# Real-Time Statechart Semantics

Holger Giese[1] and Sven Burmester[2]
Software Engineering Group
University of Paderborn
Warburger Str. 100, D-33098 Paderborn
Germany
[hg,burmi]@uni-paderborn.de

2nd June 2003

**Abstract** *Modelling complex dynamic real-time behavior is a crucial prerequisite when employing object-oriented modelling techniques successfully in the field of complex real-time systems. In this report we therefore review the state of the art of two related lines of research, namely extensions for* expressiveness *such as Statecharts as proposed in the field of software engineering to support the description of complex behavior and extensions to specify the explicit* temporal behavior *for state-based behavior such as Timed Automata. We review the Hierarchical Timed Automata model which provides already the most fundamental concepts for structure and time as well as a reasonable semantics for the domain of temporal behavior. To enable the modeling of complex control behavior which incorporates most of the fundamental extensions for expressiveness and time in a manner appropriate for real-time systems, we then propose an extension in form of Extended Hierarchical Timed Automata and Real-Time Statecharts. Different restricted forms reflecting the different requirements of system elements such as physical entities, communication channel abstractions and controllers running on particular platforms are further identified.*

**Keywords:** Real-Time Statecharts, Hierarchical Timed Automata, Timed Automata, real-time systems, distributed systems, behavior specification.

[1]supported by the german research council (DFG) within the collaborative SFB 614 named "Self-optimizing Concepts and Structures in mechanical Engineering"

[2]supported by the International Graduate School of Dynamic Intelligent Systems.

# Contents

# 1 Introduction

To model complex reactive real-time systems, state machine specification techniques are one successful approach. However, they have to face a number of problems. On the one hand, a number of extensions for the *expressiveness* of state machines such as Statecharts [Har87] have been proposed in the field of software engineering to support the description of complex behavior. On the other hand, to specify the required temporal behavior for state-based behavior well-founded formal concepts such as Timed Automata have to be employed. While both research directions have their own merits, a combination of both worlds has to face serious problems. The Statechart extensions for structure and micro- and macro-steps which achieve the required degree of *expressiveness* assume an infinite fast machine and therefore assumes that the environment will not interfere with its execution. In contrast to this approach assumes the Timed Automata models and related research that temporal properties of the automaton are explicitly modeled to study whether the *temporal behavior* is appropriate. To obtain an applicable solution for developing real-time control software, it is further required that the specification formalism can be analyzed to proof the required properties (*predictability*) and used to derive a correct operating software (*realizability*). We will show that current proposals are not able to cover all four requirements and therefore present a solution that takes expressiveness, temporal behavior, realizability and predictability into account.

First, the basic characteristics of real-time systems and the resulting main requirements for an applicable description technique are identified in Section 2. Then, we review the underlying notion of Timed Automata and its foundations in Section 3. This includes the basic principles of state machines and its extension to the notion of Timed Automata. Extensions with the concepts of Statecharts for expressiveness as proposed with the notion of Hierarchical Timed Automata are then reviewed and defined in Section 4. In Section 5 (ExHTA) the syntax and semantics for the proposed extension in form of the *extended hierarchical timed automata* model which takes into account all four aspects and addresses the most of the crucial issues identified in Section 2 is described. The report closes with some final conclusion in Section 8.

# 2 Requirements

To identify the relevant requirements for a temporal state-based formalism that serves expressiveness as well as temporal aspects, we start with reviewing what are the main characteristics and obstacles to modelling of real-time systems in Section 2.1. In Section 2.2 we further review which main structural modelling extensions proposed in the software engineering context are appropriate in the specific domain of real-time systems. Besides reactive behavior and complex state spaces of course the temporal behavior is another prerequisite discussed in Section 2.3. Then the specific question of predictability is addressed in Section 2.4 looking on the analysis opportunities for different timed models. Efforts for the higher-level specification of a system are only reasonable, when the realizability of the concepts as reflected in Section 2.5 is also taken into account.

## 2.1 Real-Time Systems

Activities within a real-time system are described by a number of independently running tasks that interact with the environment. Tasks can be classified as *hard real-time*, *soft real-time* or *non real-time* constrained relating to a so called *utility function*. For a hard real-time task the value (utility) of a required result or effect will be zero after its deadline has been passed. For a soft real-time task the value decreases after passing the deadline and for non real-time tasks no deadline effecting the value of the task is given at all.

In general, a real-time system is composed of 3 layers: The hardware, the real-time operating system (RTOS) and the software. The hardware consists of one or multiple controllers and optional resources like memory, sensors or actuators. The RTOS is the interface between hardware and software and provides operations for accessing the resources. In every operating system, all concurrently running tasks try to access the controllers and the resources simultaneously. As every controller can handle only one task, the operating system provides a taskmanager to regulate this access. RTOS provide a so called *scheduler* as taskmanager, trying to meet the deadlines of the regulated tasks. When accessing *shared resources*, concurrent access has to be controlled, e.g. in order not to read from and write to a shared resource at the same time.

Therefore, for modelling hard real-time systems it is crucial to have appropriate means to express temporal constraints. While this can be done rather straight forward at the theoretical levels each concrete device can only execute with limited speed and therefore besides specifying what time properties are expected we still have to ensure that the resulting tasks do also meet their deadlines. For hard real-time systems therefore *predictability* is required. Usually this in only possible when for alternatives simply the worst-case is considered. E.g., usually all tasks are viewed on only by taking their *worst-case execution time* (WCET) into account.

The reactive nature of most hard real-time systems further make periodic operating tasks the underlying paradigm for real-time systems. Therefore, often multiple independent tasks scheduled either by a real-time operating system (RTOS) or run-time system are employed. The scheduling can be realized using pre-computed static schedules, e.g. controlled by a priority scheduler. Using this scheduler, a priority is assigned to every task determining the order of execution. Note, that even for a single threaded system with explicit ordering of tasks, the resulting worst- and best-case execution give lower or upper bounds on their periodic occurrence which permits to estimate whether the required deadlines will be met.

As outlined above, controllers have to cope with the described restrictions, while real world physical entities may behave arbitrary fast. Therefore, a notion for temporal state-based behavior should permit to distinguish both kinds of models and detect whether a model can be executed on a limited device or is a physical abstraction that cannot be realized by a controller. We further name these two different kinds of models

- *limited*, if they can be realized on a man made digital controller or
- *unlimited*, if they describe the behavior of a physical device that cannot be realized on a man made digital controller.

It is to be noted, that certain abstractions might result in a mix of both models. A communication subsystem, for example, consists besides the technical components and software of a transmission media or channel which may result in possible failures or delays that result in an unlimited behavior. For a given hardware, a *limited* model might still be *realizable* or not depending on its resource requirements.

## 2.2 Expressiveness

To support the design of complex reactive behavior *statecharts* [Har87] extended the flat unstructured control state space of traditional finite state machines by supporting specific concepts such as hierarchy, parallel AND-states and deep and shallow history states. Therefore, rather complex state-based behavior can often be described in a compact form. Note that besides modelling comfort also the expressiveness of automata models with or without concepts like AND states vary. Therefore the number of required states to model a specific complex behavior depends on the availability of these higher-level description concepts (cf. [DH94]).

For Statecharts the temporal behavior can be specified by the `after` and `when` constructs. A transition labeled with `after(x)` fires `x` time units after entering the source state, `when(x)` fires at point `x` in time. So `after` is used for the declaration of a relative date, `when` for an absolute. In this form, support of an unlimited device is needed to identify the point of times without delay. Besides these explicit constructs which permit to describe timeout behavior as required for the real-time domain, the temporal behavior of Statecharts is usually not specified. Thus for all side-effects and required computations no timing information is present.

Statecharts abstract from concrete timing information and assume an infinite fast device. As outlined in the preceding part about real-time systems, for a controller such an assumption is not realistic. In practice the assumption can be reduced by simply stating that the Statechart is executed fast enough to react on all relevant environment changes. Fast enough is difficult to determine and a model is required that somehow also represents this information. Therefore, an extension with explicit timing information is required.

In the case of the UML state machine [Obj01] the related micro- and macro-step execution order has been adjusted to ensure that the machine does first compute all internal effects of a received event (run-to-completion) before new external events are considered. While this simplifies the implementation of UML state machines on the other hand the delay in reaction to external events can be rather prohibitive for a real-time system. Therefore, when looking for an appropriate real-time modelling notation we have to keep in mind that reasonable worst-case reaction time may be in conflict with some features of the macro- and micro-step semantics and run-to-completion semantics.

## 2.3 Temporal Behavior

In real world physical entities operations are time-consuming. When they are executed on a limited device, their maximum duration is denoted by the worst case execution time (WCET). *Deadlines* are used to specify that an action has to finish its execution in the given time bound. Typically automata are implemented by a control task, that checks periodically the activation of transitions. So an activated transition is not fired at the point of activation, but rather during the following period of the control task. This results in a delay, caused by the limitation of the device. Specification of the deadline gives – w.r.t. the WCET – an upper bound for the acceptable delay.

When deadlines and WCETs are specified, well-known methods and algorithms can be used to determine if a schedule exists, affecting that all deadlines are met, and to determine this schedule; e.g. using a priority scheduler and assigning the priorities according to the Deadline Monotonic Priority Assignment.

The employed notion of time in a real-time system is usually limited by the maximal clock resolution of the underlying RTOS or hardware. Therefore, the clock values we can compare in a program can always be encoded by a countable set of time points. When instead of absolute clock values only relative time differences are used, a straight forward extension of an automata models towards discrete time exists where different clock values are used to determine time-dependent behavior. However, when using such clock values this usually does not include the assumption that time elapses only in discrete steps. Instead a clock value $n$ is assumed to represent the interval $[n, n+1)$ rather than a discrete point in time. Therefore, the read clock values and clock value comparisons check whether upper or lower bounds actually hold for a clock value and the underlying interpretation is that the clock therefore is in the time interval determined by the corresponding interval.

## 2.4 Predictability

The required support for predictability can vary to a great extent. Traditionally, the periodic nature of all real-time tasks is exploited to predict whether all deadlines will be met (*feasibility analysis*). If more specific system properties should be predicted or the assumption of independent periodic tasks is not valid, however, other techniques for automatic *formal verification* such as *model checking* may be employed.

### 2.4.1 Feasibility Analysis

Most approaches to real-time system design support a *feasibility analysis* to check whether a given set of periodic tasks can be served in a manner that all deadlines are met. As there are many different kinds of schedulers, for every approach exist different algorithms, for checking the feasibility.

Scheduling can be preemptive or non-preemptive, static or dynamic. One widespread manner is the *static, preemptive priority scheduling*. Using this scheduler, a priority is assigned to every task. Whenever the scheduler is assigning tasks to processors and has to distinguish between multiple tasks, it chooses the one with the highest priority. If the processor is busy with a low priority task and a high priority task arrives, then the active task is preempted and the processor is assigned to the task with the higher priority. When it is possible to change the priorities during runtime, the scheduling is called *dynamic*, otherwise *static*. Thus, the feasibility of a set of tasks strongly depends on the manner of priority assignment.

The *Rate Monotonic Approach (RMA)* [BR99], that assigns higher priorities to tasks with shorter periods, results in an optimal scheduling. That means, that the set of tasks, having rate monotonic priorities, is schedulable, if any priority assignment exists, that makes the set schedulable. RMA is limited to periodic tasks. A set of tasks, containing aperiodic tasks, the *Deadline Monotonic Priority Assignment* [BW01] (a task with a short deadline achieves a high priority), is optimal. Both approaches are static, as the priorities are computed in advance of starting the tasks and are not changed during runtime. As these approaches are widespread, there exist a couple of algorithms for accomplishing a feasibility analysis. The theorems from Liu and Layland resp. Lehocky, Sha and Ding [BR99] confirm the feasibility of a set of tasks scheduled by RMA. In [Jos96] an algorithm is presented, accomplishing an analysis for priority scheduling independent of the manner of priority assignment.

In general, different tasks may access the same resources (like memory). Obviously, the access to these *shared resources*, has to be controlled as there will be problems, when a task, writing to a memory address, is interrupted by an higher prioritized thread, which writes to the same address. To avoid this problem a task, accessing a shared resource, blocks all (higher prioritized) tasks, that try to access this resource. Unfortunately, this solution results in an effect called *Priority Inversion*. This occurs when due to blocking the highest prioritized task finishes its execution at last (see [Jos96, BR99] for a complete example). Because of this effect, protocols have been investigated to minimize the blocking times. Widespread protocols are the *Priority Inheritance Protocol* and the *Priority Ceiling Emulation* [BR99].

### 2.4.2 Formal Verification

A behavioral modelling technique with well defined semantics permits to analyze the required temporal model properties using, e.g., temporal logic [Pnu77]. Given a specification in a temporal logic $\phi$ and a model $M$ there exist a number of techniques to prove that the model fulfils the specification ($M \models \phi$). Following [Lam77] such specifications can be further separated into *safety* and *liveness* conditions. While safety conditions ensure that something bad never happens, do liveness conditions ensure that something good will happen. Note, that thus liveness includes progress (something will happen).

The basic notion of temporal logic does only cover causal ordering, but does not permit to express properties with explicit real-time constraints. Therefore, a number of extensions with different time models have been proposed.

A first natural approach is to employ and extend available techniques for un-timed systems such as model checking. If a countable set of time points are sufficient a discrete notion of time can be used where, for example, the natural numbers denote all possible time points. When instead of absolute values only relative time differences are used, a straight forward extension of an automata model towards discrete time exists and available techniques for its analysis can be employed (cf. [EMSS90]).

A Tool for such a discrete timed model is, for example the model checker RAVEN [RK99, RK00] which employs clocked computation tree logic (CCTL) as specification language. For a timed step with a delay of more than one minimal time unit in such a discrete interpretation of time the question arises what is the correct interpretation of the intermediate states. Do these intermediate states correspond to "copies" of the source state , "copies" of the target state, or is in fact nothing known about these intermediate states. Therefore, in [LS01] a temporal logic and related model checking techniques are proposed that assume the latter while in RAVEN the first case is assumed.

However, the assumption that time elapses only in discrete steps is rather strong. A more serious problem in practice is to chose a sufficient small minimal discrete time value. On the one hand the choice has to ensure that the above assumption of a correct time abstraction holds. On the other hand the applicability of analysis techniques crucially depends on a chosen minimal discrete time value large enough. Therefore a systematic way to determine such a minimal discrete time value is required, before these approaches can be safely employed to automatically predict and analyze the required system properties.

As outlined in [HNSY92], when a digital clock value $n$ is assumed to represent the interval $[n, n+1)$ rather than a discrete point in time, for restricted cases of models or temporal logic formulas results proven for the digitalization will also hold for the more realistic continuous time model. When also stronger specifications in the digital model are used and the digitization granularity is refined we can further approximate step-wise the required continuous time property.

To further automate this task, an approach that automatically determines a finite abstraction of a given real-time system is more appropriate than a manual approximation. For a continuous model of time, *timed automata* [AD90] as automata theoretic approach provides such a solution. It supports continuous clocks and employs the property that the clocks are only tested for a finite number of rational constants for only rational constants to construct of a finite abstraction in form of a region-graph (cf. [AD90]). With COSPAN [AK96], HyTECH [HHWT95a], KRONOS [DOTY96], and UPPAAL [LPY97, PL00] a number of tools for model checking Timed Automata exist. Some of them further support hybrid and parameterized verification.

## 2.5 Realizability

A real-time system consists of hardware (controller, resources, actuators, sensors etc.) and its software. Concerning the integration between software and hardware there are 2 common platforms: (1) a real-time operating system (RTOS) is used to decouple the real-time application from the hardware and (2) the special I/O architecture of a micro-controller is directly employed to connect the real-time application with its environment. The advantages of the RTOS approach are that it provides operations with uniform access to the underlying hardware and can control the scheduling of multiple tasks which includes their access to the (shared) resources. Alternatively, when a micro-controller is employed, the whole logic of the application has to be covered by a single task. In this case the scheduler, respectively the RTOS, is not needed. One of the advantages of the multi-task architecture is, that the tasks can control each other and handle errors. In a single thread application, this is not possible.

Real-time systems are often also *embedded systems* and thus besides the described heterogeneity w.r.t. scheduling also the available resources and execution restrictions can vary. When modelling a controller device these different forms (programmable logic controller (PLC), micro-controller (MC), and computer) restrict the delay and procedure the software can observe and control its environment. E.g., for PLC we usually have a bounded cycle time, while computer equipped with a RTOS permit a great variety of task periods which are served by the scheduler.

For the considered case of high-level behavior models, a suitable mapping to an implementation is required. Whether a mapping is possible or not has thus to be detected also at the high-level model. Otherwise, a delayed detection can result in rather wasted design efforts. Therefore, feasibility analysis at the level of the high-level behavior model is required.

# 3 Timed Automata

To discuss the foundations for extended state machine models for modelling temporal behavior, we introduce the required terminology and formal treatment by first reviewing the basic definition and terms for a finite automaton.

The finite automaton is the most fundamental concept for state-based behavior. Their main elements are a finite set of control states (locations) denoted by $S$, a contained set of start locations $S_0 \subseteq S$ and a set of transitions $T$, where each $t \in T$ further describes which changes between the locations are permitted depending either on internal choice or the environment. In a hardware oriented view the environment is modelled by input $i \in I$ and output $o \in O$ signals. The resulting output $o$ can be further described either as function of $S$ (Moore automata) or $S$ and $I$ (Mealy automata). In the considered software context instead a Mealy automata sending and receiving possible events $a \in A$ rather than permanently provided input and output signals is usually employed.

**Definition 1** *A* finite automaton $M$ *is a* 4*-tuple* $(S, S_0, A, T)$ *with*

- $S$ *a finite non empty set of locations,*
- $S_0 \subseteq S$ *the subset of start locations,*
- $A$ *a finite set of events including the internal event* $\tau \in A$*.*
- $T \subseteq S \times A \times S$ *a finite set of transitions* $t = (s, a, s') \in T$ *with*
  - $s \in S$ *the source location,*
  - $a \in A$ *the related action, and*
  - $s' \in S$ *target location.*

By convention for a transition $t = (s, a, s') \in T$ we also write $s \xrightarrow{a} s'$. Using $S$ as the set of possible configurations the resulting transition system is as follows.

**Definition 2** *For a* finite automaton $M = (S, S_0, A, T)$ *the set of all possible configurations is simply* $S$*. The initial possible configurations are the elements of* $S_0$*. For a finite automate with* *current configuration* $l_0$ *we further have the enabling condition* $\text{Enabled}(l_0, (l, a, l')) \iff (l_0 = l)$ *and related configuration transition function* $\Gamma_{(l,a,l')}(l) := l'$ *resulting in the following action rule:*

$$\frac{\text{Enabled}(l, t)}{l \xrightarrow{t} \Gamma_t(l)}\ \textit{action}$$

In the simply case of a finite automaton the distinction between the syntactical notion of the location set and the possible configurations is rather superficial. However, in the following we will study extended concepts for automata where both notions are not synonymous any more.

As long as finite behavior is considered additionally often a set of accepting locations $S_f \subseteq S$ is defined. The valid finite traces $s_0 \xrightarrow{a_0} s_1 \cdots s_n$ with $s_0, s_1, \ldots, s_n \in S^*$ are given by all finite sequence of transitions starting with any start location $s_0 \in S_0$ which end at an accepting location $s_n \in S_f$ (acceptance condition). For infinite traces $s_0 \xrightarrow{a_0} s_1 \cdots$ with $s_0, s_1, \ldots \in S^\omega$, however, *Büchi automata* [Büc62] have to be used where instead for any possible infinite trace all states $s_a$ of the accepting location set $S_a$ have to be visited infinite often ($|\{i|s_i = s_a\}| = \infty$) to fulfill the acceptance condition.

This way to describe acceptance of infinite behavior is rather elegant, however, not really intuitive when modelling a system. This mismatch is related to the following underlying fact: For the acceptance of finite or non finite regular languages in form of an automata *angelic non-determinism* [HU79] is employed, which essentially assumes that choices are always made with global knowledge. When modelling system behavior in a constructive fashion by composing multiple independently operating subsystems, however, *erratic non-determinism* [Bro86] is natural, where each subsystem is assumed to behave arbitrary.[3] Therefore, a different technique to model the liveness and progress properties of the system is required when modelling systems rather than characterizing the acceptance of a language.

## 3.1 Timed Automata

Automata can be extended to also describe the temporal behavior using a continuous notion of time. The term *timed automata* also includes a number of automata theoretic approaches with real-value clocks. The main characteristic is that for the models often a finite abstraction can be constructed, which results in a number of related model checking approaches. One basic notion of Timed Automata also named *timed graphs* [ACD90] uses real-value clocks to further restrict the occurrence of transitions. This notion, however, does not exclude that the automata may idle forever in any reached configuration. The original proposal [AD90] avoids this problem extending the concepts of Büchi automata [Büc62] and thus accepts only such infinite runs which also infinitely often visits states marked as accepting. While this notion can be used to specify *progress conditions* in [HNSY92] a notion for Timed Automata also named *timed safety automata* is proposed that avoids to require accepting states and instead introduces clock constraints for control states to model progress guarantees explicitly. We will further use this last notion as basic notion of *timed automata* and refer to [AD94, Pet99] for more detailed overviews.

The basic extension w.r.t. finite automata is that besides the control locations $S$ a set of real-value clocks $C$ is also part of the transition system configuration in form of a clock binding $\nu \in C_\nu = [C \to \mathbb{R}_+]$. $\nu^0$ is the special binding which assigns to each clock variable 0. To control the temporal behavior clock constrains $C_c = [[C \to \mathbb{R}_+] \to \{\mathsf{true}, \mathsf{false}\}]$ in form of time guards for transitions and time invariants for locations are employed. To reset the clock values as required specific clock updates $C_u = [C_\nu \to C_\nu]$ are used.

[3] For a more detailed discussion on the different forms of non-determinism see [WM97].

**Definition 3** *A* Timed Automaton *$M$ is a tuple $(S, S_0, T, Inv, A, C)$ where*

- *$S$ a finite non empty set of locations,*
- *$S_0 \subseteq L$ the subset of start locations,*
- *$Inv : S \to C_c$ assigns each state a clock constraint,*
- *$A$ a finite set of events including the internal event $\tau \in A$,*
- *$C$ a finite set of clocks and,*
- *$T \subseteq S \times A \times C_c \times C_u \times S$ a finite set of transitions $t = (l, a, g, r, l') \in T$ with*
  - *$l \in S$ the source location,*
  - *$a \in A$ the related action,*
  - *$g \in C_c$ a clock constraint named time guard,*
  - *$r \in C_u$ a clock update, and*
  - *$l' \in S$ target location.*

In the resulting transition system two forms of transitions are present. The transitions associated with each transition of $T$ occur in a timeless step. Additionally, so called time steps which remain within the same control state (location) are possible where any arbitrary time delta $d$ elapses.

**Definition 4** *For a* Timed Automaton *$M = (S, S_0, T, A, C)$ the state space is build by all pairs $(l, \nu) \in S \times C_\nu$. The initial possible states are all such pairs $(l, \nu^0)$ with $l \in S_0$. For a Timed Automaton in configuration $(l, \nu)$ we further define enabling by $\mathrm{Enabled}((l'', \nu), (l, a, g, r, l')) \iff l'' = l \wedge g(\nu) \wedge Inv(l')(r(\nu))$ and the resulting configuration transition $\Gamma_{(l,a,g,r,l')}(l, r(\nu)) := (l', r(\nu))$. Thus, for a transition $t \in T$ the following action rule results:*

$$\frac{\mathrm{Enabled}((l, \nu), t)}{(l, \nu) \xrightarrow{t} \Gamma_t(l, \nu)} \; action.$$

*Time can further elapse in so called delay steps and for the corresponding update function $\Gamma_d((l, \nu)) := (l, \nu + d)$, where $\nu + d$ stands for the current clock assignment plus the delay for all the clocks, we have*

$$\frac{\mathrm{Inv}(l)(\nu + d)}{(l, \nu) \xrightarrow{d} \Gamma_d(l, \nu)} \; delay.$$

Typically the updates and constraints for the continuous clock variables are further syntactically restricted to ensure that a finite abstraction [AD90] exists. For the transition updates $C_u$ hold that clocks are usually only reset to 0 or integer numbers. For the time constraints in form of time invariants for locations and time guards for transitions usually the expressions are limited for $x$, $y$, clocks and constants $n$ to the conjunctive combination of the form $x \Box n$ or $x - y \Box n$ with $\Box \in \{<, \leq, =, \geq, >\}$.

For this restricted notion of time constraints the reachability problem for Timed Automata is decidable when the coefficients ($n$) in the guards are rational numbers [AD90]. The reachability problem becomes as soon undecidable as the coefficients are chosen, for example, from the set $\{1, \sqrt{2}\}$ [Pur99]. When additive clock constraints are used, already for no more than 4 clocks the emptiness problem is in general undecidable [BD00].

## 3.2 Consistency

The described Timed Automata model can further result in subtle semantic problems. For example, for the employed notion of a Timed Automaton a configuration without any further possible timeless step named deadlock may be reached.

**Definition 5** *A configuration* $(l,\nu)$ *is a* deadlock *iff* $\forall t \in T$ *hold* $\neg\mathrm{Enabled}((l,\nu),t)$.

A configuration without further external visible timeless steps still describes a reasonable behavior. If, however, the related time invariant of the control state (location) expires and further time elapses an inconsistent temporal behavior has been specified.

**Definition 6** *A configuration* $(l,\nu)$ *is a* time stopping deadlock *iff it is a* deadlock *and for any* $d > 0$ *hold* $\mathrm{Inv}(l)(\nu + d) = \mathsf{false}$.

Another problem of the presented approach to model continuous time behavior is caused by the employed interleaving of time steps and discrete action steps. While for arbitrary delta $d$ and $d'$ time steps still describe a reasonable behavior, a sequence $(l_0,\nu_0)\xrightarrow{d_0}(l_1,\nu_1)\xrightarrow{t_0}(l_2,\nu_2)\xrightarrow{d_1}(l_3,\nu_3)\xrightarrow{t_1}(l_4,\nu_4)\ldots$ with alternating time and action step is not reasonable when infinitely many discrete action steps occur in a finite amount of time ($\exists d = \sum_{i=1}^{\infty} d_i$).

**Definition 7** *An infinite sequence of configurations and steps* $(l_0,\nu_0)\xrightarrow{x_0}(l_1,\nu_1)\xrightarrow{x_1}(l_2,\nu_2)\xrightarrow{x_2}(l_3,\nu_3)\xrightarrow{x_3}(l_4,\nu_4)\ldots$ *with* $x_i \in (T \cup \mathbb{R}_+)$ *is considered to be* zeno *iff the number* $|\{i | x_i \in T\}|$ *is infinite while for the elapsed time a finite upper bound* $d$ *exists such that* $\sum_{x_i \in \mathbb{R}_+} d_i \leq d$.

The problem of zeno behavior will usually only occur for unlimited systems, because there exists a lower bound $t_l$ for all the times required to execute a transition in a specific situation. Therefore, as long as besides the zeno behavior a reasonable non-zeno behavior exists it is usually assumed to simply exclude the traces with zeno behavior. When, however, unlimited systems and their physics is considered, the result of an infinite but convergent chain of transitions and time steps may indeed determine the correct behavior of the system.

## 3.3 Extensions

Besides this basic notion of Timed Automata several extensions have been proposed to simplify the specification of temporal behavior.

For a configuration $(l,\nu)$ any enabled transition $t$ may occur or not as long as the time invariant of the current location is valid. The only case when some transition will be executed definitely is when the invariant does not permit any further delay. Therefore, the concept of *urgent transitions* [HHWT95b, DY95, BLL+95] has been proposed where time can only elapse when no urgent transition is enabled.

Formally for $T_u \subseteq T$ the subset of all urgent transitions which are often further restricted to have empty guards ($g = \mathsf{true}$) we can define whether an urgent transition is enabled $\mathrm{UrgentEnabled}((l,\nu)) := \exists t \in T_u \mathrm{Enabled}((l,\nu),t)$ and simply adjust the delay rule accordingly.

$$\frac{\mathrm{Inv}(l)(\nu + d) \wedge \neg\mathrm{UrgentEnabled}((l,\nu))}{(l,\nu)\xrightarrow{d}(l,\nu + d)}\ \mathit{delay}$$

To simplify the modelling of complex state spaces besides explicit control states also *integer variables* [LPY97] have been invented. The resulting reachable system configurations simply have to include a binding for the variables, too.

Note, that in the presented basic notion of Timed Automata the labels $a \in A$ do not effect the execution of a transition. For a system of multiple parallel executed Timed Automata *synchronous* *communication* using channels in a *Calculus of Communicating Systems (CCS)* [Mil89] style has been proposed in [LPY97]. This permits to coordinate two parallel Timed Automata in a single atomic step. Two transitions with compatible channel events $e!$ and $e?$ are combined into a single transition of the combined system.

Another way to let multiple parallel operating Timed Automata interact are *shared variables* [LPY97]. In contrast to the synchronous communication do shared variables facilitate an asynchronous exchange of information between parallel Timed Automata.

### 3.4 Summary

A system of Timed Automata with synchronous communication and shared variables provides already the required fundamental concepts to describe the complex real-time behavior of large systems consisting of a number of components. For reasonable small systems of Timed Automata model checking techniques further provide the required analysis and help to predict whether required properties hold. Therefore, the requirements temporal behavior and predictability are covered. However, the presented basic formalisms are still limited with respect to expressiveness and realizability. In the next section we therefore consider an approach that further extend the only flat structures supported by Timed Automata towards complex state spaces as supported by Statecharts.

## 4 Hierarchical Timed Automata

As identified in Section 2.2 the fundamental extensions to cope with complex behavior are hierarchy, parallelism and history as provided by *statecharts* [Har87] and its UML derivation *UML state machines* [Obj01]. The notion of a Hierarchical Timed Automata model [DMY02] provides the most fundamental structural concepts as well as a reasonable continuous time semantics. This notion supports model checking by a mapping to plain Timed Automata as employed in the UPPAAL tool [LPY95]. As part of the AIT-WOODDES research project a corresponding UML state machine profile is further planed. In the following we define Hierarchical Timed Automata as introduced in [DMY02] and summarize their exact limitations w.r.t. the in Section 2 identified requirements.

### 4.1 Syntax

**Definition 8** *A* Hierarchical Timed Automaton *is a tuple* $\langle S, S_0, \delta, \sigma, V, C, Inv, Ch, T\rangle$ *where*

- $S$ *is a finite set of locations.* $root \in S$ *is the root.*
- $S_0 \in S$ *is a set of initial locations.*
- $\delta : S \to 2^S$ *is a function, mapping a location* $l$ *to all possible substates of* $l$ *and generating a tree structure with* $root$ *as the root. Applying* $\delta$ *on sets, delivers the intuitively expected results.*
- $\sigma : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ *associates a type to every location*
- $V, C, Ch$ *are sets of variables, clocks, and channels and are used for the specification of* Guard*,* Reset*,* Sync*, and* Invariant *as described in the following.*

- *$Inv : S \to$ Invariant assigns an invariant to every location.*
- *$T \subseteq S \times (Guard \times Sync \times Reset \times \{\mathsf{true}, \mathsf{false}\}) \times S$ is the set of transitions. A transition is composed of a source ($l$) and a target location ($l'$), a guard $g$, an assignment $r$ (including clock resets), and an urgency flag $u$. The notation $l \xrightarrow{s,g,r,u} l'$ is used for a transition $t = (l, s, g, r, u, l')$. We omit $s, g, r, u$ when they are necessarily absent (or* false*, in the case of $u$).*

*Guards, synchronizations, resets, and assignments use* Guard*,* Reset *and* Sync*, as described in the following:*

- *$V$ is the finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a superstate $l$.*
- *Let $C$ be the finite set of clock variables. $C(l) \subseteq C$ contains the clocks local to a superstate $l$. If $l$ is a history state, $C(l)$ contains only clocks, that are declared as forgetful. The other clocks of $l$ (local clocks) belong to $C(root)$.*
- *Let $Ch$ be the finite set of synchronization channels. $Ch(l) \subseteq Ch$ contains the channels that are local to a superstate $l$, i.e., it is not possible to synchronize along a channel $c \in Ch(l)$ between one transition inside $l$ and one outside $l$.*
- *$Ch$ leads to the finite set of channel synchronizations(*Sync*). It holds: $c \in Ch \Rightarrow c?, c! \in Sync$. If $s \in Sync$, then $\overline{s}$ denotes the corresponding complementary(e.g. $\overline{c!} = c?$ and $\overline{c?} = c!$).*
- *Data constraints are boolean expressions of the form $A \Box A$ with $A$ is an arithmetic combination of elements from $V$ and $\Box \in \{<, >, =, \leq, \geq\}$. Clock constraints are boolean expressions of the form $x \Box n$ or $x - y \Box n$, with $x, y \in C$, $n \in N$ and $\Box \in \{<, >, =, \leq, \geq\}$. A clock constraint is called* downward closed*, if $\Box \in \{<, =, \leq\}$. A guard is a finite conjunction over data and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. The set of guards and invariants, joint with $\{\mathsf{true}, \mathsf{false}\}$ are called* Guard *respectively* Invariant*.*
- *Clock resets are of the form $x := 0$, with $x \in C$. Data assignments are of the form $v := A$, with $v \in V$ and $A$ is an arithmetic combination of elements from $V$.* Reset *is the set of clock resets and data assignments.*

The function $TYPE : S \to \{\mathsf{true}, \mathsf{false}\}$ is used as an abbreviation for the expression $\sigma(l) = AND$. As the type $HISTORY$ is an important one, the function $HENTRY$ is defined similary as $HENTRY(l) = ENTRY(l) \vee HISTORY(l),\ l \in S$.

For support, we define the following functions:

- the parent function $\delta^{-1}(l) = l'$ with $\delta(l') = l$ and $l \neq root$.
- $\delta^*(l)$ returns the set of all nested locations of $l$.
- $\delta^{-*}(l)$ denotes the set of ancestors of the location $l$, including $l$.
- $\delta^\times(l) = \delta^*(l) \setminus \{l\}$
- $\widetilde{\delta}(l) = \{n \in \delta(l) | BASIC(n) \vee XOR(n) \vee AND(n)\}$ to denote the children-locations, being of Type $BASIC$, $XOR$, or $AND$.
- $V^+(l) = \bigcup_{n \in \delta^{-*}(l)} V(l)$ to denote all variables in the scope of the location $l$. $C^+(l)$ and $Ch^+(l)$ are defined analogously.

In [DMY02] a given hierarchical Timed Automaton is only well-formed when the following conditions are fulfilled.

**Definition 9** *A Hierarchical Timed Automaton is* well-formed *iff*

1. - *The Hierarchical Timed Automaton is rooted at* root*:* $S = \delta^*(root)$*,*
   - *Basic nodes are empty:* $BASIC(l) \Leftrightarrow \delta(l) = \emptyset$*,*
   - *substates of AND superstates are not basic:* $AND(l) \wedge n \in \delta^*(l) \Rightarrow \neg BASIC(n)$*, and*
   - *invariants of "pseudo-locations" are trivial:* $HENTRY(l) \vee EXIT(l) \Rightarrow Inv(l) = \mathsf{true}$*.*

2. $S_0$ *belongs to a situation of consistent and proper control, i.e.,* $root \in S_0$ *and for every* $l \in S_0$ *the following holds:*
   - $l$ *is a proper location:* $BASIC(l) \vee XOR(l) \vee AND(l)$*,*
   - $S_0$ *does contain all parents:* $l = root \vee \delta^{-1}(l) \in S_0$*,*
   - *only one alternative of an XOR-state is present:* $XOR(l) \Rightarrow |\delta(l) \cap S_0| = 1$*, and*
   - *all substates of an AND-state are proper states:* $AND(l) \Rightarrow \delta(l) \cap S_0 = \widetilde{\delta}(l)$*.*

3. *There are no conflicts in assignments in synchronizing transitions:* $l_1 \xrightarrow{g,c!,r,u} l_1'$, $l_2 \xrightarrow{g',c?,r',u'} l_2' \in T \Rightarrow vars(r) \cap vars(r') = \emptyset$*, where* $vars(r)$ *is the set of integer variables occurring in* $r$*. A similar constraint is used to handle transitions originating in the entry of an AND superstate. For* $l \xrightarrow{g,s,r,u} l' \in T, g, r$ *are defined over* $V^+(\delta^{-1}(l)) \cup C^+(\delta^{-1}(l))$ *and* $s$ *is defined over* $Ch^+(\delta^{-1}(l))$ *to ensure a proper static scoping.*

4. *Let* $e \in S, HENTRY(e)$*.*
   - *If* $XOR(\delta^{-1}(l))$*, then* $T$ *contains exactly one transition* $e \xrightarrow{r} l'$*.*
   - *If* $AND(\delta^{-1}(l))$*, then* $T$ *contains exactly one transition* $e \xrightarrow{r} e_i$ *for every proper substate* $l_i \in \widetilde{\delta}(\delta^{-1}(l))$*, and* $e_i \in \delta(l_i)$*.*
   - *In case of* $HISTORY(e)$*, the default history locations are specified by the outgoing transitions. If* $AND(s) \wedge HISTORY(s)$*, then every substate* $l$ *of* $s$ *has to provide either a history entry or a default entry.*

5. *Transitions, connecting States, that are neither entries nor exits, at different hierarchical levels, are disallowed. Note that transitions are not allowed to lead directly from entries to exits. A Transition* $l \xrightarrow{g,s,r,u} l'$ *is called "pseudo-transition" when* $HENTRY(l)$ *or* $EXIT(l')$ *holds. If* $HENTRY(l)$ *holds, only the form* $l \xrightarrow{r} l'$ *is valid. In the case of* $EXIT(l')$*, only the form* $l \xrightarrow{g} l'$ *is allowed. If* $EXIT(l) \wedge EXIT(l')$ *holds, the form is restricted to* $l \rightarrow l'$*.*

## 4.2 Semantics

The semantics of the Hierarchical Timed Automata model is further defined in [DMY02] using a configuration of the form $(\rho, \mu, \nu, \theta)$ which contains the active locations, the integer variable values, the clock values, and the history.

- $\rho : S \rightarrow 2^S$ delivers a set containing all active substates of a superstate $s$. An inactive state $s$ results in $\rho(s) = \emptyset$. We define another function for support: $Active(l) = l \in \rho^\times(root)$, where $\rho^\times(l) = \rho \setminus \{l\}$. Notice that $Active(l) \Leftrightarrow l \in \rho(\delta^{-1}(l))$.

- $\mu : S \rightarrow (\mathbb{Z})^*$ gives rise to the values of the variables of a given superstate $l$. $\mu$ returns a finite tuple of integer numbers. It is required that $|\mu(l)| = |V(l)|$ and that $\mu$ is consistent with respect to the value of shared variables (i.e., always maps to the same value). $\mu(l)(a)$ defines the value of variable $a \in V(l)$. If $\neg BASIC(l)$ the tuple is extended by local variables initialised by a default value (0). We use the abbreviation $0^{V(l)} = (0, 0, ..., 0)$ with arity $|V(l)|$. We demand $\neg Active(l) \Rightarrow \mu(l) = \lambda$ (the empty tuple).

- $\nu : S \rightarrow (\mathbb{R}^+)^*$ gives –similar to $\mu$– rise to the values of the clocks of state $l$. It is required that $|\nu(l)| = |C(l)|$ and $\neg Active(l) \Rightarrow \nu(l) = \lambda$.

- $\theta = (\theta_{state}, \theta_{var})$ denotes the substate(s) and local variables of a superstate that will become valid by entering this superstate via a history entry. Thus, $\theta_{state}$ is used to restore $\rho(l)$, while $\theta_{var}(l)$ returns a tuple of integer values to restore $V(l)$. Clocks are not influenced by the history.

The existence of a history entry is handled with the predicate $HasHistory(l) = \exists n \in \delta(l)|HISTORY(n)$. $HEntry(l)$ is the unique history entry of a superstate $l$ if $HasHistory(l)$ holds, the default entry of $l$ otherwise. If $BASIC(l)$ then $HEntry(l) = l$ holds. $HEntry(l)$ is undefined for all other cases. Initially, $\forall l \in \{S|HasHistory(l)\} \Rightarrow \theta_{state}(l) = HEntry(l) \wedge \theta_{var}(l) = 0^{V(l)}$ holds.

The function $Targets_\theta$ to describes the locations reached by following a fork is defined as

$$Targets_\theta(l) = L \cup \bigcup_{l \in L} \{n|n \in \theta_{state}(l) \wedge HISTORY(l)\} \cup \{n|(l, \tau, \mathsf{true}, r, n) \in T \wedge ENTRY(l)\} .$$

$Targets_\theta(l)$ is used as abbreviation for $Targets_\theta(\{l\})$. $Targets^*_\theta$ is the reflexive transitive closure of $Targets_\theta$.

Firing a transition $t : l \xrightarrow{g,s,r,u} l'$ can be split up into 3 steps: 1. executing a join to exit $l$, 2. taking the proper transition $t$ itself, and 3. executing a fork at $l$. In case of $BASIC(l)$ (respectively $BASIC(l')$), part 1. (respectively 3.) is trivial. The 3 steps define a run-to-completion step. A run-to-completion step is formally represented by a transformation function $\Gamma_t$, where $t$ is a transition. The 3 parts of the step are described as follows:

1. join : $(\rho, \mu, \nu, \theta)$ is transformed to $(\rho^1, \mu^1, \nu^1, \theta^1)$ as follows: $\rho$ is updated to $\rho^1 := \rho[\forall n \in \rho^\times(l)|n \rightarrow \emptyset]$. $\mu$ is updated to $\mu^1 := \mu[\forall n \in \rho^\times(l)|n \rightarrow \lambda]$. $\nu$ is updated to $\nu^1 := \nu[\forall n \in \rho^\times(l)|n \rightarrow \lambda]$. If $EXIT(l)$, the history is recorded. Let $H$ be the set of superstates $h \in \rho^\times(\delta^{-1}(l))$, where $HasHistory(h)$ holds. Then $\theta^1_{state} := \theta_{state}[\forall h \in H|h \rightarrow HEntry(\rho(h))]$ and $\theta^1_{var} := \theta_{var}[\forall h \in H|h \rightarrow \mu(h)]$. If $\neg EXIT(l)$ or $H = \emptyset$, then $\theta^1 := \theta$.

2. proper transition part : $(\rho^1, \mu^1, \nu^1, \theta^1)$ is transformed to $(\rho^2, \mu^2, \nu^2, \theta^2) := (\rho^1[l'/l], r(\mu^1), r(\nu^1), \theta^1)$. $r(\mu^1)$ denotes the updated values of the integers after the assignments and $r(\nu^1)$ the updated clocks after the resets.

3. fork : $(\rho^2, \mu^2, \nu^2, \theta^2)$ is transformed to $(\rho^3, \mu^3, \nu^3, \theta^3)$ by moving the control to all proper locations reached by the fork, i.e. those in $Targets^*_{\theta^2}(l')$. Note that $\rho^2(n) = \emptyset$ for all $n \in \delta^\times(l')$. Thus we can compute $\rho^3$ from $\rho^2$ by modifying it iterativly for all $n \in Targets^*_{\theta^2}(l')$ for $ENTRY(n)$ such that $\rho^3(\delta^{-2}(n)) := \rho^3(\delta^{-2}(n)) \cup \{\delta^{-1}(n)\}$ or else $\rho^3(\delta^{-1}(n)) := \{n\}$. $\mu^3$ is derived from $\mu^2$ by first initializing all local variables of the superstates $s$ in $Targets^*_{\theta^2}(l')$, i.e. $\mu^3(V(s)) := 0^{V(s)}$. If $HasHistory(s), \theta_{var}(s)$ is used instead of $0^{V(s)}$. Then all variable assignments and clock-resets along the pseudo-transitions belonging to this fork are executed to update $\mu^3$ and $\nu^3$. The history does not change, $\theta^3$ identical to $\theta^3$.

As mentioned above, part 1. and 3. correspond to the identity transformation, if $BASIC(l)$ and $BASIC(l')$ hold. The transformation function $\Gamma_t$ of a transition $t$ returns the result of the 3 steps: $\Gamma_t(\rho,\mu,\nu,\theta) = (\rho^3,\mu^3,\nu^3,\theta^3)$. $\rho^{\Gamma_t} = \rho^3$ and $\nu^{\Gamma_t} = \nu^3$ is used if the context is definite.

A superstate $s$ can only be exited, if all its parallel substates can synchronize on this exit. The function $\mathrm{PreExitSets}\,(l)$ returns a familiy of sets. Every element $X$ of $\mathrm{PreExitSets}\,(l)$ is a set of exits. By firing the transitions of *all* exits in $X$, all substates synchronize.

$$\mathrm{PreExitSets}\,(l) = \begin{cases} \left.\begin{array}{l} \bigcup_{n_1,\ldots,n_k} \oplus_{1\le i\le k}\mathrm{PreExitSets}\,(n_i)\,, \text{ where} \\ k = \left|\widetilde{\delta}\left(\delta^{-1}(l)\right)\right|,\ \{n_1,\ldots,n_k\}\subseteq \delta^{\times}\left(\delta^{-1}(l)\right), \\ \forall i.\mathrm{EXIT}\,(n_k)\wedge n_i \to l \in T \\ \delta^{-1}(\{n_1,\ldots,n_k\}) = \widetilde{\delta}\,(l) \end{array}\right\} \text{if } \mathsf{EXIT}\,(l)\wedge \mathsf{AND}\left(\delta^{-1}(l)\right) \\ \{\{l\}\}\cup\left(\bigcup_{m\in\delta(\delta^{-1}(l))\wedge m\xrightarrow{g,r} l\in T}\mathrm{PreExitSets}(m)\right), \text{if } \mathsf{EXIT}(l)\wedge\mathsf{XOR}(\delta^{-1}(l)) \\ \{\}\,, \text{if } \mathsf{BASIC}\,(l) \end{cases}$$

The operator $\oplus$ is the below described product over families of sets which is like intuitionally expected extended for a given arbitrary finite number of arguments.

$$\oplus : \left(2^{2^S}\right)\times\left(2^{2^S}\right)\to\left(2^{2^S}\right) \qquad \oplus\left(\{A_1,\ldots,A_a\},\{B_1,\ldots,B_a\}\right) = \{A_1\cup B_1,\ldots,A_a\cup B_a\}.$$

To denote the active leaves of the tree, described by $\rho$, $Leaves(\rho,\ \mathrm{l}) = \{n\in\rho^{\times}(l)|\rho(n)=\emptyset\}$ is used. $Leaves(\rho,\ \mathrm{l})$ is used to define predicates for the transition system rules for $\longrightarrow$.

The predicate, expressing that all the substates of a state $l$ can synchronize on a join, is:

$\mathrm{JoinEnabled}\,(\rho,\ \mu,\ \nu,\ l) = \mathrm{BASIC}(l)\vee\exists X\in\mathrm{PreExitSets}\,(l)\,.\,\forall n\in\mathrm{Leaves}\,(\rho,\ l):\ \exists n'\in X.\, n\xrightarrow{g} n'\wedge g\,(\mu,\nu)$

The invariants of a location are denoted by $Inv_\nu : S\to\{\mathsf{true},\mathsf{false}\}$, which evaluates the invariant of a location dependant on a clock evaluation $\nu$. The predicate $Inv(\rho,\nu) = Inv(\rho,\nu) = \bigwedge_{n\in\rho^{\times}(root)} Inv_\nu(n)$ is used to express, that for the state denoted by $\rho$ and for the clock valuation $\nu$ all invariants hold. The predicate $Enabled : T\to\{\mathsf{true},\mathsf{false}\}$ describes whether $t$ is enabled:

$$Enabled\left(t : l\xrightarrow{g,\,s,\,r,\,u} l',\rho,\mu,\nu\right) = \neg EXIT(l')\wedge g(\mu,\nu)\wedge JoinEnabled(\rho,\mu,\nu,l)\wedge Inv(\rho^{\Gamma_t},\nu^{\Gamma_t}).$$

As urgency has precedence over delay, it has to be well-known wether an urgent transition is enabled. This information is delivered by the predicate UrgentEnabled over a configuration.

$$\begin{aligned} \mathrm{UrgentEnabled}\,(\rho,\mu,\nu) &= \left(\exists t : l\xrightarrow{g,r,u} l'\wedge u\wedge Enabled(t,\rho,\mu,\nu)\right)\bigvee \\ &= \Big(\exists t_1 : l_1\xrightarrow{g_1,c!,r_1,u_1} l_1', t_2 : l_2\xrightarrow{g_2,c?,r_2,u_2} l_2'\wedge \\ &\quad Enabled(t_1,\rho,\mu,\nu)\wedge Enabled(t_2,\rho,\mu,\nu)\wedge(u_1\wedge u_2)\Big) \end{aligned}$$

The resulting rules for the transition system steps $\longrightarrow$ for a transition $t\in T$ (action), a time step (delay), and synchronization (sync) are:

$$\frac{Enabled\left(t : l\xrightarrow{g,r,u} l',\rho,\mu,\nu\right)}{(\rho,\mu,\nu,\theta)\xrightarrow{t}\Gamma_t(\rho,\mu,\nu,\theta)}\ action \qquad \frac{Inv(l)(\rho,\nu+d)\quad \neg\mathrm{UrgentEnabled}(\rho,\mu,\nu)}{(\rho,\mu,\nu,\theta)\xrightarrow{d}(\rho,\mu,\nu+d,\theta)}\ delay$$

$$\frac{\begin{array}{ll} Enabled\left(t_1 : l_1\xrightarrow{g_1,c!,r_1,u_1} l_1',\rho,\mu,\nu\right) & l_1\notin\delta^{\times}(l_2) \\ Enabled\left(t_2 : l_2\xrightarrow{g_2,c?,r_2,u_2} l_2',\rho,\mu,\nu\right) & l_2\notin\delta^{\times}(l_1) \end{array}}{(\rho,\mu,\nu,\theta)\xrightarrow{t_1,t_2}\Gamma_{t_2}\circ\Gamma_{t_1}(\rho,\mu,\nu,\theta)}\ sync$$

### 4.3 Summary

While the Hierarchical Timed Automaton covers to a certain degree the three requirements temporal behavior, predictability and expressiveness a number of limitations remain: At first the approach is rather limited w.r.t. the data model. Only integer variables and their assignment are supported. Therefore, it is not possible to include complex data and operations even at the semantics level. The supported synchronous event model is also a rather strong limitation. Often asynchronous events rather than synchronous ones are used to decouple the detection and reaction on a specific external event. The formalism also does not support priorities for transitions. The only way to ensure that a certain precedence between two transitions activated at the same time holds will be the use of the urgency flag. However, usually if the first transition is not enabled we would expect the second one to fire immediately.

## 5 Extended Hierarchical Timed Automata

The most proposed modelling techniques focus either on structural expressiveness or temporal properties. The in the last section defined notion of Hierarchical Timed Automata addresses both aspects, however, a number of serious limitations and the realizability problem remain. We have to first take into account that the rest of the UML model and code builds a data model which is observed and modified with queries and updates, respectively. The provided synchronous events mechanism is also not sufficient in the long term and asynchronous events are additionally required. In practice different priorities rather than only urgent or non urgent transitions would permit to describe the precedence of transitions in a more appropriate manner.

### 5.1 Extensions

To overcome the presented limitations we propose a notion of an Extended Hierarchical Timed Automaton for modelling of complex real-time behavior by extending the Hierarchical Timed Automata model accordingly.

**Data Model** To integrate the UML data model with queries and updates within the formalism we extend the guards and resets with more generic elements for manipulation of a data model. When parallel access to the same part of the model occurs inconsistencies due to shared access may be possible. Therefore, we have to assume that the data model modifications are split into smaller steps which each guarantees not interference with other ones.

The intention behind such a more general treatment of the data model is, that such a formalization enables the following two strategies to incorporate the effects of complex data models:

1. Sophisticated data modelling approaches such as graph grammars may be employed to describe the corresponding semantics.

2. We may abstract from data model effects using non-determinism.

While we could fully replace the integer variables $V$, we better keep them to permit the explicit state space encoding by means of integer or enum types variables when appropriate.

**Asynchronous Events** To support asynchronous events in a systematic manner in addition to the synchronous channels also asynchronous ones are supported. We support the same style of scoping as for synchronous channels and realize a queue management that erases only events when explicitly consumed.

**Priorities** The support for urgent transitions does not permit to specify more detailed precedence information in case of multiple possible conflicting transitions. Therefore we replace the binary distinction between urgent and non urgent transitions and instead permit to specify priorities with natural numbers ($\mathbb{N}$). A priority 0 is identical to a not urgent transition in the Hierarchical Timed Automata model and for the case of synchronizing transitions $t_1$ and $t_2$ with priorities $p_1$ and $p_2$, we use $\min(p_1, p_2)$. To ensure fairness parallel transitions should not be effected by higher priority transitions in a parallel substate and therefore an enabled transition $t_1$ should only rule out another one ($t_2$) if both are conflicting transitions and $p_1 > p_2$.

## 5.2 Syntax

These extensions result in the following definition for an Extended Hierarchical Timed Automata.

**Definition 10** *The syntax of an* Extended Hierarchical Timed Automaton *is defined by a tuple* $\langle S, S_0, \delta, \sigma, D, V, C, Inv, Ch, T\rangle$ *where*

- $S$ *is a finite set of locations.* $root \in S$ *is the root.*
- $S_0 \in S$ *is a set of initial locations.*
- $\delta : S \to 2^S$. $\delta$ *maps* $l$ *to all possible substates of* $l$. $\delta$ *is required to give rise to a tree structure with root* $root$. *We readily extend* $\delta$ *to operate on sets of locations in the obvious way.*
- $\sigma : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ *is a type function on locations.*
- $D, V, C, Ch$ *are a data model, sets of variables, clocks, and channels. They give rise to* Guard, Reset, Sync, *and* Invariant *as defined in the following.*
- $Inv : S \to \text{Invariant}$ *maps every location* $l$ *to an invariant.*
- $T \subseteq S \times (Guard \times Sync \times Reset \times \mathbb{N}) \times S$ *is the set of transitions. A transition connects two locations* $l$ *and* $l'$, *has a guard* $g$, *a synchronization* $s$, *an assignment* $r$ *(including clock resets), and an priority* $p$. *We use the notation* $l \xrightarrow{g,s,r,p} l'$ *for this and omit* $g, s, r, p$ *when they are necessarily absent (or* $0$, *in the case of* $p$*).*

*The data components in guards, synchronizations, resets, and assignment expressions are defined:*

- $D$ *is a possibly infinite data model. A set of possible queries* $[D \to \{\mathsf{true}, \mathsf{false}\}]$ *and transformations* $[D \to D]$ *are assumed. To further take into account that data model updates or queries cannot be considered to be timeless and atomic their interference has to be controlled in an appropriate manner. Therefore, we assume a symmetric function* conflict *that for an update* $up$ *and an update or query* $up'$ *determines whether their concurrent execution can result in such interference* ($\text{conflict}(up, up') = \mathsf{true}$)*. A conflict can be excluded between a guard and an update only, when the update cannot effect the outcome of evaluating the guard. For two updates in contrast, basic concurrency techniques such as monitors or semaphores may be employed to ensure interference free execution which is modelled by different forms of interleaving.*

- *$V$ is the finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a superstate $l$.*

- *Let $C$ be a finite set of clock variables. The set $C(l) \subseteq C$ denotes the clocks local to a superstate $l$. If $l$ has a history entry, $C(l)$ contains only clocks, that are explicitly declared as forgetful. Other locally declared clocks of $l$ belong to $C(root)$.*

- *Let $Ch$ be a finite set of synchronization channels with subsets $Ch_s$ and $Ch_a$ for synchronous and asynchronous behavior. $Ch(l) \subseteq Ch$ is the set of channels that are local to a superstate $l$, i.e., there cannot be synchronization along a channel $c \in Ch(l)$ between one transition inside $l$ and one outside $l$. We further restrict the asynchronous communication channels to be only global ones.*

- *$Ch$ gives rise to a finite set of channel synchronizations, called* Sync. *For synchronous channels $c \in Ch_s$ ,$c?, c!, \tau \in Sync_s$. For $s \in Sync_s - \{\tau\}$, $\overline{s}$ denotes the matching complementary, i.e., $\overline{c!} = c?$ and $\overline{c?} = c!$. For events $c \in Ch_a$ we further can send and receive simultaneously on all asynchronous channels described by $Sync_a : [Ch_a \to I\!N] \times [Ch_a \to I\!N]$. We use the notation $(c_1?, c_2?, \ldots)$ respectively $(c_1!, c_2!, \ldots)$ to describe which events are received respectively send how often. A synchronization is then, for example, $(c?, (c_1?, c_2?, \ldots), (c_1!, c_2!, \ldots))$ and the overall set of synchronizations is build by* $\mathrm{Sync} := \mathrm{Sync}_s \times \mathrm{Sync}_a$ *which permits multiple asynchronous send and receives.*

- *The data model constraint is any boolean query $q_D \in [D \to \{\mathsf{true}, \mathsf{false}\}]$. A variable constraint is a boolean expressions of the form $A \Box A$, where $A$ is an arithmetic expression over $V$ and $\Box \in \{<, >, =, \leq, \geq\}$. A clock constraint is an expression of the form $x \Box n$ or $x - y \Box n$, where $x, y \in C$ and $n \in \mathbb{N}$ with $\Box \in \{<, >, =, \leq, \geq\}$. A clock constraint is downward closed, if $\Box \in \{<, =, \leq\}$. A guard is a finite conjunction over data model constraints, variable constraints and clock constraints. An invariant is a finite conjunction over downward closed clock constraints.* Guard *is the set of guards,* Invariant *is the set of invariants. Both contain additionally the constants* true *and* false *and therefore when no additional guard is required simple the constant* true *will be used.*

- *A clock reset is of the form $x := 0$, where $x \in C$. A data assignment is of the form $v := A$, where $v \in V$ and $A$ an arithmetic expression over $V$. Any* up $\in [D \to D]$ *is a data model update.* Reset *is the set of clock resets, data assignments, and data model updates.* $\mathrm{Reset}^-$ *is the set data assignments, and data model updates.*

For $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ we use the predicate notation $TYPE(l)$ for $l \in S$. E.g. $AND(l)$ is $\mathsf{true}$, exactly if $\sigma(l) = AND$. The type $HISTORY$ is a special case of an entry. We use $HENTRY(l)$ to capture simple entry or history entry, i. e., $HENTRY(l)$ stands for $ENTRY(l) \vee HISTORY(l)$.

For $l \neq$ root the parent function $\delta^{-1}(l)$ is defined as that unique $l'$ with $l \in \delta(l')$. To denote the set of all nested locations of a superstate $l$, including $l$, $\delta^*(l)$ is used. $\delta^{-*}$ is the set of ancestors of $l$, including $l$. Moreover we use $\delta^\times(l) = \delta^*(l) \setminus \{l\}$. We introduce $\widetilde{\delta}$ to refer to the children, that are proper locations: $\widetilde{\delta}(l) = \{n \in \delta(l) | BASIC(n) \vee XOR(n) \vee AND(n)\}$ . We use $V^+(l)$ to denote the variables in the scope of location $l$: $V^+(l) = \bigcup_{n \in \delta^{-*}(l)} V(l)$. $C^+(l)$ and $Ch^+(l)$ are defined analogously.

We further require the following rules to ensure consistency of a given Hierarchical Timed Automaton.

**Definition 11** *An Extended Hierarchical Timed Automaton is* well-formed *iff*

1. *The function $\delta$ has to give rise to a proper tree rooted at* root*, and $S = \delta^*(root)$. Basic nodes have to be empty ($BASIC(l) \Leftrightarrow \delta(l) = \emptyset$), and invariants of pseudo-locations are trivial ($HENTRY(l) \vee EXIT(l) \Rightarrow Inv(l) = \mathsf{true}$).*

   *Additionally, to have a well-formed hierarchical state we require for all non BASIC locations to hold that all substates are valid refinement ($\neg BASIC(l) \Leftrightarrow \forall l' \in \delta(l)(Inv(l') \Rightarrow Inv(l))$). Note, that for $Inv(l') = \mathsf{true}$ this condition does allways hold.*

2. *$S_0$ corresponds to a consistent and proper control situation, i.e. root $\in S_0$ and for every $l \in S_0$ the following holds that it is a proper location ($BASIC(l) \vee XOR(l) \vee AND(l)$), $S_0$ does contain all parents ($l = root \vee \delta^{-1}(l) \in S_0$), only one alternative of an XOR-state is present ($XOR(l) \Rightarrow |\delta(l) \cap S_0| = 1$), and all substates of an AND-state are proper states ($AND(l) \Rightarrow \delta(l) \cap S_0 = \widetilde{\delta}(l)$).*

3. *Conflicts in data model updates in synchronizing transitions and in general are disallowed by demanding that for any two locations $l_1$ and $l_2$ which can occur in parallel and any $l_1 \xrightarrow{g,c!,r,u} l_1'$ and $l_2 \xrightarrow{g',c?,r',u'} l_2'$ will hold $\neg\mathrm{conflict}(g,u')$, $\neg\mathrm{conflict}(u,g')$, and $\neg\mathrm{conflict}(u,u')$.*

4. *Conflicts in assignments in synchronizing transitions are explicitly excluded by demanding that $l_1 \xrightarrow{g,c!,r,u} l_1'$, $l_2 \xrightarrow{g',c?,r',u'} l_2' \in T$ follows $vars(r) \cap vars(r') = \emptyset$, where $vars(r)$ is the set of integer variables occurring in $r$. We require an analogous constraint to hold for the pseudo-transitions originating in the entry of an $AND$ superstate. For $l \xrightarrow{g,s,r,u} l' \in T, g, r$ has to be defined over $V^+(\delta^{-1}(l)) \cup C^+(\delta^{-1}(l))$ and $s$ is defined over $Ch^+(\delta^{-1}(l))$ to ensure a proper static scoping.*

5. *Let $e \in S, HENTRY(e)$. If $XOR(\delta^{-1}(l))$, then $T$ contains exactly one transition $e \xrightarrow{r} l'$. If $AND(\delta^{-1}(l))$, then $T$ contains exactly one transition $e \xrightarrow{r} e_i$ for every proper substate $l_i \in \widetilde{\delta}(\delta^{-1}(l))$, and $e_i \in \delta(l_i)$. In case of $HISTORY(e)$, outgoing transitions declare the default history locations. If a superstate $s$ has a history entry, then every substate $l$ of $s$ has to provide either a history entry or a default entry.*

6. *Transitions have to respect the structure given in $\delta$ and cannot cross levels in the hierarchy, except via connecting to entries or exits. Note that transitions cannot lead directly from entries to exits. Transitions $l \xrightarrow{g,s,r,u} l'$ with $HENTRY(l)$ or $EXIT(l')$ are called pseudo-transitions. They are restricted in the sense, that they cannot carry synchronizations or urgency flags, and only either guards or assignments. For $HENTRY(l)$, only pseudo-transition of the form $l \xrightarrow{r} l'$ are allowed. For $EXIT(l')$, only pseudo-transition of the form $l \xrightarrow{g} l'$ are allowed. For $EXIT(l) \wedge EXIT(l')$, this is further restricted to be of the form $l \rightarrow l'$.*

## 5.3 Semantics

The semantics of the Extended Hierarchical Timed Automaton model is further defined by using a configuration of the form $(\rho, \mu, \nu, \phi, \theta)$ which contains the active locations, the integer variable values, the clock values, asynchronous event queues, and the history.

- $\rho : S \rightarrow 2^S$ captures the control situation. $\rho$ can be understood as a partial, dynamic version of $\delta$, that maps every superstate $s$ to the set of active substates. If a superstate $s$ is not active, $\rho(s) = \emptyset$. We define $Active(l) = l \in \rho^{\times}(root)$, where $\rho^{\times}(l)$ is the set of all active substates of $l$. Note that $Active(l) \Leftrightarrow l \in \rho(\delta^{-1}(l))$.
- $\mu : S \rightarrow (\mathbb{Z})^*$. $\mu$ gives the valuation of the local integer variables of a superstate $l$ as a finite tuple of integer numbers. If $\neg Active(l)$ then $\mu(l) = \lambda$ (the empty tuple). If $Active(l)$ then we require that $|\mu(l)| = |V(l)|$ and $\mu$ are consistent with respect to the values of the

shared variables (i.e. always maps to the same value). We use $\mu(l)(a)$ to denote the value of $a \in V(l)$. When entering a non-basic location, local variables are added to $\mu$ and set to an initial value (0 by default). We use the shorthand $0^{V(l)}$f or the tuple (0, 0, ..., 0) with arity $|V(l)|$.

- $\nu : S \to (I\!R^+)^*$. $\nu$ gives the real valuation of the clocks $C(l)$ visible at location $l$, thus $|\nu(l)| = |C(l)|$. If $\neg Active(l)$ then $\nu(l) = \lambda$.
- $\phi : [Ch_a \to I\!N]$ reflects the asynchronous event channels, that have to be stored. Channels are assumed to be persistent and thus local channels are not cleared when leaving or entering a location. By default all events are also preserved until explicitly consumed.
- $\theta$ reflects the history, that might be restored by entering superstates via history entries. It is split up in the two functions $\theta_{state}$ and $\theta_{var}$, where $\theta_{state}(l)$ returns the last visited substate of $l$ – or an entry of the substates, in the case where the substate is not basic – (to restore $\rho(l)$), and $\theta_{var}(l)$ returns a vector of values for the local integer variables. There is no history for clocks at the semantics level, all non-forgetful clocks belong to $C(root)$.

To handle history entries the predicate $HasHistory(l) = \exists n \in \delta(l) | HISTORY(n)$ is used. If $HasHistory(l)$ holds, the term $HEntry(l)$ denotes the unique history entry of $l$. If $HasHistory(l)$ does not hold, the term $HEntry(l)$denotes the default entry of $l$. If $l$ is basic $HEntry(l) = l$. If none of the above is the case, then $HEntry(l)$ is undefined. Initially, it holds $\forall l \in S | HasHistory(l) \Rightarrow \theta_{state}(l) = HEntry(l) \wedge \theta_{var}(l) = 0^{V(l)}$.

In order to denote the set of locations reached by following a fork, the function $Targets_\theta : 2^S \to 2^S$ relative to $\theta$ is used.

$$Targets_\theta(l) = L \cup \bigcup_{l \in L} \left( \{n | n \in \theta_{state}(l) \wedge HISTORY(l)\} \cup \left\{ n | l \xrightarrow{r} n \wedge ENTRY(l) \right\} \right)$$

The notation $Targets_\theta(l)$ for $Targets_\theta(\{l\})$ is used, if the argument is a singleton. $Targets^*_\theta$ is the reflexive transitive closure of $Targets_\theta$.

Firing a transition $t : l \xrightarrow{g,s,r,u} l'$ entails in general 1. executing a join to exit $l$, 2. taking the proper transition $t$ itself, and 3. executing a fork at $l$. If $l$ (respectively $l'$) is a basic location, part 1. (respectively 3.) is trivial. Together, this defines a run-to-completion step. We represent a run-to-completion step formally by a transformation function $\Gamma_t$, which depends on a particular transition $t$. The three parts of this step are described as follows.

1. join: $(\rho, \mu, \nu, \phi, \theta)$ is transformed to $(\rho^1, \mu^1, \nu^1, \phi^1, \theta^1)$ as follows: $\rho$ is updated to $\rho^1 := \rho[\forall n \in \rho^\times(l) | n \to \emptyset]$. $\mu$ is updated to $\mu^1 := \mu[\forall n \in \rho^\times(l) | n \to \lambda]$. $\nu$ is updated to $\nu^1 := \nu[\forall n \in \rho^\times(l) | n \to \lambda]$. The asynchronous channels are simply preserved ($\phi^1 = \phi$). If $EXIT(l)$, the history is recorded. Let $H$ be the set of superstates $h \in \rho^\times(\delta^{-1}(l))$, where $HasHistory(h)$ holds. Then $\theta^1_{state} := \theta_{state}[\forall h \in H | h \to HEntry(\rho(h))]$ and $\theta^1_{var} := \theta_{var}[\forall h \in H | h \to \mu(h)]$. If $\neg EXIT(l)$ or $H = \emptyset$, then $\theta^1 := \theta$.
2. $(\rho^2, \mu^2, \nu^2, \phi^2, \theta^2) := (\rho^1[l'/l], r(\mu^1), r(\nu^1), \phi^1 - s_2 + s_3, \theta^1)$ is computed with $s = (s_1, s_2, s_3)$. $r(\mu^1)$ denotes the updated values of the integers after the assignments, $r(\nu^1)$ the updated clocks after the resets, and $\phi^1 - s_2 + s_3$ is computed by multi-set subtraction of the asynchronous receive part of $s$ and addition of the asynchronous send part of $s$.
3. fork: $(\rho^2, \mu^2, \nu^2, \phi^2, \theta^2)$ is transformed to $(\rho^3, \mu^3, \nu^3, \phi^3, \theta^3)$ by moving the control to all proper locations reached by the fork, i.e. those in $Targets^*_{\theta^2}(l')$. Note that $\rho^2(n) = \emptyset$ for all $n \in \delta^\times(l')$. Thus we can compute $\rho^3$ from $\rho^2$ by modifying it iteratively for all $n \in Targets^*_{\theta^2}(l')$ for $ENTRY(n)$ such that $\rho^3(\delta^{-2}(n)) := \rho^3(\delta^{-2}(n)) \cup \{\delta^{-1}(n)\}$ or

else $\rho^3(\delta^{-1}(n)) := \{n\}$. $\phi^3$ is identical to $\phi^2$ and $\mu^3$ is derived from $\mu^2$ by first initializing all local variables of the superstates $s$ in $Targets^*_{\theta^2}(l')$, i.e. $\mu^3(V(s)) := 0^{V(s)}$. If $HasHistory(s), \theta_{var}(s)$ is used instead of $0^{V(s)}$. Then all variable assignments and clock-resets along the pseudo-transitions belonging to this fork are executed to update $\mu^3$ and $\nu^3$. The history does not change $\theta^3$ identical to $\theta^3$.

Note that parts 1. and 3. correspond to the identity transformation, if $l$ and $l'$ are basic locations. The resulting configuration vector transformation $\Gamma_t$ for a transition $t : l \xrightarrow{g,s,r,u} l'$ is defined accordingly as the result of these three steps as $\Gamma_t(\rho, \mu, \nu, \theta) = (\rho^3, \mu^3, \nu^3, \phi^3, \theta^3)$. If the context is unambiguous, we use $\rho^{\Gamma_t}$ and $\nu^{\Gamma_t}$ for the parts $\rho^3$ respectively $\nu^3$ of the transformed configuration corresponding to transition $t$.

A superstate $s$ can only be exited, if all its parallel substates can synchronize on this exit. For an exit $l \in \delta(s)$ we recursively define the family of sets of exits $\mathrm{PreExitSets}(l)$. Each element $X$ of $\mathrm{PreExitSets}(l)$ is itself a set of exits. If transitions are enabled to all exits in $X$, then all substates can synchronize.

$$\mathrm{PreExitSets}(l) = \begin{cases} \left.\begin{array}{l} \bigcup_{n_1, \ldots, n_k} \oplus_{1 \le i \le k} \mathrm{PreExitSets}(n_i), \text{ where} \\ k = \left|\widetilde{\delta}\left(\delta^{-1}(l)\right)\right|, \{n_1, \ldots, n_k\} \subseteq \delta^\times\left(\delta^{-1}(l)\right), \\ \forall i.\mathrm{EXIT}(n_k) \wedge n_i \rightarrow l \in T \\ \delta^{-1}(\{n_1, \ldots, n_k\}) = \widetilde{\delta}(l) \end{array}\right\} \text{if } \mathsf{EXIT}(l) \wedge \mathsf{AND}\left(\delta^{-1}(l)\right) \\ \{\{l\}\} \cup \left(\bigcup_{m \in \delta(\delta^{-1}(l)) \wedge m \xrightarrow{g,r} l \in T} \mathrm{PreExitSets}(m)\right), \text{if } \mathsf{EXIT}(l) \wedge \mathsf{XOR}(\delta^{-1}(l)) \\ \{\}, \text{if } \mathsf{BASIC}(l) \end{cases}$$

Here, the operator $\oplus$ is a product over families of sets, i.e. it maps $(\{A_1, \ldots, A_a\}, \{B_1, \ldots, B_a\})$ to $\{A_1 \cup B_1, A_1 \cup B_2, \ldots, A_a \cup B_b\}$ and is extended to operate on an arbitrary finite number of arguments in the obvious way.

To give the transition system rules for $\longrightarrow$, predicates that evaluate conditions on the dynamic $\rho$ are further required. The set of active leaves (in the tree described by $\rho$), which are the innermost active states in a superstate $l$, are defined by: $Leaves(\rho, 1) = \{n \in \rho^\times(l) | \rho(n) = \emptyset\}$.

The predicate expressing that all the substates of a state $l$ can synchronize on a join is:

$\mathrm{JoinEnabled}(\rho, \mu, \nu, l) = \mathrm{BASIC}(l) \vee \exists X \in \mathrm{PreExitSets}(l) . \forall n \in \mathrm{Leaves}(\rho, l) : \exists n' \in X . n \xrightarrow{g} n' \wedge g(\mu, \nu)$

Note that JoinEnabled is trivially true for a basic location $l$.

The invariants of a location are denoted by $Inv_\nu : S \rightarrow \{\mathsf{true}, \mathsf{false}\}$, which evaluates the invariant of a given location with respect to a clock evaluation $\nu$ and the predicate $Inv(\rho, \nu)$ defined by $Inv(\rho, \nu) = \bigwedge_{n \in \rho^\times(root)} Inv_\nu(n)$ is used to express, that for control situation $\rho$ and clock valuation $\nu$ all invariants are satisfied. The predicate $Enabled$ over transitions $t : l \xrightarrow{g,s,r,u} l'$ is used to describe whether $t$ is enabled:

$$Enabled(t : l \xrightarrow{g, (s_1, s_2, s_3), r, u} l', \rho, \mu, \nu, \phi) := \begin{array}{l} \neg EXIT(l') \wedge g(\mu, \nu, \phi) \wedge (s_2 \le \phi) \wedge \\ JoinEnabled(\rho, \mu, \nu, l) \wedge Inv(\rho^{\Gamma_t}, \nu^{\Gamma_t}) \end{array} .$$

Since urgency has precedence over delay, we have to capture the global situation, where an urgent transition is enabled. We do this via the predicate UrgentEnabled over a configuration without taking the history into account.

$$\begin{aligned}\text{UrgentEnabled}(\rho,\mu,\nu,\phi) \;=\;& \big(\exists t : l \xrightarrow{g,r,p} l' \wedge p > 0 \wedge \mathit{Enabled}(t_1,\rho,\mu,\nu,\phi)\big) \bigvee \\ & \Big(\exists t_1 : l_1 \xrightarrow{g_1,(c!,S_1,S_2),r_1,p_1} l_1', t_2 : l_2 \xrightarrow{g_2,(c?,S_3,S_4),r_2,p_2} l_2' \wedge \\ & \mathit{Enabled}(t_1,\rho,\mu,\nu,\phi) \wedge \mathit{Enabled}(t_2,\rho,\mu,\nu,\phi) \wedge (p_1 > 0 \wedge p_2 > 0)\big)\end{aligned}$$

We further define for each priority $p$ its own transition relation $\longrightarrow_p$. For a transition $t \in T$ (action), the elapsing of time (delay), and synchronization (sync) we have:

$$\frac{\mathit{Enabled}(t : l \xrightarrow{g,(\tau,s_2,s_3),r,p} l', \rho,\mu,\nu,\phi)}{(\rho,\mu,\nu,\phi,\theta) \xrightarrow{t}_p \Gamma_t(\rho,\mu,\nu,\phi,\theta)}\ \mathit{action}$$

Here $g$ is the guard of the transition and $r$ the set of resets and assignments. The urgency flag $u$ has no effect here. This rule applies for action transitions between basic locations as well as superstates. In the latter case, this includes the appropriate joins and/or fork operations.

$$\frac{\forall d'\ 0 \le d' < d \ :\ \mathit{Inv}(l)(\rho,\nu+d') \quad \neg\text{UrgentEnabled}(\rho,\mu,\nu+d',\phi)}{(\rho,\mu,\nu,\phi,\theta) \xrightarrow{d}_0 (\rho,\mu,\nu+d,\phi,\theta)}\ \mathit{delay}$$

In the delay transition rule (delay) $\nu + d$ stands for the current clock assignment plus the delay for all the clocks. Time elapses in a configuration only when all invariants are satisfied and there is no urgent transition enabled. Note, that we also have to take possible time deltas $d'$ into account, to ensure that no urgent transition has become enabled before the time delta $d'$ has been elapsed.

$$\frac{\begin{array}{l}\mathit{Enabled}(t_1 : l_1 \xrightarrow{g_1,(c!,s_2,s_3),r_1,p_1} l_1', \rho,\mu,\nu,\phi) \quad l_1 \notin \delta^{\times}(l_2) \quad c \in Ch_s \\ \mathit{Enabled}(t_2 : l_2 \xrightarrow{g_2,(c?,s_5,s_6),r_2,p_2} l_2', \rho,\mu,\nu,\phi) \quad l_2 \notin \delta^{\times}(l_1)\end{array}}{(\rho,\mu,\nu,\phi,\theta) \xrightarrow{t_1,t_2}_{\min(p_1,p_2)} \Gamma_{t_2} \circ \Gamma_{t_1}(\rho,\mu,\nu,\phi,\theta)}\ \mathit{sync}$$

Since priority can result in precedence between conflicting transitions, we have to formalize what conflict means. Therefore, the structural relation between the source states reflected by their common root are important.

$$\mathit{CommonRoot}(l_1,l_2) := l_c \text{ with } l_c \in S' \wedge \delta(l_c) \notin S' \text{ for } S' = \delta^{-*}(l_1) \cap \delta^{-*}(l_2).$$

For all location pairs such a common root exists and the two related transitions are in conflict iff it is not a parallel AND-state that permits to execute them independently.

$$\mathit{ConflictingTransition}(t_1 : l_1 \xrightarrow{\vec{a}_1} l_1', t_2 : l_2 \xrightarrow{\vec{a}_2} l_2') := \mathit{AND}(\mathit{CommonRoot}(l_1,l_2)).$$

The notion of conflicting transitions can be further extended to all possible priority specific transition labels $\alpha, \beta \in T \cup T^2 \cup I\!R^+$. For $\mathit{Trans}(\alpha)$ defined as $\emptyset$ for $\alpha \in I\!N$, $\alpha$ for $\alpha \in T$, and $\{t_1,t_2\}$ for $\alpha = (t_1,t_2) \in T^2$ we have

$$\mathit{Conflicting}(\alpha,\beta) := \bigvee_{t_\alpha \in \mathit{Trans}(\alpha), t_\beta \in \mathit{Trans}(\beta)} \mathit{ConflictingTransition}(t_\alpha,t_\beta).$$

The resulting rule for the transition system steps $\longrightarrow$ for $\alpha$, $\beta$ elements of $T \cup T^2 \cup I\!R^+$ is then:

$$\frac{(\rho,\mu,\nu,\phi,\theta) \xrightarrow{\alpha}_{p_\alpha} (\rho',\mu',\nu',\phi',\theta') \quad \forall \beta, \Delta > 0 \ \Big((\rho,\mu,\nu,\phi,\theta) \xrightarrow{\beta}_{p_{\alpha+\Delta}} \Rightarrow \neg\text{Confliciting}(\alpha,\beta)\Big)}{(\rho,\mu,\nu,\phi,\theta) \xrightarrow{\alpha} (\rho',\mu',\nu',\phi',\theta')}\ \mathit{prio}.$$

### 5.4 Summary

The model of the Extended Hierarchical Timed Automata covers –like the Hierarchical Timed Automata– the requirements temporal behavior, predictability and expressiveness. Furthermore the data model has been extended, so that it is possible to model complex data operations. In addition Extended Hierarchical Timed Automata support synchronous events as well as asynchronous and associate priorities with transitions in order to achieve determinism on the modelling level.

Thus Extended Hierarchical Timed Automata allow the modeling of different kinds of behaviors. So it is possible to specify limited behavior as well as unlimited behavior (cf. Section 2.1). To ensure, that a specification is only limited and thus can be implemented on a real physical controller, another model is required. Such a model is presented in the next section.

## 6 Real-Time Statecharts

In [Bur02] a Statechart notion named *Real-Time Statecharts* has been developed, which permits to model only limited behavior as defined in Section 2.1. Also code generation for the Java Real-Time platform has been realized as an extension of the Fujaba UML CASE tool.[4]

### 6.1 Extension

In order to specify complex temporal behavior, the advantages of modelling complex behavior with Statecharts is combined with the advantage of specifying temporal behavior with ExHTA, resulting in the real-time extension of Statecharts.

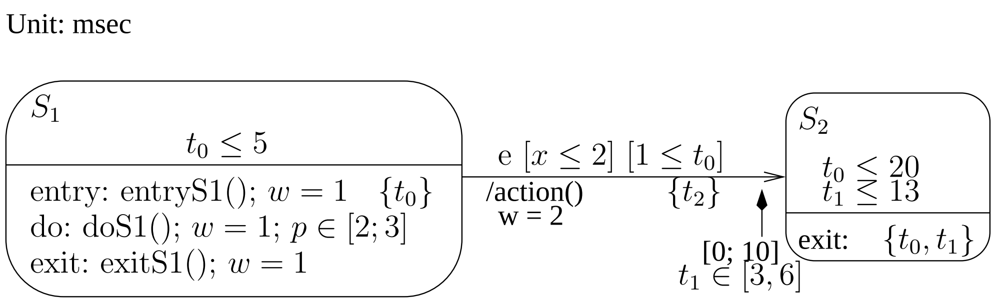


Figure 1: Real-Time Statechart

Figure 1 depicts a (part of a) Real-Time Statechart, consisting of 2 states and 1 transition. The state $S_1$ has the time invariant $t_0 \leq 5msec$, $S_2$'s invariant is $t_0 \leq 20msec \wedge t_1 \leq 13msec$ ($t_0$ and $t_1$ denote clocks). The transition is triggered by the event `e`, the guard $[x \leq 2]$ and by the time guard $[1msec \leq t_0]$. When firing $t_2$ is reset and `action()`, which has the WCET $w = 2msec$, is executed. The firing of the transition has to be finished at the latest $10msec$ after being triggered *and* at $t_1 == 6$. The earliest point in time, when $S_2$ may be entered is at $t_1 == 3$. Worst-case execution times are attached to the `entry()`-, `exit()`- and `do()`-operations, too, clock can be reset when entering and exiting the states, too. As the `do()`-operation is executed periodically, an interval is specified from where a fixed period is chosen.

Giving a compact overview, Real-Time Statecharts offer the following constructs for states: Hi-

[4]www.fujaba.de

erarchy, parallelism, and time invariants. Transitions make use of the constructs: Events, guards, time guards, resets, priorities, and synchronization.

It is shown, that they can be mapped to a corresponding ExHTA. The semantics of the Real-Time Statecharts is defined by these mapping rules.

## 6.2 Semantics

Before outlining the semantics by showing the mapping rules, we will introduce an abbreviation. Figure 2 a) shows an ExHTA with an urgent transition from location $S_1$ to $S_2$ and a non-urgent transition from $S_2$ to $S_3$. $S_2$ is a hierarchical state with the invariant $t_0 \leq 10$ and the substates $S_2^1$, $S_2^2$ and $S_2^3$ which have the same invariant (in this example); $action_1()$ and $action_2()$ are resets. Such a hierarchical state can be abbreviated as shown in Figure 2 b). There, the sequential executed resets $action_1()$ and $action_2()$ are combined to $action()$ and the substates disappear.

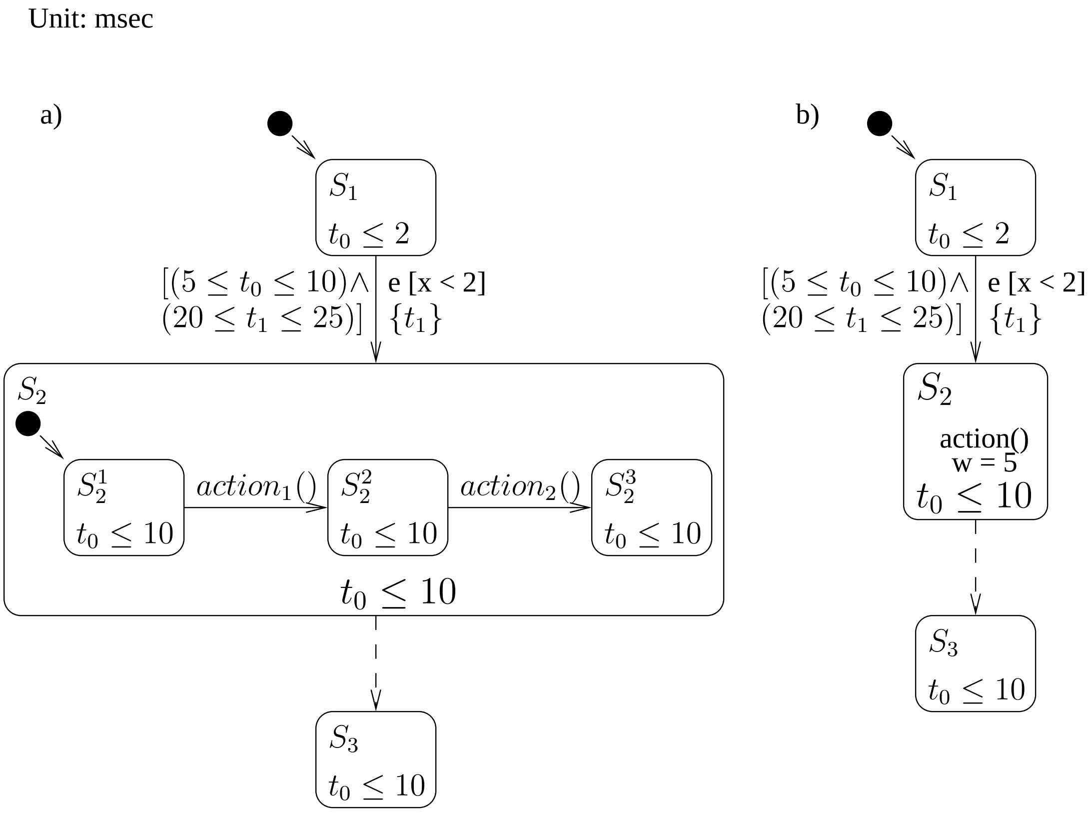


Figure 2: Extended Hierarchical Timed Automata

Using this abbreviation, a Real-Time Statechart can be mapped to an ExHTA –as depicted in Figures 3 and 4. Figure 3 shows how non-hierarchical states are mapped to locations: The name as well as the invariant are held. The `entry()`- and `exit()`-operations and the sets of clock-resets will be rolled out of the states. To modell the periodic execution of the `do()`-operation, a transition with its specific state as source and target will be created. This transition resets the new introduced clock $t_{int}$ and is only triggered at $t_{int} == p$, where $p \in [p_{low}, p_{up}]$.

When firing a transition, the `exit()`-operation of the source state, the data assignment and the target state's `entry()`-method are executed sequentially. This is modelled in the ExHTA by putting

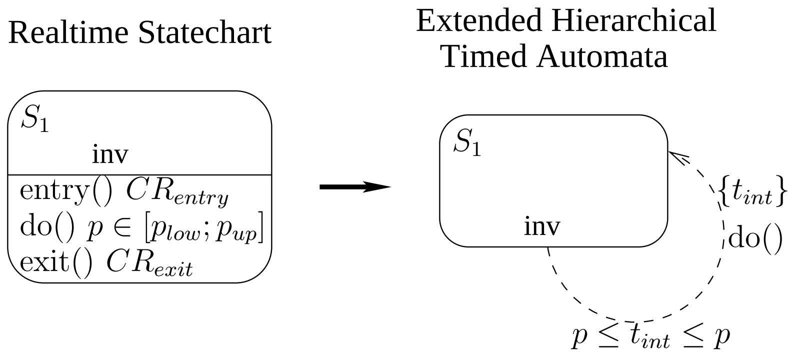


Figure 3: Mapping of non-hierarchical states

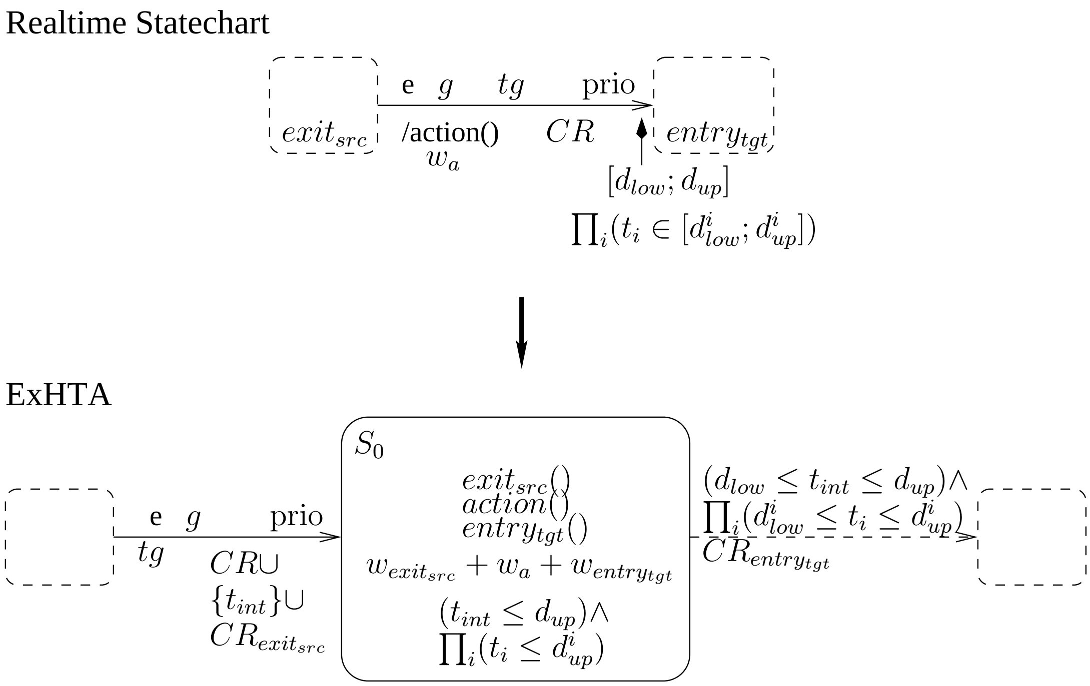


Figure 4: Mapping of transitions

them in a hierarchical state –similar to Figure 2– and using the abbreviation, introduced above. This leads to 2 different kinds of locations: *Statelocations*, representing states and *actionlocations* in which operations are executed. The WCET of these 3 actions is the sum of their single WCETs. In Figure 4 is shown, that a transition is mapped to an actionlocation and 2 transitions (an urgent and a non-urgent one), connecting it with the source and the target state. The event, guard, timeguard and priority from the Real-Time Statechart is recovered in the urgent transition as these attributes shall trigger the actions. The deadline is recovered at multiple places. On the one hand it is used to extend the invariant of the actionlocation, on the other hand it is used to prevent the actionlocation to be left before it may be left, specified by the lower bound of the deadline. As the last point of time, when the actionlocation may be left, is specified by the upper bound of the deadline, it is used for the upper bound of the timeguard of the non-urgent transition.

The constructs of parallelity, hierarchie and history can be adopted without changes, because Real-Time Statecharts offer them as well as ExHTA.

## 6.3 Realizability

To achieve a mapping from the model to real physical threads, deadlines and WCETs are required for every transition. These deadlines specify the point in time, when the execution has to be ended relative to the moment of triggering or relative to one or multiple clocks. Thus, the modeller has to respect, that the firing of a transition (respectively the execution of resets and data assignments) is not infinitely fast and needs time to be executed. Usually, the WCETs are dependant on the target platform. Note that the WCET of an operation needs to be less than its deadline. The timedifference between WCET and deadline respects that other tasks require resources simultaneously and that on real physical devices a triggered transition is usually recognized with a delay. This delay is compensated by that timedifference. Thus, the designation of WCETs and deadlines permits the mapping of the model to real physical threads that can be executed on a (limited) physical digital controller. These threads can be implemented in every programming language, supporting real-time functionality. The CASE-Tool Fujaba[5] delivers an implementation of Real-Time Statecharts for Real-Time Java.

When multiple Real-Time Statecharts are executed in parallel on the same processor, WCETs and deadlines are used to perform a schedulability analysis before code is generated.

The specification of invariants, deadlines, guards and WCETs potentiates to model time inconsistencies, e.g. an invariant of a state expires before any transition, leaving from this state, is triggered. By analysing the structure of the Real-Time Statechart and the coherency of the temporal annotations, some conflicts and inconsistencies can be detected with an appropriate complexity (e.g. no state space exploration, usually caused by model checking tools). For cases where this analysis is not capable to determine whether a timing fault is present or not, model checking should be employed.

## 6.4 Summary

With the model of Real-Time Statecharts it is possible to specify complex behavior by the use of hierarchy and parallelism as well as temporal behavior. Additionally, it is possible to specify operations executed at the moment of entrance or exit of a state (`entry()`-/`exit()`-operation) or being periodically invoked during residence in a state (`do()`-operation). Real-Time Statecharts fulfill all requirements pointed out in Section 2 and ensure that only limited behavior is specified that is realizable.

[5] www.fujaba.de

# 7 Related Work

Harel introduced in [Har98] 2 extensions of Statecharts to bring in time constraints: Using the *timeout*-construct an event $e$ and a time $t$ is associated with a transition. The transition is triggered $t$ time-steps after the occurence of the event $e$. If a reset or data assignment is marked as *scheduled action*, it is not executed during firing a transition, but a specified time later (although the state is changed immediately). In [KP92] an extension is described, that allows to specify a kind of delay interval for a transition. The bounds of this interval define how much time elapses at least and at most between triggering and firing the transition. In [PAS94] a similar approach is used. There, the interval describes the points in time, when the source state can be left earliest and when it has to be left at the latest. In addition, every event obtains a duration, indicating how long it can be consumed after creation. All the constructs, described above, are useful to specify temporal behavior for unlimited systems. To map them on man made controllers, more informations are needed (e.g. WCETs, deadlines, etc.).

In [SKW00] the model of an automaton is mapped to a so called *multi-tasking model*. The multi-tasking model describes how the actions of the model are placed in physical threads. The annotation of WCETs and deadlines in the model makes this mapping possible. The automaton offers communication via synchronous and asynchronous events. Compared to high-level automata, it is restricted, as it is not possible to use hierarchy, parallelism or to trigger transitions dependant on clocks or variables.

[LQV01] deals with validation of real-time models. UML (in particular class-diagrams and extended Statecharts) is used to specify the structure and behavior, based on time. As no WCETs and deadlines are required and time comes in just by the use of the after-construct, the problems mentioned above, occure here, too. The claimed invariants of the system behavior are expressed by OCL constraints. The specified model can be translated into the first order temporal logic language *TRIO*, that can be used as input for the modelchecker TRIO-Matic in order to validate the properties of the model. To formalize the semantics of the extended Statechart model, it is mapped to an extension of Timed Statecharts in [BLM02].

In [LC96] *Duration Calculus* –an extended first order logic language– is used to extend Statecharts with duration. Thus, there exist a couple of functions determining if and for which time states are active within a specific timeinterval. This framework, expressing temporal requirements as functions, was regarded as breakthrough in validation of not only functional but also temporal correctness. Another approach for verifying real-time systems is given in [KMR02], where behavior is specified by UML Timed State Machines, that are mapped to Timed Automata, serving as input for the Modelchecker *UPPAAL* [DMY02]. Also in this work, it is assumed that the Timed State Machine runs on an unlimitted, infinite fast device. Anyhow, it is assumed, that the Statechart is distributed and it is possible to specify an upper bound for the delay of communication. In addition to the Timed State Machine, so called *collaborations* are specified by the use of sequence diagrams. These sequence diagrams are transformed to the UPPAAL conformant *Observer Timed Automata*. After that UPPAAL is used to verify the Timed Automaton against the Observer Timed Automaton.

As model checking requires a formal semantics, that is not provided by UML, [EW00] delivers another formal semantics for Statecharts at the requirements-level, orientated at the StateMate semantics. In the paper, a comparison is given of structured and object oriented Statechart semantics as well as a comparison of Statecharts at the requirements- and at the implementation-level. The main aspect describes the communication between several Statecharts.

All this work and the extensions are useful for verification and validation of systems, whose behavior is based on time. As none of the models have been designed with the aim of generating code, they do not provide solutions appearing in this domain.

# 8 Conclusion

For the proposed notion of Extended Hierarchical Timed Automata the required modeling notation properties expressiveness, temporal behavior and predictability follows from the extension of the hierarchical Timed Automata concepts. While the proposed extension of priorities limit at least the ability to employ model checking the resulting extended hierarchical state machines, integrating an arbitrary complex data, permit to even check an finite abstraction of real models.

The presented formal extension of the hierarchical Timed Automata has been employed in order to define the semantics for Real-Time Statecharts [Bur02]. Each Real-Time Statechart is mapped to a limited Extended Hierarchical Timed Automaton which is bi-partite with respect to do and activity states. The realizability of consistent models of this limited notion of Extended Hierarchical Timed Automata has been shown in Section 6. The presented notion therefore supports all four identified requirements and thus better serves the required modeling of complex reactive real-time systems.

## Acknowledgements

The authors thank Martin Hirsch and Matthias Meyer for comments on earlier versions of the report.

## References

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for Real-Time Systems. In *Proc. of Logic in Computer Science*, pages 414–425. IEEE Press, June 1990.

[AD90] Rajeev Alur and David Dill. Automata for Modelling Real-Time Systems. In *Proc. of the International Colloquium on Algorithms, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Verlag, July 1990.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AK96] R. Alur and R. P. Kurshan. Timing Analysis in COSPAN. In R. Alur, T. A. Heizinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 220–231. Springer Verlag, 1996.

[BD00] Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Information Processing Letters*, 75(1-2):1–7, July 2000.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[BLM02] V. Del Bianco, L. Lavazza, and M. Mauri. A Formalization of UML Statecharts for Real-Time Software Modelling. 2002.

[BR99] Loic P. Briand and Daniel M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach.* IEEE Press, 1999.

[Bro86] Manfred Broy. A theory of nondeterminism, parallelism, communication and concurrency. *Theoretical Computer Science*, 45, 1986.

[Büc62] J. R. Büchi. On a decission method in restricted second order arithmetic. In *Proc. of the 1960 Int. Congt. on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[Bur02] Sven Burmester. *Generierung von Java Real-Time Code für zeitbehaftete UML Modelle*. 2002.

[BW01] Alan Burns and Andy Wellings. *Real-Timer Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 3rd edition, 2001.

[DH94] D. Drusinsky and D. Harel. On the Power of Bounded Concurrency I: Finite Automata. *Journal of the ACM*, 41(3), 1994.

[DMY02] Alexandre David, Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-time Extensions. In Ralf-Detler Kutsche and Herbert Weber, editors, *Proceedings of FASE 2002*, number 2306 in Lecture Notes in Computer Science, pages 218–232. Springer Verlag, 2002.

[DOTY96] C. Daws, A. Oliverio, S. Tripakis, and S. Yovine. The Tool KRONOS. In R. Alur, T. A. Heizinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer Verlag, 1996.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirat timed automata with KRONOS. In *Proc. of the 16th IEEE Real-Time Symposium*. IEEE Press, December 1995.

[EMSS90] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings of the Second Workshop on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 136–145, 1990.

[EW00] R. Eshuis and R. Wieringa. Requirements-level semantics for uml statecharts. In *S.F. Smith and C.L. Talcott, editors, Formal Methods for Open Object-Based Distributed Systems IV (FMOODS 2000), pages 121 140. Kluwer Academic Publishers.* Kluwer Academic Publishers, 2000.

[Har87] D. Harel. STATECHARTS: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[Har98] D. Harel. *Modeling reactive systems with statecharts*. McGraw-Hill, 1998.

[HHWT95a] T. A. Heizinger, P.-H. Ho, and H. Wong-Tai. A Userguide to HYTECH. In E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algortihms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer Verlag, 1995.

[HHWT95b] Thomas A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The Next Generation. In *Proc. of the 16th IEEE Real-Time Symposium*. IEEE Press, December 1995.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symposium on Logic in Computer Science*. IEEE Press, 1992.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Series in Computer Science. Addison-Wesley, Reading, MA, USA, 1979.

[Jos96] Mathai Joseph. *Real time systems : specification verification and analysis.* Prentice Hall international series in computer science. Prentice Hall, 1996.

[KMR02] A. Knapp, S. Merz, and C. Rauh. *Model Checking timed UML State Machines and Collaborations.* 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, Lecture Notes in Computer Science volume 2469 pages 395-414. Springer-Verlag, 2002.

[KP92] Y. Kesten and A. Pnueli. Timed and Hybrid Statecharts and their Textual Representation. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[LC96] K. Leung and D. Chan. Extending Statecharts with Duration. In *20th International Computer Software Applications Conference*, volume 2306 of *IEEE Press*, 1996.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology*, 1(1), 1997.

[LQV01] L. Lavazza, G. Quaroni, and M. Venturelli. Combining UML and Formal Notations for Modelling Real-Time Systems. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Vienna Austria*, September 2001.

[LS01] G. Logothetis and K. Schneider. Symbolic Model Checking of Real-Time Systems. In *International Symposium on Temporal Representation and Reasoning*, pages 214–223, Cividale del Friuli, Italy, June 2001.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Obj01] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4*, September 2001. OMG document ad/01-09-67.

[PAS94] A. Peron and A.Maggiolo-Schettini. Transitions as Interrupts: A New Semantics for Timed Statecharts. In *Proceedings of TACS '94*, volume 789 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice.* Unpublished, Department of Computer Science, Uppsala University, Denmark, February 1999.

[PL00] Paul Pettersson and Kim G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Press, November 1977.

[Pur99] Anuj Puri. An Undecidable Problem for Timed Automata. *Discrete Event Dynamic Systems*, 9(2), May 1999.

[RK99] J. Ruf and T. Kropf. Modeling Real-Time Systems with I/O-Interval Structures. In *GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 91–100. Shaker Verlag, March 1999.

[RK00] J. Ruf and T. Kropf. Analyzing Real-Time Systems. In *Design, Automation and Test in Europ (DATE)*, Paris France, March 2000. IEEE Computer Press.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic Synthesis of Multi-Tasking Implementations from Real-Time Object-Oriented Models. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport Beach, California, 2000.

[WM97] Michal Walicki and Sigurd Meldal. Algebraic Approaches to Nondeterminism—an Overview. *ACM Computing Surveys*, 29(1):30–81, March 1997.