Formal Verification of UML Statecharts with Real-Time Extensions*

Alexandre David[1], M. Oliver Möller[2], and Wang Yi[1]

[1] Department of Information Technology, Uppsala University,
{adavid,yi}@docs.uu.se,
[2] BRICS Basic Research in Computer Science, Aarhus University,
omoeller@brics.dk.

Abstract. We present a framework for formal verification of a real-time extension of UML statecharts. For clarity, we restrict ourselves to a reasonable subset of the rich UML statechart model and extend this with real-time constructs (clocks, timed guards, and invariants). We equip the obtained formalism, called *hierarchical timed automata* (HTA), with an operational semantics.
We outline a translation of one HTA to a network of flat timed automata, that can serve as input to the real-time model checking tool Uppaal. This translation can be used to faithfully verify deadlock-freedom, safety, and unbounded response properties of the HTA model. We report on an XML-based implementation of this translation, use the well-known pacemaker example to illustrate our technique, and report run-time data for the formal verification part.

1 Introduction

Computer-dependent systems are experiencing an enormous increase in complexity. Maintaining consistency and compatibility in the development process of industrial-sized systems makes it necessary to describe systems on various levels of detail in a coherent way. Modern software engineering answers the challenge with powerful modeling paradigms and expressive yet abstract formalisms. Object orientation concepts provide—among many other features—a consistent methodology to abstract away from implementation details and achieve a high level view of a system.

Modeling languages, like UML, go a step further. They describe high-level structure and behavior, rather than implementations of solutions. Thus they help organizing design and specifications in different views of a system, meeting the needs of developers, customers, and implementors. In particular, they capture a notion of *correctness*, in terms of requirements the system has to meet. Formal methods typically address *model correctness*, for they operate on a purely mathematical formalization of the model. This makes it possible to prevent errors inexpensively at early design stages.

* Supported by the European AIT-WOODDES project, No IST-1999-10069.

For real-time systems correctness does not only depend on functionality but also timeliness. This adds another dimension of complexity and make early validation an even more crucial step. Industrial CASE tools, e.g., VisualState™ [19], exemplify how implementations benefit from high level analysis. One particularly interesting part of a complete model is the behavioral view, since it captures the dynamics of a system. The action and inter-action of components is often non trivial. Therefore a variety of formalisms allow *execution* of the model, that unfolds and visualizes system behavior.

The UML statechart formalism focuses on the control aspect, where event communication and data determines possible sequences of states. Often the behavior is dependent on real-time properties [5] and is therefore supported by industrial tools like Rhapsody [18, 6]. The generated traces of the system model can be validated to coincide with the intuitive understanding of the system. However, we feel that in order to talk about correctness of a system the notion of a *formal requirement* is needed, that is either fulfilled or violated.

High-level requirements have to be communicated among collaborators with often very non-homogeneous backgrounds. It is desirable to express requirements in a simple yet powerful language with a clearly defined meaning. In this paper we use a formal *logical* language for this purpose, equipped with constructs to express real-time properties, namely *timed computation tree logic* TCTL [8]. Logically expressed properties are completely unambiguous, and automated validation and verification is possible for a reasonable class of systems. If the system does not satisfy a required logical formula, this reflects a design flaw.

In addition, it is necessary to establish sanity properties of the model, like deadlock freedom. If a behavioral model can enter a deadlock state, where no further changes are possible, the behavior of an implementation is typically (flawfully) unspecified. Simulators, e.g., ObjectGeode [16], can execute behavioral descriptions and can help to *validate* systems, i.e., discover design flaws, if they occur in a simulation session. Similar to testing, simulators cannot show the absence of errors. In contrast, *formal verification* establishes correctness by mathematical proof. If a model satisfies a property, there is no way to misbehave, at least not for the model.

Properties only carry over safely to the implementation under certain assumptions, e.g., that a local hardware bus can be accessed in below 2μs. These values can often be included as parameters.

Related Work. Statecharts have been analyzed by means of model-checking earlier. In [17] a formal semantics in terms of clocked transition systems is given, that allows to benefit from the analysis tools developed for this formalism. However, this work treats time in a discrete lock-step fashion.

In [13] a formalization of UML statecharts is presented. The formalization is given in terms of an operational semantics and is implemented in the vUML tool that uses the model-checker SPIN [9]. However the timing aspects are not treated in this approach.

In contrast, we propose dense time extensions of statecharts for formal verification purposes. As a prerequisite, we give formal syntax and semantics. Then we sketch a translation of our (hierarchical) formalism into a parallel composition of timed automata, that serve as input to the Uppaal verification tool. We establish deadlock-freedom and TCTL safety and (unbounded) response properties of a pacemaker model. The detailed version of the paper is found in [4].

Organization. Section 2 gives the formal syntax of our statechart restriction, extended with real-time constructs. Section 3 contains the formal semantics. In Section 4 we sketch a translation of our formalism to the Uppaal tool. Section 5 reports on formal verification of the pacemaker example and gives run-time data for the tool executions. Section 6 summarizes and outlines further work.

2 Hierarchical Timed Automata

In this section we define the formal syntax of hierarchical timed automata. This is split up in the data parts, the structural parts, and a set of well-formedness constraints. Before we present the formal syntax we introduce some restrictions on the UML statecharts.

2.1 A Restricted Statechart Formalism

In this paper we address the formal verification of a restricted version of the UML statechart formalism. We add formal clocks in order to model timed behavior.

Unlike in the UML, where statecharts give rise to the incarnation of objects, we treat a statechart itself as a behavioral entity. The notion of thread execution is simplified to the parallel composition of state machines. Relationships to other UML diagrams are dropped.

Our formalism does not support exotic modeling constructs, like synchronization states. Some UML tools allow to use C++ as an action language, i.e., C++ code can be arbitrarily added to transitions. Formal verification of this is out of scope of this work, we restrict ourselves to primitive functions and basic variable assignments. Event communication is simplified to the case, where two parts of the system synchronize via handshake.

Some of the restrictions we make can be relaxed, as explained in the Future Work Section 6. What we preserve is the essence of the statechart formalism: hierarchical structure, parallel composition at any level, synchronization of remote parts, and history.

2.2 Data Components

We introduce the data components of hierarchical timed automata, that are used in guards, synchronizations, resets, and assignment expressions. Some of this data is kept local to a generic super-state, denoted by l. A *super-state* is a state containing other states.

Integer variables. Let V be a finite set of integer variables. We later define their scope locally.

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

Channels. Let Ch a finite set of synchronization channels. $Ch(l) \subseteq Ch$ is the set of channels that are local to a super-state l, i.e., there cannot be synchronization along a channel $c \in Ch(l)$ between one transition inside l and one outside l.

Synchronizations. Ch gives rise to a finite set of channel synchronizations, called $Sync$. For $c \in Ch$, $c?, c! \in Sync$.

Guards and invariants. A data constraints is a boolean expression of the form $A \sim A$, where A is an arithmetic expression over V and $\sim \in \{<, >, =, \leq, \geq\}$. A clock constraint is an expression of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$ and $n \in \mathbb{N}$ with $\sim \in \{<, >, =, \leq, \geq\}$. A clock constraint is downward closed, if $\sim \in \{<, =, \leq\}$. A guard is a finite conjunction over data 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**.

Assignments. A clock reset is of the form $x := 0$, where $x \in \mathcal{C}$. A data assignment is of the form $v := A$, where $v \in V$ and A an arithmetic expression over V. *Reset* is the power set of clock resets and data assignments.

2.3 Structural Components

We give now the formal definition of our hierarchical timed automaton.

Def 1 *A hierarchical timed automaton is a tuple* $\langle S, S_0, \delta, \sigma, V, \mathcal{C}, Ch, \mathrm{Inv}, T\rangle$ *where*

- *S is a finite set of locations. root $\in S$ is the root.*
- *$S_0 \in S$ is a set of initial locations.*
- *$\delta : S \to 2^S$. δ maps l to all possible sub-states of l. δ is required to give rise to a tree structure. We readily extend δ to operate on sets of locations in the obvious way. If $\delta(l) \neq \varnothing$, then l is called a* super-state*.*
- *$\sigma : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ is the type function for locations.*
- *$V, \mathcal{C}, Ch$ are sets of variables, clocks, and channels. They give rise to Guard, Reset, Sync, and Invariant as described in Section 2.2.*
- *$\mathrm{Inv} : S \to$ Invariant maps every locations l to an invariant expression, possibly to the constant* **true***.*
- *$T \subseteq S \times (Guard \times (Sync \cup \{\varnothing\}) \times Reset \times \{\mathbf{true}, \mathbf{false}\}) \times S$ is the set of transitions. A transition connects two locations l and l', has a guard g, (optionally) a synchronization s, an assignment r (including clock resets), and an urgency flag u. We use the notation $l \xrightarrow{g,s,r,u} l'$ for this and omit g, s, r, u, when they are necessarily absent (or* **false***, in the case of u).*

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

We define the parent function

$$\delta^{-1}(l) \overset{def}{=} \begin{cases} n, \text{ where } \ l \in \delta(n) \text{ if } l \neq \mathit{root} \\ \varnothing \qquad\qquad\qquad\qquad \text{otherwise.} \end{cases}$$

We use $\delta^*(l)$ to denote the set of all nested locations of a super-state l, including l. δ^{-*} is the set of all ancestors of l, including l. Moreover we use $\delta^\times(l) \overset{def}{=} \delta^*(l) \backslash \{l\}$. We introduce $\tilde{\delta}$ to refer to the children, that are proper locations.

$$\tilde{\delta}(l) \overset{def}{=} \{n \in \delta(l) \mid \mathit{BASIC}(n) \vee \mathit{XOR}(n) \vee \mathit{AND}(n)\}$$

We use $V^+(l)$ to denote the variables in the scope of location l: $V^+(l) = \bigcup_{n \in \delta^{-*}(l)} V(n)$. $\mathcal{C}^+(l)$ and $\mathit{Ch}^+(l)$ are defined analogously.

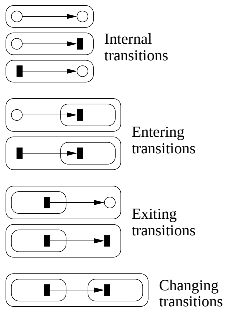
2.4 Well-Formedness Constraints

We give only the major well-formedness constraints to ensure consistency, grouped according to the synctactic categories variables, entries, and transitions.

Variable constraints. We explicitly disallow conflict in assignments in synchronizing transitions:
It holds that $l_1 \xrightarrow{g,c!,r,u} l_1',\ l_2 \xrightarrow{g',c?,r',u'} l_2' \in T \Rightarrow \mathit{vars}(r) \cap \mathit{vars}(r') = \varnothing$, where $\mathit{vars}(r)$ 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* super-state. Static scope: For $l \xrightarrow{g,s,r,u} l' \in T$, g, r are defined over $V^+(\delta^{-1}(l)) \cup \mathcal{C}^+(\delta^{-1}(l))$ and s is defined over $\mathit{Ch}^+(\delta^{-1}(l))$.

Entry constraints. Let $e \in S$, $\mathit{HENTRY}(e)$. If $\mathit{XOR}(\delta^{-1}(l))$, then T contains exactly one transition $e \xrightarrow{r} l'$. If $\mathit{AND}(\delta^{-1}(l))$, then T contains exactly one transition $e \xrightarrow{r} e_i$ for every proper sub-state $l_i \in \tilde{\delta}(\delta^{-1}(l))$, and $e_i \in \delta(l_i)$.

Transition constraints. Transitions have to respect the structure given in δ and cannot cross levels in the hierarchy, except via connecting to entries or exits. The set of legal transitions is given in Table 1. Transitions $l \xrightarrow{g,s,r,u} l'$ with $\mathit{HENTRY}(l)$ or $\mathit{EXIT}(l')$ 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 $\mathit{HENTRY}(l)$, only pseudo-transitions of the form $l \xrightarrow{r} l'$ are allowed. For $\mathit{EXIT}(l')$, only pseudo-transition of the form $l \xrightarrow{g} l'$ are allowed. For $\mathit{EXIT}(l) \wedge \mathit{EXIT}(l')$, this is further restricted to be of the form $l \rightarrow l'$.



Comment	l	l'	Constraint
	BASIC	*BASIC*	
Internal	*BASIC*	*EXIT*	$\delta^{-1}(l) = \delta^{-1}(l')$
	HENTRY	*BASIC*	
Entering	*BASIC*	*HENTRY*	
and fork	*HENTRY*	*HENTRY*	$\delta^{-1}(l) = \delta^{-2}(l')$
Exiting	*EXIT*	*BASIC*(l)	
and join	*EXIT*	*EXIT*	$\delta^{-2}(l) = \delta^{-1}(l')$
Changing	*EXIT*	*HENTRY*	$\delta^{-2}(l) = \delta^{-2}(l')$

Table 1. Overview over all legal transitions $l \xrightarrow{g,s,r,u} l'$.

3 Operational Semantics of HTAs

We present the operational semantics of our hierarchical timed automaton model. A *configuration* captures a snapshot of the system, i.e., the active locations, the integer variable values, the clock values, and the history of some super-states. Configurations are of the form (ρ, μ, ν, θ), where

– $\rho : S \to 2^S$ captures the control situation. ρ can be understood as a partial, dynamic version of δ, that maps every super-state s to the set of active sub-states. If a super-state s is not active, $\rho(s) = \varnothing$. We define $\mathit{Active}(l) \overset{def}{=} l \in \rho^{\times}(root)$, where $\rho^{\times}(l)$ is the set of all active sub-states of l. Notice that $\mathit{Active}(l) \Leftrightarrow l \in \rho(\delta^{-1}(l))$.
– $\mu : S \to (\mathbb{Z})^*$. μ gives the valuation of the local integer variables of a super-state l as a finite tuple of integer numbers. If $\neg\mathit{Active}(l)$ then $\mu(l) = \lambda$ (the empty tuple). If $\mathit{Active}(l)$ then we require that $|\mu(l)| = |V(l)|$ and μ is consistent with respect to the value of 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 μ and set to an initial value (0 by default). We use the shorthand $0^{V(l)}$ for the tuple $(0, 0 \ldots 0)$ with arity $|V(l)|$.
– $\nu : S \to (\mathbb{R}^+)^*$. ν gives the real valuation of the clocks $\mathcal{C}(l)$ visible at location l, thus $|\nu(l)| = |\mathcal{C}(l)|$. If $\neg\mathit{Active}(l)$ then $\nu(l) = \lambda$.
– θ reflects the history, that might be restored by entering super-states via history entries. It is split up in the two functions θ_{state} and θ_{var}, where $\theta_{state}(l)$ returns the last visited sub-state of l—or an entry of the sub-state, in the case where the sub-state 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 $\mathcal{C}(root)$.

History. The predicate $\mathit{HasHistory}(l) \overset{def}{=} \exists n \in \delta(l).\ \mathit{HISTORY}(n)$ captures the existence of a history entry. If $\mathit{HasHistory}(l)$ holds, the term $\mathit{HEntry}(l)$ denotes the unique history entry of l. If $\mathit{HasHistory}(l)$ does not hold, the term $\mathit{HEntry}(l)$

denotes the default entry of l. If l is basic $\mathit{HEntry}(l) = l$. If none of the above is the case, then $\mathit{HEntry}(l)$ is undefined.

Initially, $\forall l \in S.\mathit{HasHistory}(l) \Rightarrow \theta_{state}(l) = \mathit{HEntry}(l) \wedge \theta_{var}(l) = 0^{V(l)}$.

Reached locations by forks. In order to denote the set of locations reached by following a fork, we define the function $\mathit{Targets}_\theta : 2^S \rightarrow 2^S$ relative to θ.

$$\mathit{Targets}_\theta(L) \overset{def}{=} L \cup \bigcup_{l \in L} \{n | n \in \theta_{state}(l) \wedge \mathit{HISTORY}(l)\} \cup \{n | l \xrightarrow{r} n \wedge \mathit{ENTRY}(l)\}$$

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

Configuration-vector transformation. Taking 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. We represent this complex transition by a transformation function $\mathcal{T}_t$, which depends on a particular transition t.

The three parts of this step are described as follows.

1. *join:*
   (ρ, μ, ν, θ) is transformed to $(\rho^1, \mu^1, \nu^1, \theta^1)$ as follows:
   ρ is updated to $\rho^1 := \rho[\forall n \in \rho^\times(l).\ n \mapsto \varnothing]$.
   μ is updated to $\mu^1 := \mu[\forall n \in \rho^\times(l).\ n \mapsto \lambda]$.
   ν is updated to $\nu^1 := \nu[\forall n \in \rho^\times(l).\ n \mapsto \lambda]$.

   If $\mathit{EXIT}(l)$, the history is recorded. Let H be the set of super-states $h \in \rho^\times(\delta^{-1}(l))$, where $\mathit{HasHistory}(h)$ holds. Then
   $\theta^1_{state} := \theta_{state}[\forall h \in H.\ h \mapsto \mathit{HEntry}(\rho(h))]$ and
   $\theta^1_{var} := \theta_{var}[\forall h \in H.\ h \mapsto \mu(h)]$.
   If $\neg\mathit{EXIT}(l)$ or $H = \varnothing$, 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 $\mathit{Targets}^*_{\theta^2}(l')$. Note that $\rho^2(n) = \varnothing$ for all $n \in \delta^\times(l')$. Thus we can compute ρ^3 as follows:

```
$\rho^3 := \rho^2$
FORALL  $n \in \mathit{Targets}^*_{\theta^2}(l')$
    IF $\mathit{ENTRY}(n)$
        THEN $\rho^3(\delta^{-2}(n)) := \rho^3(\delta^{-2}(n)) \cup \{\delta^{-1}(n)\}$
        ELSE $\rho^3(\delta^{-1}(n)) := \{n\}$      /* BASIC */
```

   μ^3 is derived from μ^2 by first initializing all local variables of the super-states

s in $\mathit{Targets}^*_{\theta^2}(l')$, i.e., $\mu^3(V(s)) := 0^{V(s)}$. If $\mathit{HasHistory}(s)$, $\theta_{var}(s)$ 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 μ^3 and ν^3. The history does not change, θ^3 is identical to θ^2.

Note that parts 1. and 3. correspond to the identity transformation, if l and l' are basic locations.

We define the configuration-vector transformation $\mathcal{T}_t$ for a transition $t : l \xrightarrow{g,s,r,u} l'$:

$$\mathcal{T}_t(\rho,\mu,\nu,\theta) \overset{def}{=} (\rho^3,\mu^3,\nu^3,\theta^3)$$

If the context is unambiguous, we use $\rho^{\mathcal{T}_t}$ and $\nu^{\mathcal{T}_t}$ for the parts ρ^3 respectively ν^3 of the transformed configuration corresponding to transition t.

Starting points for joins. A super-state s can only be exited, if all its parallel sub-states can synchronize on this exit. For an exit $l \in \delta(s)$ we note by $\mathit{PreExitSets}(l)$ the family of sets of exits. If transitions are enabled to all exits in $X \in \mathit{PreExitSets}(l)$, then all sub-states can synchronize.

Rule predicates. To give the rules, we need to define predicates that evaluate conditions on the dynamic tree ρ. We introduce the set set of active leaves (in the tree described by ρ), which are the innermost active states in a super-state l:

$$\mathit{Leaves}(\rho,l) \overset{def}{=} \{n \in \rho^{\times}(l) \mid \rho(n) = \varnothing\}$$

The predicate expressing that all the sub-states of a state l can synchronize on a join is:

$$\begin{aligned}\mathit{JoinEnabled}(\rho,\mu,\nu,l) \overset{def}{=}\ & \mathit{BASIC}(l) \vee \\ & \exists X \in \mathit{PreExitSets}(l).\ \forall n \in \mathit{Leaves}(\rho,l).\ \exists n' \in X.\ n \xrightarrow{g} n' \ \wedge\ g(\mu,\nu)\end{aligned}$$

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

For the invariants of a location we use a function $\mathit{Inv}_\nu : S \to \{\mathbf{true},\mathbf{false}\}$, that evaluates the invariant of a given location with respect to a clock evaluation ν. We use the predicate $\mathit{Inv}(\rho,\nu)$ to express, that for control situation ρ and clock valuation ν all invariants are satisfied.

$$\mathit{Inv}(\rho,\nu) \overset{def}{=} \bigwedge_{n \in \rho^{\times}(root)} \mathit{Inv}_\nu(n)$$

We introduce the predicate *TransitionEnabled* over transitions $t : l \xrightarrow{g,s,r,u} l'$, that evaluates to **true**, if t is enabled.

$$\begin{aligned}&\mathit{TransitionEnabled}(t : l \xrightarrow{g,s,r,u} l',\rho,\mu,\nu) \overset{def}{=} \\ &\qquad g(\mu,\nu) \wedge \mathit{JoinEnabled}(\rho,\mu,\nu,l) \wedge \mathit{Inv}(\rho^{\mathcal{T}_t},\nu^{\mathcal{T}_t}) \wedge \neg \mathit{EXIT}(l')\end{aligned}$$

Since urgency has precedence over delay, we have to capture the global situation, where some urgent transition is enabled. We do this via the predicate *UrgentEnabled* over a configuration.

$$
\begin{aligned}
\mathit{UrgentEnabled}(\rho,\mu,\nu) \overset{def}{=}\ & \exists t : l \xrightarrow{g,r,u} l'.\ \mathit{TransitionEnabled}(t,\rho,\mu,\nu)\ \wedge\ u \\
& \vee\ \exists t_1 : l_1 \xrightarrow{g_1,c!,r_1,u_1} l_1', t_2 : l_2 \xrightarrow{g_2,c?,r_2,u_2} l_2'. \\
& \qquad \mathit{TransitionEnabled}(t_1,\rho,\mu,\nu)\ \wedge \\
& \qquad \mathit{TransitionEnabled}(t_2,\rho,\mu,\nu)\ \wedge\ (u_1 \vee u_2)
\end{aligned}
$$

Rules. We give now the action rule. It is not possible to break it in join, action, and fork because the join can be taken only if the action is enabled and the action is taken only if the invariants still hold after the fork. The predicate *TransitionEnabled* takes into account the join, the action, and the fork conditions. The inferred transition is computed with the configuration-vector transformation.

$$
\frac{\mathit{TransitionEnabled}(t : l \xrightarrow{g,r,u} l',\ \rho,\mu,\nu)}{(\rho,\mu,\nu,\theta) \xrightarrow{t} \mathcal{T}_t(\rho,\mu,\nu,\theta)}\ \mathit{action}
$$

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 super-states. In the later case, this includes the appropriate joins and/or fork operations.

The delay transition rule is:

$$
\frac{\mathit{Inv}(l)(\rho,\nu+d) \qquad \neg\mathit{UrgentEnabled}(\rho,\mu,\nu)}{(\rho,\mu,\nu,\theta) \xrightarrow{d} (\rho,\mu,\nu+d,\theta)}\ \mathit{delay}
$$

where $\nu + d$ stands for the clock assignment ν shifted by the delay d. Time can elapse only if all the invariants stay satisfied and no urgent transition is enabled.

The last transition rule reflects the situation, where two action transitions synchronize via a channel c.

$$
\frac{\begin{array}{ll}\mathit{TransitionEnabled}(t_1 : l_1 \xrightarrow{g_1,c!,r_1,u_1} l_1',\ \rho,\mu,\nu) & l_1 \notin \delta^\times(l_2) \\ \mathit{TransitionEnabled}(t_2 : l_2 \xrightarrow{g_2,c?,r_2,u_2} l_2',\ \rho,\mu,\nu) & l_2 \notin \delta^\times(l_1)\end{array}}{(\rho,\mu,\nu,\theta) \xrightarrow{t_1,t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho,\mu,\nu,\theta)}\ \mathit{sync}
$$

The order $\mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}$ could equivalently be replaced by $\mathcal{T}_{t_1} \circ \mathcal{T}_{t_2}$ since the assignments cannot conflict with each other (according to the well-formedness constraints on transitions).

If no action transition is enabled or becomes enabled when time progresses, we have a *deadlock* configuration, which is typically a bad thing. If in addition time is prevented to elapse, this is a *time stopping deadlock*. Usually this is an error in the model, since it does not correspond to any real world behavior.

Our rules describe all legal sequences of transitions. A *trace* is a finite of infinite sequence of legal configurations that start at the initial configuration S_0 with all variables and clocks set to 0. Any two subsequent configurations are connected according to one of the transition rules. For our purposes it suffices to associate a hierarchical timed automaton semantically with the (typically infinite) set of all derivable traces.

4 Translation of Hierarchical Timed Automata to Uppaal Timed Automata

In this section we outline the procedure for translating one hierarchical timed automaton to a parallel composition of (flat) Uppaal timed automata [12]. We use the model of a pacemaker as a running example. We implemented our procedure in Java.

4.1 Uppaal Timed Automata

Uppaal [12] is a tool box for modeling, verification and simulation or real-time systems developed jointly by Uppsala University and Aalborg University. It is appropriate for systems that can be described as collection of non-deterministic parallel processes. The model used in Uppaal is the timed automaton and corresponds to the flat version of our hierarchical timed automaton where each process is described as a state machine with finite control structure, real-valued clocks and integers. Processes communicate through channels and (or) shared variables [11]. The tool has been successfully applied in many case studies [14, 15, 7].

4.2 Flattening a Hierarchical Timed Automaton

Syntactically, HTAs are generated by a template mechanism that has to be instantiated. The number of templates can be substantially smaller than the number of super-states in the hierarchical state machine.

On the topmost level, conceptually under an implicit root, we find a parallel composition of instantiated templates. Each corresponds to a super-state S_i, that can itself instantiate templates in sub-states and so on. This gives rise to an *instantiation tree*, which expresses the actual behavior of the hierarchical timed automaton.

The translation proceeds in three phases:

1. *Collection of instantiations:* the hierarchical instantiation tree is traversed and for every hierarchical super-state, the skeleton of a (flat) template is constructed.
2. *Computation of global joins:* transitions originating from super-states can require a cascade of sub-state exits—called *global join*—in order to be taken. All combinations of possible start configurations are computed; this yields a guard condition, that evaluates to **true** if an only if one such cascade can be taken to completion.
3. *Post-processing channel communication:* if a transition in the hierarchical timed automaton formalism starts at a super-state S and carries a synchronization, it cannot synchronize with a transition *inside* S. Since the sub-state/super-state relation is lost in the translation, we resolve this scope conflict explicitly. We do so by introducing duplications of channels and transitions.

Every super-state S in the hierarchical timed automaton model corresponds exactly to one UPPAAL timed automaton $\hat{S}$. We can relate control locations ρ in the hierarchical timed automaton model to a *control vector* $\hat{\rho}$ in the UPPAAL model. This correspondence allows us to trace back an error sequence obtained with the flat representation to the original hierarchical one.

5 Formal Verification of a Cardiac Pacemaker

In this case study, we use a cardiac pacemaker example, as it is described in various UML books, e.g. [5]. We translate our hierarchical timed automaton model of it to an equivalent (flat) UPPAAL timed automata model and report on run-time data of the formal verification of deadlock, one safety, and one liveness property.

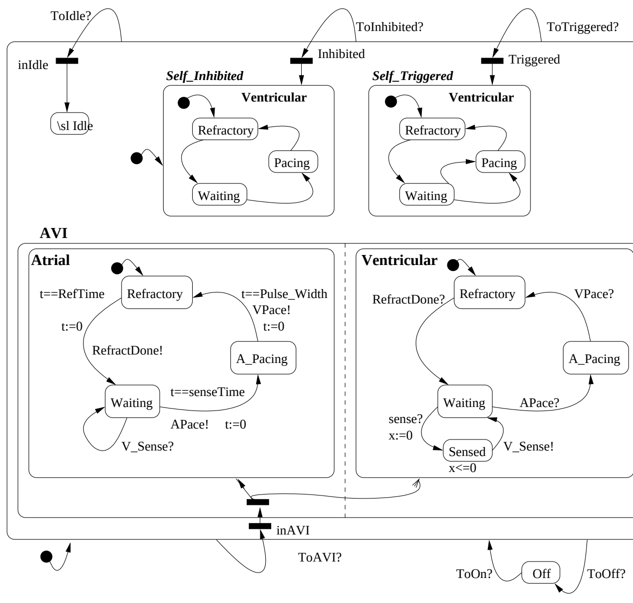


Fig. 1. Overview of our hierarchical timed automaton pacemaker model. Initially, the VVI mode is entered.

5.1 The Cardiac Pacemaker Model

The main component of the pacemaker is a *XOR* super-state with the two substates *Off* and *On*. If the pacemaker is on, it can be in the different modes *Idle*, AAI, AAT, VVI, VVT, and AVI. The first letter indicates, to which chamber of the heart an electrical pacing pulse is sent (articular or ventricular). The second letter indicates, which chamber of the heart is monitored (articular or ventricular). In the *Self_Inhibited* (I) modes, a naturally occurring heartbeat blocks a pulse from being sent, whereas in the *Self_Triggered* (T) modes a pacing pulse will always occur, either triggered by a timeout or by the heart contraction itself.

For simplicity, we restrict to the operation modes *Idle*, VVT, VVI, and AVI. Of particular interest is the AVI mode, which is described as an *AND* super-state with two parallel sub-states that are entered on demand. Thus, in our example only the ventricular chamber is observed, but a pace signal my be sent either to the ventricular or articular chamber.

Programmer Model. The signals `commandedOn!` `commandedOff!` `toIdle!` `toVVI!` `toVVT!` `toAVI!` are issued by a medical person, called the *programmer* in our context. We do not make assumptions, on how or in which order she issues these signals, but require a time delay of at least `DELAY_AFTER_MODESWITCH` after each signal. If one of the signals `commandedOff!` or `toIdle!` was issued, this is recorded in the binary variable `wasSwitchedOff`.

Note that we equipped the pacemaker with default exits, thus it can *always* synchronize with these signals.

Composed Model. The complete hierarchical timed automaton model contains in parallel the pacemaker, the programmer, and a model of a heart, that might spontaneously cease beating on its own (not described here).

	HTA model	Uppaal model
# XML tags	549	1233
# proper control locations	35	45
# pseudo-sates / committed locations	31	62
# transitions	47	174
# variables and constants	33	90
# formal clocks	6	6

Table 2. Translations of a hierarchical timed automaton description to an equivalent flat Uppaal model. For the cardiac pacemaker example, the increases are moderate. Both data formats are described in terms of XML grammars.

5.2 Model-Checking the Uppaal Model

The automatic translation of the pacemaker model yielded a gentle expansion in size, as recorded in Table 2. The high number of committed location indicates, that most of the additional control structure is purely auxiliary and does not contribute significantly to the state space of the translation.

We used the translation as input to the Uppaal tool. All run-times were measured on a Sun Enterprise 450 with UltraSPARC-II processors, 300 MHz, It took 0.92 seconds to establish deadlock-freedom. We verified two desirable properties in the obtained hierarchical timed automaton model.

```
(i)   A[] ( heart_sub.FLATLINE => (wasSwitchedOff == 1) )
(ii)  A[] ( heart_Sub.AfterAContraction =>
                A<> heart_Sub.AfterVContraction )
```

Property *(i)* is a safety property and establishes, that the heart never stops for too long, unless the pacemaker was switched off by the programmer (in which case we cannot give any guarantees). Property *(ii)* is a response property and states, that after an articular contraction, there will *inevitably* follow a ventricular contraction. In particular, this guarantees that no deadlocks are possible between these control situations.

The latest version of the Uppaal tool[1] was able to perform the model-checking of both properties successfully in 13.30 respectively 4.11 seconds. The verification of the typically more expensive property *(ii)* was faster, since here we were able to apply a property preserving convex hull over-approximation. This approximation yields false negatives for property *(i)*. We note that using Uppaal's powerful optimization options, in particular the active clock reduction, reduces also model-checking times drastically.

```
REFRACTORY_TIME  = 50
SENSE_TIMEOUT    = 15

DELAY_AFTER_V = 50
DELAY_AFTER_A =  5

HEART_ALLOWED_STOP_TIME = 135

MODE_SWITCH_DELAY  = 66
```

Fig. 2. Constants that yield property *(i)*.

It is worthwhile to mention that validity of property *(i)* is strongly dependent on the parameter setting of the model. We used the constants from Figure 2. If the programmer is allowed to switch between modes very fast, it is possible that she prevents the pacemaker from doing its job. E.g., for `MODE_SWITCH_DELAY = 65` the property *(i)* does not hold any more. In practice it is often a problem to find parameter settings, that entail a safe or correct operation of the system.[2]

6 Conclusion & Further Work

We extract a subset of the behavioral part of UML for the purpose of formal verification. We extend it with real-time constructs, i.e., with real-valued formal clocks, invariants, and timed guards. We use a simple hand-shake synchronizations mechanism to express dependencies among components. For this formalism we give a formal semantics to capture the exact behavior. This makes it possible to translate our hierarchical structure to a flat timed automaton model while preserving properties like timed reachability. We make use of this by applying

[1] A release version that supports—among other new features—the possibility to model-check response properties is available since April 2001.

[2] In related work, an extended version of Uppaal is used to derive parameters yielding property satisfaction automatically, see [10].

a mature model-checking algorithm and by this means established time-critical safety and response properties of a pacemaker model.

Our formal extension of statecharts to timed statecharts is about to be finalized in a UML profile in the context of the European AIT-WOODDES project No IST-1999-10069. Here, our proposed method is applied in the verification part of a design methodology for real-time and embedded systems. Among other tools, the mature UPPAAL model-checking engine is used as a back-end. The run-time data we get from our pacemaker example is encouraging—it suggests that reasonable-sized models are in the reach of algorithmic treatment with formal method tools.

The pacemaker example indicates, that clocks, guards, and invariants are a feasible selection of real-time constructs. Though not necessarily familiar to the designer, these constructs are expressive enough to capture essential real-time behavior and nevertheless stay in a decidable fragment of real-time properties. For every real-time model that can be encoded in our formalism, this opens the way for formal and fully automated algorithmic verification in many interesting cases. This suggests that real-time temporal logics can be included into the UML requirement specification language.

Future Work. Event communication can be coded by hand with the help of channel synchronizations and global variables. The inclusion of events into hierarchical timed automata can be expressed by this way. Extensions of the action language to other data types are planned, and the possibility of safe over-approximation of C++ statements has to be investigated.

Since checking real-time temporal logics is computationally hard under various aspects [2, 1], it is desirable to try our technique on larger examples from industrial designs. Currently the formal verification part is possible via a translation to a flattened version of the system. However, there is indication that the hierarchical structure can be exploited. We plan to investigate this further in the context of the UPPAAL tool, see [3].

References

1. Luca Aceto and François Laroussinie. Is your Model Checker on Time? In *Proc. 24th Int. Symp. Math. Found. Comp. Sci. (MFCS'99), Szklarska Poreba, Poland, Sep. 1999*, volume 1672 of *Lecture Notes in Computer Science*, pages 125–136. Springer–Verlag, 1999.
2. Rajeev Alur and Thomas A. Henzinger. Real-time Logics: Complexity and Expressiveness. *Information and Computation*, 1(104):35–77, 1993. preliminary version appeared in Proc. 5th LICS, 1990.
3. Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Proc. of the Summer School on Modelling and Verification of Parallel Processes (MOVEP'2k), Nantes, France, June 19 to 23, 2001.*

4. Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001. see `http://www.brics.dk/RS/01/11/`.
5. Bruce Powel Douglass. *Real-Time UML, Second Edition - Developing Efficient Objects for Embedded Systems.* Addison-Wesley, 1999.
6. David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 7(30):31–42, July 1997.
7. Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, December 1997.
8. Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.
9. Gerand J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
10. Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. Research Series RS-01-5, BRICS, Department of Computer Science, University of Aarhus, January 2001. 44 pp.
11. Paul Pettersson Kim G. Larsen and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of the 10th International Conference on Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer–Verlag, 1995.
12. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
13. Johan Lilius and Ivan Porres. Formalising UML State Machines for Model Checking. In *UML'99 - The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer–Verlag, October 1999.
14. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In *Proc. of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems.*, volume 1384 of *Lecture Notes in Computer Science*, pages 281–297. Springer–Verlag, 1998.
15. Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Start-Up Mechanism. In *Proc. of IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997.
16. ObjectGeode is a commercial product of Verilog. Documentation and whitepapers are available from `http://www.telelogic.com/ObjectGeode/Geode_Articles.asp`.
17. Carsta Petersohn and Luis Urbina. A timed semantics for the STATEMATE implementation of statecharts. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 553–572. Springer–Verlag, September 1997. ISBN 3-540-63533-5.
18. Rhapsody is a commercial product of I-Logix. Documentation and whitepapers are available from `http://www.ilogix.com/quick_flinks/white_papers/index.cfm`.
19. visualState™ is a commercial product of IAR Systems. Detailled information is available from `http://www.iar.com`.