly required by waiting conditions. This view allows the expression of time *constraints* by means of time guards, decisions and explicit *error states*. Nevertheless, such error states do not exist in SDL, and they make the models cumbersome. Moreover, some of the identified events, such as "reception of a signal in the input queue" has no syntactic representation and can therefore not be constraint.

Alternatives for the expression of time constraints exist, which can be extended in such a way as to serve our purpose.

MSC: are an event based formalism allowing the expressing of constraints on time elapsing between events. By introducing also instances of agents which are implicit in SDL, such as the input queue of a process or the agent associated with a process set, all introduced events can be represented. MSC are indeed used in practice to express time constraints of SDL specifications. But MSC are not expressive enough for this purpose. For example, they do not allow the expression of constraints of the form "**if** the entire scenario in between two constraint events occurs, **then** the time constraint *must* hold, otherwise no constraint is imposed". Live Sequence Charts [DH99] are more powerful as they allow to express such implications by means of *cold* event occurrences and *hot* time constraints.
An extension of MSC, with the concepts of LSC would be very useful and allow the expression of timed scenarios in practice. Notice however, that scenario based formalisms can be very cumbersome when the number of possible alternative scenarios is high or when only events at the interface of a single agent are constraint, and they should not be the only means for expressing time constraints.

Annotations: Some tools allow the expression of duration constraints on actions in the form of special comments. The inconvenience is that each tool defines its own notation and semantics.
For simple constraints associated with processes tasks, signals,... specifying execution times of tasks or durations like response times, process execution periods,...as they occur for example in contracts specifying interfaces, annotations, using the event names and duration expressions defined in Section 3.3, are a convenient means. These annotations can take the form of special comments, but their syntax and semantics shouldn't be left open as a tool issue.

Observers: Some SDL tools provide the facility to define *observers* as a means for defining constraints and properties. Observers allow to impose constraints on occurrences of events like MSC, but they can also observe and constrain the global state of the observed system, and look almost like SDL processes. Observers are a very powerful means for expressing properties and constraints, but they exist only in particular tools, SDL does not provide any standard notation for them.

Triggers of observers are any of the defined events, and they define constraints on their occurrence time may using either time stamps (e.g. $now - t \leq 3$) or the duration expressions defined in Section 3.3 (e.g. *duration(ev,now)* ≤ 3). In order to define properties and constraints, they may use the above mentioned error states or an explicit urgency like in timed automata.

In order modelling languages to be used in practice, it is very important that the user has the freedom to express time constraints in the most convenient manner, depending on each situation. For this reason all three above mentioned approaches are useful. The interest to separate functional specification and timing information, can be motivated by the fact that timing information is mostly platform or implementation dependent. Separation makes it easier to adapt the time related specifications to a different platform. In all three types of formalisms, we distinguish between *constraints* which are part of the definition of the system, and *properties* which must be derivable from the system definition and constraints.

In order to take into account scheduling and deployment related information, new notations have to be introduced in SDL. The simplest way is almost the one proposed in QSDL [MTMC99]. At the architecture level, a list of *resources* with an attribute defining their preemptibility is defined. For block and process agents as well as tasks, it is specified on which resource they are executed, where inner agents and tasks run on the same resource as long not otherwise stated. Scheduling policies are defined by keywords, such as RMA or EDF, or by priority rules. They are attached with resources or with agents, in an hierarchical fashion. Within a process, priorities are defined between transitions or their triggers, and within a block, they are defined between sub-agents. Priorities may be dynamic, where dynamic priorities can be specified in a declarative way, that is depending on some precondition, or by means of an *observer* (attached with the concerned agent), which explicitly updates priorities depending on the observed states and events.

Perspectives of simulation and validation

In section 3.2 we have already discussed the the simulation issue: any SDL model with time constraints can be simulated. This is an important property, as this means that such a model can be used for model based testing.

Obviously, any interesting property is in general undecidable on an SDL specification due to infiniteness of data domains, and unbounded message buffers and agent creation. Nevertheless, the verification of time related properties can often been done on a finite control abstraction, that is a system with finite data domains, bounded message buffers and bounded agent creation. A number of interesting verification problems are decidable on such a finite control abstraction under the condition that in the timed automata, obtained by translation, clocks can only be reset to zero, stopped and restarted, and the only allowed tests are comparisons of clock values or differences of clock values with **constants**. This means that the only allowed constraints imposed on events are boolean combinations of comparisons of the type "duration since the occurrence of some event lays within an interval" or "the difference of occurrence times between two past events lays within an interval".

MSC which impose only interval constraints on occurrence times of events or on durations between pairs of events and timers, are in this restricted set.

Observers satisfy this constraint if they use only constraints which are Boolean combinations of interval constraints on expressions of the form *duration*(e_1,e_2)[15]. An alternative, is using time stamps of the form "$t := now$" and comparisons of the form "$now - t \in [2,3]$" or "$t_2 - t_1 \in [2,3]$". The general time stamping mechanism - which allow the *comparison* of time distances with different end points (e.g. "$(now - t_2) \leq (t_2 - t_1)$"), as they are used in auto-adaptive algorithms - is more expressive and outside the decidable set.

Schedulers make use of *integrators*, that is, they imply clocks used for counting execution time which are stopped when the process is suspended and restarted when it becomes executing again. As long as only interval constraints on such integrators are tested, which is typically the case when worst-case execution times and deadlines are specified, decidability is preserved.

There exist several tools based on timed automata allowing some verification under the above mentioned restrictions, such as Hytech [HHWT97], the Kronos/IF tool [Yov97,BGGM00] and Uppaal [LPY97].

Let us consider a number of relevant time related verification problems:

- Consistency checking of a time constraint system and verification of properties expressing requirements of the same kind, that is interval constraints on distances between events are very similar in nature, and can be done with the same tools.
  Notice that, when execution time constraints and deadlines are specified, and several concurrent behaviours are executed on the same resource, consistency checking and verification includes schedulability checking.
- Incomplete specifications, as in Figure 3, are problematic, as they may require backtracking during simulation. Under the before mentioned restrictions, the verification algorithm does synthesize for any transition the weakest constraint guaranteeing the satisfaction of all future constraints. When constraints are not "cyclically overlapping", completing an incomplete sequential specification can be done in linear time, even when interval bounds have parameters. This is useful for simplifying simulation, even in the case where the overall system is infinite.
  Nevertheless, constraint propagation can be done automatically with a reasonable effort only within a sequential behaviour (defined by an agent or a small set of agents. In general, budgeting over concurrent agents, must be provided by the user.

Notice that in the case where the intervals defining constraints are closed, there always exists an exact discretization. Thus which time model is used is not important. In practise, dense time leads often to more tractable models.

[15] or *first-duration*(e_1,e_2) or any of the predefined duration expression. Notice that *pipeline-duration* poses a problem when no bound on the maximal number of concurrently active constraints can be given a priori

What to do when the system does not satisfy the requirements making the verification problem decidable? As long as all constraints are expressible by linear inequalities on time points or durations, the above mentioned verification problems are "semi-decidable", that is, successor sets can still be computed and termination detected, but there is no guarantee that the underlying model is finite and the verification procedure terminates. For these systems, one can still hope that the actually generated model is finite and small enough to be verified. Other remedies consist in using approximations of fixpoints or approximations based on reformulations of the constraints so that they can be analyzed by the above mentioned tools which use a more restricted internal representation than general linear inequalities.

References

[ADL01+01] J.M. Alvarez, M. Diaz, L.M. Llopis, E. Pimentel, and J.M. Troya. Deriving hard-real time embedded systems implementations directly from SDL specifications. In *Int. Symposium on Hardware/Software Codesign CODES*, 2001.

[AGS00] K. Altisen, G. Gössler, and J. Sifakis. A methodology for the construction of scheduled systems. In *proc. FTRTFT 2000*, LNCS 1926, 2000.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *proc. Hybrid Systems II, LNCS 999*, 1995.

[Art01] *ARTiSAN*, 2001.

[BAL97] H. Ben-Abdalla and S. Leue. Expressing and analysing timing constraints in message sequence chart specifications. Tech. report, U. Waterloo, 1997.

[BB93] F. Bause and P. Buchholz. Qualitative and quantitative analysis of timed SDL specifications. In *Proc. KiVS'93*, Informatik aktuell, 1993.

[BDW00] T. Bienmüller, W. Damm, and H. Wittke. The Statemate Verification Environment – Making it real. In *Conf. Computer Aided Verification, CAV*, LNCS 1855, 2000.

[BGGM00] M. Bozga, L. Ghirvu, S. Graf, and L. Mounier. IF: A validation environment for timed asynchronous systems. In *Comp. Aided Verification, CAV*, LNCS 1855, 2000.

[BGM02] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for component-based real-time systems. In *CAV*, LNCS, 2002.

[BGS00] S. Bornot, G. Gössler, and J. Sifakis. On the construction of live timed systems. In *Proc. TACAS 2000*, *LNCS* 1785, 2000.

[BGM+01] M. Bozga, S. Graf, L. Mounier, Iulian Ober, and D. Vincent. Timed extensions for SDL. In *SDL Forum 2001*, *LNCS* 2078, 2001.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.

[BST98b] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. *Int. Symp. Compositionality - The Significant Difference*, LNCS 1536, 1998.

[DH94] W. Damm and J. Helbig. Combining visual formalisms: A compositional proof system for the correctness of statechart implementation against timing diagrams specifications. *IFIP Conf. PROCOMET'94*, Elsevier, 1994.

[DH99] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *FMOODS'99 IFIP TC6/WG6.1 Conference on Formal Methods for Open Object-Based Distributed Systems.* 1999.

[DHHMC95] M. Diefenbruch, E. Heck, J. Hintelmann, B. Müller-Clostermann. Performance evaluation of SDL systems adjunct by queuing models. *SDL-Forum*, 1995.

[FPY02] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: schedulability and decidability. In *TACAS*, LNCS 2280, 2002.

[GB99] I. Jacobson G. Booch, J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford Univ. Press, 1995.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HHK01] T. Henzinger, B. Horowitz, Ch. Kirsch. Giotto: A time-triggered language for embedded programming. *Embedded Software*, LNCS 2211, 2001.

[HHWT97] T. Henzinger, Pei-Hsin Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *J. on Software Tools for Techn. Transfer*, 1(1-2), 1997.

[HM02] D. Harel and R. Marelly. Playing with time: on the specification and execution of time-enriched LSC. In *Proc. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, 2002.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Symp. on Logic in Computer Science*, 1992.

[Ilo] Ilogix. Rhapsody development environment.

[IT00a] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Z-100, Int. Telecom. Union – Standard. Sect., 2000.

[IT00b] ITU-T. Recommendation Z.120. Message Sequence Charts. Z-120, Int. Telecommunication Union – Standard. Sect, 2000.

[Leu95] S. Leue. Specifying real-time requirements for SDL. *PSTV*, 1995.

[LPY97] K. Larsen, P. Petterson, and W. Yi. Uppaal: Status & Developments. In *CAV'97, LNCS 1254*, 1997.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS'95, LNCS 900*, 1995.

[MSDH01] R. Münzenberger, F. Slomka, M. Dörfel, R. Hofmann. General approach for the specification of real-time systems with SDL. *SDL forum*, LNCS 2078, 2001.

[MTMC99] A. Mitschele-Thiehl and B. Müller-Clostermann. Performance engineering of SDL/MSC systems. *Computer Networks 31(17)*, 1999.

[OMG02] OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[SRW02] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS 24(3): 217-298*, 2002.

[Tel99a] Telelogic. Objectgeode 4-1 reference manual, 1999.

[Tel99b] Telelogic. *TAU Reference Manual*, 1999.

[Tel02] Telelogic. *TAU Generation 2 Reference Manual*, 2002.

[UML99] *OMG Unified Modeling Language Specification,* Version 1.3, June 1999.
[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.
[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1-2), 1997.
[YRSW03] E. Yahav, T. Reps, M Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP 2003*, LNCS, 2003.

6 Appendix : Predefined events

The set of predefined events which can be used in time-constraints are the following ones. Moreover, the user can define additional events by means of "named" instantaneous skip actions (corresponding to SDL informal tasks)

The events associated with an *operation call* are:
1. *Invoke*: emission of the call request;
2. *Receive*: reception of the call by the provider
3. *Accept*: start of the actual processing of the call by the provider;
4. *Invokereturn*: emission of the return reply;
5. *Receivereturn*: reception of the return by the caller;
6. *Acceptreturn*: consumption of the return[16].

Events associated with a *signal exchange* are:
1. *Send*: sending a signal;
2. *Receive*: reception of the signal by the target (i.e. when the signal is added in the queue);
3. *Consume*: start of treatment of the signal (triggering of transition or the moment the signal is discarded).

The events associated with an *action* are:
1. *Start*: starting time of the action;
2. *End*; termination time of the action;
3. *Startend*: simultaneous start and an termination of an *instantaneous* action..

The events associated with a *transition* of a state machine are:
1. *Start*: starting time of a transition;
2. *End*: termination of a transition (and entering the next state);

The events associated with a *state* of a state machine are:
1. *Entry*: time at which a state is entered;
2. *Exit*: time at which a state is exited;

The *change events* associated with *Boolean conditions* are:
1. *Trued* time at which the condition becomes true;
2. *Falsed* time at which the condition becomes false.

The events associated with *timers and clocks* are (some of can be seen as synonyms for some other kinds of events described above):
1. *set/reset/start/stop* (a timer or clock): events corresponding to the instantaneous actions defined for timers and clocks;
2. *Occur* : reaching of timeout time and notification of timeout;
3. *Tick*: equivalent to occur, but related to clocks;
4. *Consume*: timeout or tick consumption.

[16] in blocking call semantics *Acceptreturn* it is the same as *Receivereturn* or *Invokereturn*