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@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 a rule-based formal semantics. To formally verify deadlock-freedom as well as safety and response properties expressed in TCTL using an existing model-checker e.g. UPPAAL, we present a translation of HTAs to networks of timed automata, that are enriched with data types, committed locations, and hand-shake synchronization. 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 paradigm and expressive yet abstract formalisms. Object orientation concepts provide—among many other things—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 concepts, 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 (possibly very close) mathematical formalization of the model. This makes it possible to prevent errors inexpensively at early design stages.

In the area of real-time systems—where correctness does not only depend on functionality but also timeliness—it is crucial to validate complex layouts. Industrial CASE tools, e.g., visualState™ [vis], 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 formalism allow *execution* of the model, that unfolds and visualize system behavior.

The UML statechart formalism focuses on the control aspect, where event communication and data steers the sequence of system model states. Often the behavior is dependent on real-time properties [Dou99] and is therefore supported by industrial tools like Rhapsody [Rha,HG97]. 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 *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

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

purpose, equipped with construct to express real-time properties, namely *timed computation tree logic* (TCTL, [HNSY94]). 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 (flaw-fully) unspecified. Simulators, e.g., ObjectGeode [Obj], can execute behavioral descriptions and can help to *validate* systems, i.e., discover design flaws, if they occur in a simulation session. However, similar to testing, simulators cannot show 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. The carry-over of properties relies on assumptions about a realization, e.g., that a local hardware bus can be accessed in below 2μs. Sometimes these values are included as parameters. More often, we choose to ignore these details if their total effect is negligible.

In this paper we describe a way to extract a behavioral part of a UML model for formal verification. In order to resolve ambiguities, we equip this part with formal syntax and semantics. We sketch a translation of our (hierarchical) formalism into a parallel composition of timed automata, that serve as input to the UPPAAL verification tool. The detailed description can be found in [DM01]. We establish deadlock-freedom and TCTL safetey and (unbounded) response properties of a pacemaker model.

Organization. The following section describes the formal syntax of our UML statechart restriction, extended with real-time constructs. Section 3 contains the formal semantics. In Section 4 we sketch a translation of this formalism to 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 Formal Syntax of HTAs

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.

2.1 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 location, denoted by l.

Integer variables Let V be a finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a superstate l.

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 superstate 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 superstate 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 $c \in Ch$, $c?$, $c! \in Sync$. For $s \in Sync$, $\bar{s}$ denotes the matching complementary, i.e., $\bar{c!} = c?$ and $\bar{c?} = c!$.

Guards and invariants A data constraints is a boolean expressions of the form $A \sim A$, where A is an arithmetic expression over V and $\sim \in \{<, >, =, \leq, \geq\}$. A clock constraints is an expressions 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 set of clock resets and data assignments.

2.2 Structural Components

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

Def 1 *A hierarchical timed automaton is a tuple* $\langle S, S_0, \delta, \sigma, V, \mathcal{C}, Ch, T\rangle$ *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 substates of* l. δ *is required to give rise to a tree structure with root root. We readily extend* δ *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.*
- $V, \mathcal{C}, Ch$ *are sets of variables, clocks, and channels. They give rise to Guard, Reset, Sync, and Invariant as described in Section 2.1.*
- $Inv : S \to$ *Invariant maps every locations* l *to an invariant, possibly to the constant* **true**.
- $T \subseteq S \times (Guard \times Sync \times Reset \times \{\mathbf{true}, \mathbf{false}\}) \times S$ *is the set of transitions. A transition connects two locations* l *and* l', *has a guard* g, *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 $TYPE(l)$ for $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HIS$ $l \in S$. E.g., $AND(l)$ is 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 $ENTRY(l) \vee HISTORY(l)$.

We define the parent function

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

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

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

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

2.3 Well-Formedness Constraints

We give the rules to ensure consistency of a given hierarchical timed automaton.

Location constraints We require a number of sanity properties on locations and structure.

The function δ has to give rise to a proper tree rooted at *root*, and $S = \delta^*(root)$.

Basic nodes are empty: $BASIC(l) \Leftrightarrow \delta(l) = \varnothing$.

Substates of *AND* superstate are not basic: $AND(l) \wedge n \in \delta(l) \Rightarrow \neg BASIC(n)$.

Invariants of pseudo-locations are trivial: $HENTRY(l) \vee EXIT(l) \Rightarrow Inv(l) = \mathbf{true}$.

Initial location constraints S_0 has to correspond to a consistent and proper control situation, i.e., $root \in S_0$ and for every $l \in S_0$ it the following holds:

(i) $BASIC(l) \vee XOR(l) \vee AND(l)$,
(ii) $l = root \vee \delta^{-1}(l) \in S_0$,
(iii) $XOR(l) \Rightarrow |\delta(l) \cap S_0| = 1$, and
(iv) $AND(l) \Rightarrow \delta(l) \cap S_0 = \tilde{\delta}(l)$.

Variable constraints We explicitly disallow conflict in assignments in synchronizing transitions: It holds that $l_1 \xrightarrow{g,c!,r,u} l_1'$, $l_2 \xrightarrow{g',c?,r',u'} l_2' \in T \Rightarrow vars(r) \cap vars(r') = \varnothing$, 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.

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 $Ch^+(\delta^{-1}(l))$.

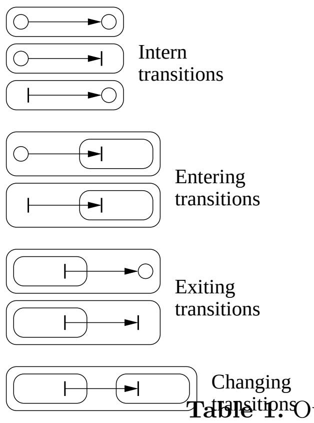
Entry constraints Let $e \in S$, $HENTRY(e)$. If $XOR(\delta^{-1}(l))$, then T contains exactly one transition $e \xrightarrow{r} l'$. If $AND(\delta^{-1}(l))$, then T contains exactly one transitions $e \xrightarrow{r} e_i$ for every proper substate $l_i \in \tilde{\delta}(\delta^{-1}(l))$, and $e_i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

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.

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 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 $l \longrightarrow l'$.



Comment	l	l'	Constraint
Intern	*BASIC*	*BASIC*	$\delta^{-1}(l) = \delta^{-1}(l')$
	BASIC	*EXIT*	
	HENTRY	*BASIC*	
Entering and fork	*BASIC*	*HENTRY*	$\delta^{-1}(l) = \delta^{-2}(l')$
	HENTRY	*HENTRY*	
Exiting and join	*EXIT*	*BASIC*(l)	$\delta^{-2}(l) = \delta^{-1}(l')$
	EXIT	*EXIT*	
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 superstates. Configurations are of the form (ρ, μ, ν, θ), where

- $\rho : S \rightarrow 2^S$ captures the control situation. ρ can be understood as a partial, dynamic version of δ, that maps every superstate s to the set of active substates. If a superstate s is not active, $\rho(s) = \varnothing$. We define $\mathit{Active}(l) \overset{def}{=} l \in \rho^{\times}(\mathit{root})$, where $\rho^{\times}(l)$ is the set of all active sub-states of l. Notice that $\mathit{Active}(l) \Leftrightarrow l \in \rho(\delta^{-1}(l))$.
- $\mu : S \rightarrow (\mathbb{Z})^*$. μ gives the valuation of the local integer variables of a superstate 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 $|V(l)|$.
- $\nu : S \rightarrow (\mathbb{R}^+)^*$. 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 superstates via history entries. It is split up in the two functions θ_{state} and θ_{var}, where $\theta_{state}(l)$ returns the last visited substate of l—or an entry of the substate, 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 $\mathcal{C}(\mathit{root})$.

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

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

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. Together, this defines a run-to-completion step. We represent a run-to-completion step formally 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 superstates $h \in \rho^{\times}(\delta^{-1}(l))$, 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 superstates 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 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 $\mathit{PreExitSets}(l)$. Each element X of $\mathit{PreExitSets}(l)$ is itself a set of exits. If transitions are enabled to all exits in X, then all substates can synchronize.

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

Here, the operator $\boxtimes : (2^{2^S}) \times (2^{2^S}) \rightarrow 2^{2^S}$ is a product over families of sets, i.e., it maps $(\{A_1, \ldots, A_a\}, \{B_1, \ldots, B_b\})$ 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.

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 superstate l:

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

The predicate expressing that all the substates 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.

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

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.

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

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 superstates. 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 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.

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

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

We choose a particular order here but it is not crucial since our well-formedness constraints ensure, that the assignments cannot conflict with each other.

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 or infinite sequence of legal transitions, that start at the initial configuration S_0, with all variables and clocks set to 0. For our purposes it suffices to associate a hierarchical timed automaton semantically with the (typically infinite) set of all derivable traces.

4 From Hierarchical Timed Automata to Uppaal

In this section we outline our procedure for translation of our hierarchical timed automata to a parallel composition of (flat) Uppaal timed automata [LPY97]. We use the model of a pacemaker as a—hopefully—running example. The detailed flattening algorithm can be found in [DM01].

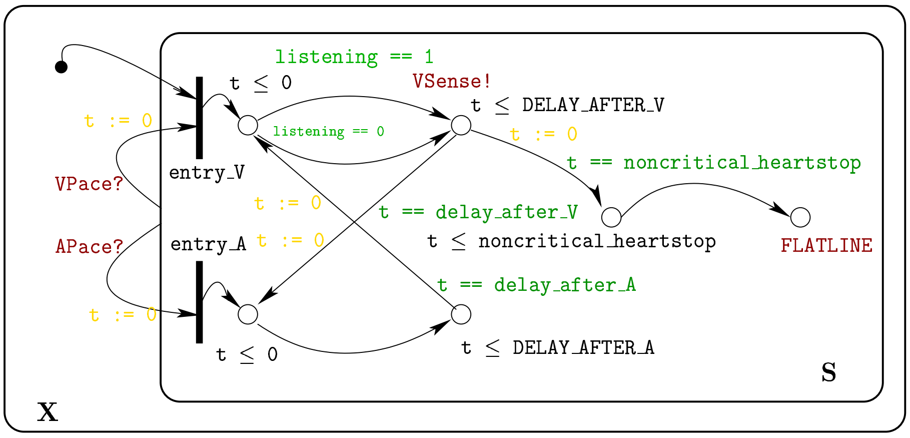


Fig. 1. A simplified model of a human heart, that might require pacing. The human heartbeat is in fact a complex sequence of chamber contractions, in this model we consider two chambers the (left) *atrial* and *ventricular*. A healthy heart will contract those in a steady rhythm, dictated by the time delays `DELAY_AFTER_V` and `DELAY_AFTER_A`. We use the local clock `t` to model this rhythm. Since in our example we only monitor the ventricular chamber, this one synchronizes on `VSense`, in case that anybody is listening (indicated by `listening == 1`).
After the contraction of the ventricular chamber, our model might non-deterministically stop beating on own account. If it does so for too long, the critical state FLATLINE is reached.
A pacemaker can send an impulse either to the atrial or ventricular chamber, i.e., synchronize on channels `APace` or `VPace`. The particular heart chamber then is scheduled for contraction in the very next moment, no matter when these signals occur. This is modeled by using the default exit and re-entering at one of the leftmost locations.

4.1 Uppaal Timed Automata

Uppaal [LPY97] 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

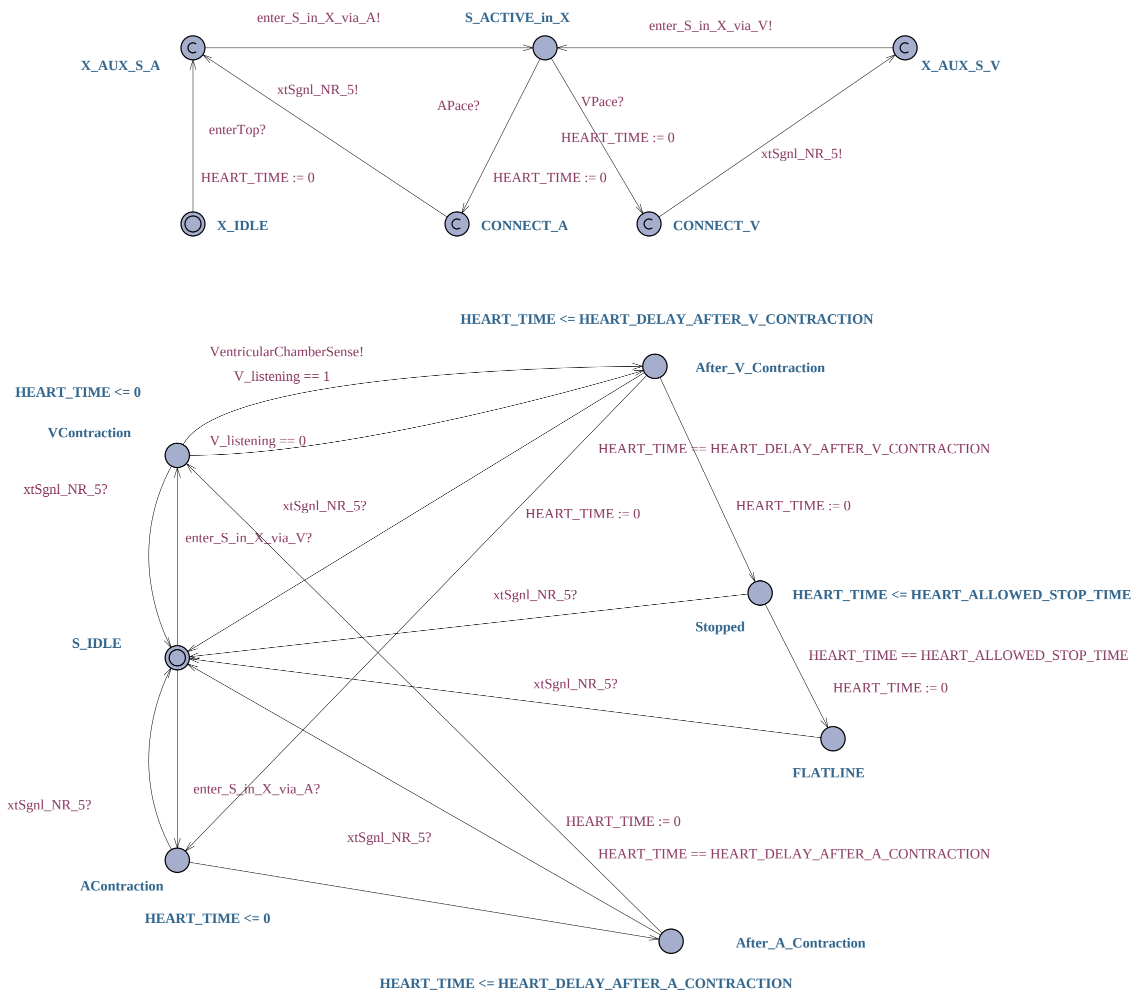


Fig. 2. Two UPPAAL timed automata templates encoding the heart model in Figure 1. The upper automaton is responsible for selecting the entry `VContraction` or `AContraction`, the lower automaton encodes the behavior.

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 [KGLY95]. The tool has been successfully applied in many case studies [LPY98,LP97,HSLL97].

As a special modeling construct, locations can be declared *committed*, indicated by a `c`. If a committed location l is active, it must be left as soon as possible, i.e., no time delay is possible and all transitions originating in non-committed locations are blocked, unless they synchronize with a transition leaving l. Committed locations can be used to encode more complex behavior, but also to reduce the number of possible interleavings and thus render state space exploration more efficient.

4.2 Translating Superstates

The fundamental concept of our flattening algorithm is the translation of every hierarchical superstate into one UPPAAL automaton. All these automatons are put in parallel. This can lead to an exponential blow-up in terms of templates, or in other words, of the state space. This is a consequence of the fact that hierarchical models can be exponentially more concise [AKY99]. Some auxiliary structures have to be introduced in order to mimic the behavior of hierarchical machines adequately.

Translation of XOR *Superstates.* In a hierarchical *XOR* template X, at most one location is active at the same point in time. To represent the situation that none is active, we introduce —in the translation $\hat{X}$—the special location `X_IDLE`, which is also the initial state. All entries are translated by a transition from `X_IDLE`.

For every sub-state S of X we introduce a location `S_ACTIVE_IN_X` in $\hat{X}$. In Figure 1, the *XOR* superstate X has only one substate **S**. **X** and **S** are translated to the two timed automatons in Figure 2.

Moreover, for every entry e of S we introduce an auxiliary location in X, called `X_AUX_S_`e. These are declared committed and are connected to `S_ACTIVE_IN_X` with a transition, that synchronizes on a signal `enter_S_in_X_via_`e. Transitions leading originally to a S-entry e in X are represented in the translation by leading to `X_AUX_S_`e and trigger—without interleaving with other components—the activation of the sub-state S.

Exits of this sub-state S are more complicated, for they are only possible, if all sub-components of S can exit. This is described as global joins, see Section 4.3.

If superstate X is inactivated, this is realized in our translation by transitions to `IDLE_X`, that are triggered by an `exit_X` synchronization channel. If the superstate X has a default exit, every non-auxiliary location in $\hat{X}$ has a transition to `IDLE_X`.

Translation of AND *Superstates.* A hierarchical *AND* machine A is a parallel composition of sub-machines, where either none or all of them are active. In the translation $\hat{A}$ (Figure 3), these situations are represented by the locations `A_IDLE` and `A_ACTIVE`. If A is activated, this is always specific to a designated entry $\mathtt{e}_i$ of A. The sub-machines S_i of A are all entered, but the signals `enter_Si_via_ej` depend on the choice of `ej`. Therefore, for every entry there is a separate chain leading from `A_IDLE` to `A_ACTIVE`. The auxiliary locations in between are declared committed (marked by a `c`), thus there are no time delays possible.

The exit of A is represented in $\hat{A}$ a transition from `A_ACTIVE` to `A_IDLE`, which carries the synchronization signal `exit_A`.

4.3 Global Joins

Transitions originating from superstates are a subtle issue, for they may require a cascade of substate exits, in order to be taken. In Figure 4 (a), the substates S1, S2, and S3 have to be exited, before `LOC` can be reached. If Sn is active in S2, it has to be exited as well.

This cascade of exits is encoded the sequence of in Figure 4 (b). The auxiliary variable `trigger` keeps track of the number of active basic locations, that are connected to this global join via a transition to an exit. It has to reach the threshold value `N` to enable the first transition. Moreover, it has to be possible to mimic the transition to `LOC`, i.e., the `guard` (if any) has to be satisfied and synchronization (if any) has to be possible. We assume that synchronization is not possible with transitions *inside* S1. If this situation arises in the given HTA model, we introduce new channels to avoid this conflict and duplicate transitions accordingly. An auxiliary boolean variable `BLOCK` in order to exclude interleaving with other global joins.

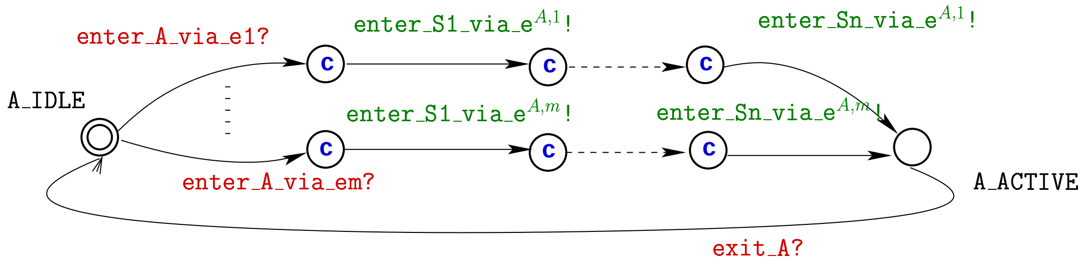
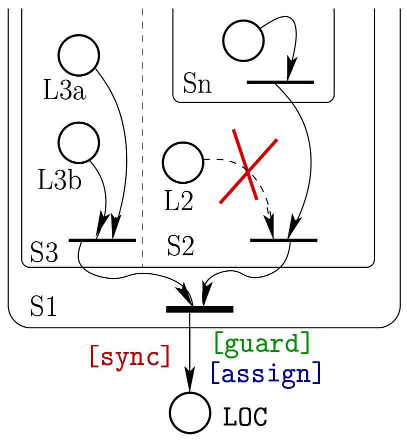
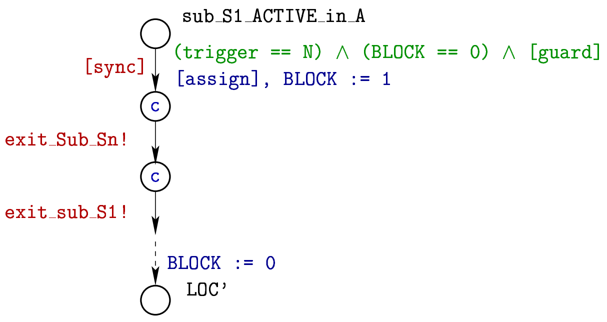


Fig. 3. Translation of entering and exiting an *AND* component.



(a) Part of hierarchical timed automaton X



(b) Translation of the global join in $\hat{X}$

Fig. 4. Translation of a global join in the topmost *XOR* component *X*.

One transition in a HTA can give rise to a number of global joins, possibly exponential in the depth of hierarchical structure. In Figure 4, the locations `L3a` and `L3b` can be treated uniformly, but the location `L1` has to be encoded in a *different* global join, where there is no exit of sub-state `Sn`.

4.4 Putting the Pieces Together

One level below the root, we find a parallel composition of superstates S_i. Their translations $\hat{S}_i$ are initially in their `Si_IDLE` state. We add one automaton *Global_Kickoff*, that sends an entry signal to every single one of them.

The topmost superstates S_i do not synchronize on exit, but may be enabled to become in-active on their own via following an exit transition. Once a machine becomes inactive, this status can never be revoked in our hierarchical timed automaton formalism, since there is no machine that could accommodate a transition *to* some S_i. If a superstate S should be able to be both inactivated and activated again, it cannot nest at the root level, but must be part of a state machine itself, e.g., one containing the additional basic control location `S_is_IDLE`.

Every superstate 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. For all *XOR* superstates X, $\hat{\rho}$ contains at position $\hat{X}$ either a translation of a basic state $\hat{l}$, `sub_S_active_in_X`, or `IDLE`, depending on whether $\rho(X)$ maps to a basic state, to a substate S, or to $\varnothing$. For *AND* superstate A, $\hat{\rho}(\hat{A}) =$`IDLE` if $\rho(A) = \varnothing$ and $\hat{\rho}(\hat{A}) = \{\hat{S} \mid S \text{ parallel substate of } A\}$ otherwise. The value of the introduced auxillary variables is completely dependend on the current control location, i.e., it is redundand for the state and only serves to enabele or disable transitions.

Proposition A hierarchical state $s = (\rho, u)$ is reachable if and only if a corresponding state $\hat{s} = (\hat{\rho}, u)$ is reachable.

Since entries and exits in the UPPAAL translation are guaranteed to take place without time delay (due to encoding with committed locations), data and clock evaluations u carries over without changes. If a hierarchical trace t exits, it can be mimiced by the translation in each step. Likewise, if a translation $\hat{t}$ of a hierarchical trace is valid in the UPPAAL model, this is due to a sound sequence of entries and exits and corresponds to a trace in the hierarchical timed automaton formalism.

5 Formal Verification of a Cardiac Pacemaker

In this section we describe a hierarchical timed automaton version of a cardiac pacemaker example, as described in various UML books, e.g. [Dou99]. We translate this description to an equivalent (flat) UPPAAL timed automata model and report on run-time data of the formal verification of safety properties.

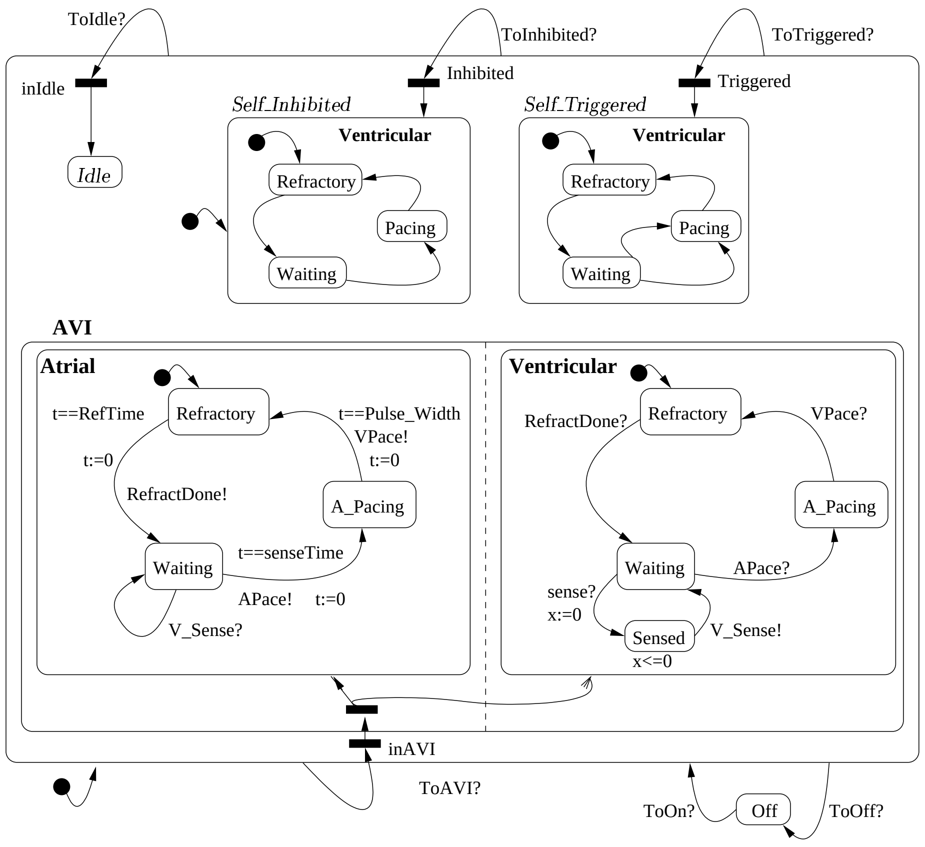


Fig. 5. Overview of our hierarchical timed automaton pacemaker model. Initially, the VVI mode is entered.

5.1 The Cardiac Pacemaker Model

The main component of the pacemaker is a *XOR* superstate with the two sub-states *Off* and *On*. If the pacemaker is on, it can 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* superstate with two parallel substates 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.

	hierarchical timed automaton model	UPPAAL model
# XML tags	549	1233
# proper control locations	35	45
# peudo-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 auxillary and does not contribute significantly to the state space of the translation.

We used the translation as input to the UPPAAL tool. 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.

```
REFRACTORY_TIME  = 50
SENSE_TIMEOUT    = 15

DELAY_AFTER_V = 50
DELAY_AFTER_A =  5

HEART_ALLOWED_STOP_TIME = 135

MODE_SWITCH_DELAY  = 66
```

Fig. 6. Constants that yield property *(i)*.

The latest version of the UPPAAL tool[1] was able to perform the model-checking of both properties successfully in 13.30 repectively 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 overaproximation, that was not preservative with respect to property *(i)*. We used a Sun Enterprise 450 with UltraSPARC-II processors, 300 MHz, and made use of UPPAAL's rich set of optimization options. In particular the active clock reduction reduced also model-checking time drastically.

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

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 6. If the programmer is allowed to swich 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 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 formalized 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 others, 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.

With respect to UML, our investigations indicate a reasonable selection of real-time constructs common to the formal verification community. 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. Furthermore, we propose the inclusion of real-time temporal logics as a unambiguous and (to some extend) algorithmically treatable part of the UML requirement specification language.

Further Work History can be coded by hand with the help of global variables. The inclusion of history directly into hierarchical timed automata can be expressed by this way. The same applies for the event-communication codable with synchronizations. As straightforward extensions we consider treatment of history states and a more expressive communication mechanism via events.

Since checking real-time temporal logics is computationally hard in various ways [AH93,AL99], 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 [ABB+].

References

[ABB+] 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. To appear in Proceedings of the Summer School on Modelling and Verification of Parallel Processes (MOVEP'2k), Nantes, France, June 19 to 23, 2001. Available at `http://www.docs.uu.se/~paupet/`.

[AH93] 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.

[2] In related work, an extended version of Uppaal is used to derive parameters yieling property satisfaction automatically, see [HRSV01].

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating Hierarchical State Machines. In *Proc. of the 26th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer–Verlag, 1999.

[AL99] 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.

[DM01] Alexandre David and M. Oliver Möller. From Hierarichcal Timed Automata to UPPAAL. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001.

[Dou99] Bruce Powel Douglass. *Real-Time UML, Second Edition - Developing Efficitnt Objects for Embedded Systems.* Addison-Wesley, 1999.

[HG97] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 7(30):31–42, July 1997.

[HNSY94] 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.

[HRSV01] 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.

[HSLL97] 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.

[KGLY95] 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.

[LP97] 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.

[LPY97] 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.

[LPY98] 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.

[Obj] ObjectGeode is a commercial product of Verilog. Documentation and whitepapers are available from `http://www.telelogic.com/ObjectGeode/Geode_Articles.asp`.

[Rha] Rhapsody is a commercial product of I-Logix. Documentation and whitepapers are available from `http://www.ilogix.com/quick_links/white_papers/index.cfm`.

[vis] visualState™ is a commercial product of IAR Systems. Detailled information is available from `http://www.iar.com`.