BRICS RS-01-11 David & Möller: From Hierarichcal Timed Automata to UPPAAL

BRICS

Basic Research in Computer Science

From HUPPAAL to UPPAAL

A Translation from Hierarchical Timed Automata to Flat Timed Automata

Alexandre David
M. Oliver Möller

BRICS Report Series RS-01-11

ISSN 0909-0878 March 2001

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/01/11/

From HUppaal to Uppaal

A translation from hierarchical timed automata to flat timed automata

Alexandre David* and M. Oliver Möller†

* **Uppsala University**
Department of Computer Systems
Uppsala University
Box 325
SE - 75105 Uppsala
adavid@docs.uu.se

† BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Århus C, Denmark
omoeller@brics.dk

Version: Vanilla-1 March, 2001

Abstract

We present a hierarchical version of timed automata, equipped with data types, hand-shake synchronization, and local variables. We describe the formal semantics of this *hierarchical timed automata* (HTA) formalism in terms of a transition system.

We report on the implementation of a flattening algorithm, that translates our formalism to a network of Uppaal timed automata. We establish a correspondence between symbolic states of an HTA and its translations, and thus are able to make use of Uppaal's simulator and model checking engine.

This technique is exemplified with a cardiac pacemaker model. Here, the overhead introduced by the translation is tolerable. We give run-time data for deadlock checking, timed reachability, and timed response analysis.

1 Introduction

Hierarchical structures are a powerful mechanism to describe complex systems. They benefit from concepts like modularity and encapsulation and scale up well in industrial settings.

Modeling languages—like UML [BRJ98]—use hierarchical structures to organize 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. Of particular interests are statechart-like models [Har87, HG97], that describe a behavioral view and allow execution of a model on a high level.

Our ambition is to build a hierarchical real-time formalism—called hierarchical timed automata model or HTA model for short—, that can be used as input for model-checking tools. Correctness requirements are expressed in a dialect of the TCTL [HNSY94] logic. If we want to preserve decidability, this dictates restrictions on the expressiveness of the model. We need a formal definition of the *semantics* of this formalism to define the set of legal executions.

HTAs are based on preliminary work done by Wang and David [DY00], but apply some significant changes. In HTAs, both entries and exits of superstates as immediate and non-blocking. In order to give a clear semantics, our abstract—and concrete XML—syntax contains strong well-formedness constraints. These do not weaken the expressiveness, but make the task of constructing models more tedious in practice. As a remedy for the user, a sufficiently smart editor can operate on a less restrictive input-language, that is pre-compiled to our XML syntax, while possibly asking the user to resolve all ambiguities on demand.

For the real-time aspect, we use constructs from timed automata theory, i.e., formal clocks, clock resets, and invariants.

The multitude of modeling elements in hierarchical structures makes it subtle to construct a both clean and usable model-checking engine. Algorithms formulated for simpler structures tend to be more reliable, better analyzed, and tuned for efficiency in many cases. To experiment with our formalism, we describe a translation of HTAs to (flat) UPPAAL timed automata, and use this as a test-bed for our formalism. To represent HTAs physically, we use a XML document type definition [XML], that shapes the internal model representation of a fictitious tool (nicknamed HUPPAAL).

Plan This report is organized as follows. Section 2 gives a informal description of our hierarchical timed automata model. Section 3 contains the formal syntax of HTAs. Section 4 gives a formal semantics for the HTA model. Section 5 describes our implementation of a flattening procedure. In Section 6 we exemplify our flattening procedure on a cardiac pacemaker model and give run-times for applying UPPAAL's model checking engine on the translation. Section 7 lists unresolved issues and future work. The Appendix contains the full document type definitions of (hierarchical) HUPPAAL and (flat) UPPAAL, and finishes with a small glossary.

2 Hierarchical Timed Automata: Informal Overview

This section contains an informal description of hierarchical timed automata. They find physical representation through the XML document type definition in Appendix A.

Elements of the HTA Structure

Hierarchical timed automata are basically hierarchical state machines, that can be put in parallel on various levels. The basic units of control are called *locations*, which may be *basic states* or *superstates*, i.e., itself a (hierarchical) state machine. In the latter case, the contained locations are called *substates*. At any point, a location is either *active* or *inactive*. Superstates can be of type *XOR* (where exactly one substate is active, if the superstate is active) or of type *AND* (where every substate is active if the superstate is active).

Transitions connect locations. If source or target of a transitions is a superstate, the transition connects to distinguished *entries* and *exits*. These are auxiliary structures and sometimes referred to as *pseudo-locations*. Pseudo-locations are different from ordinary (*proper*) locations, since they mark intermediated steps in a more complex transition, and cannot be part of a proper configurations, see below.

Transitions can be equipped with *guards*, *assignments* and at most one *synchronization*. Transitions are enabled, if their guards evaluate to **true** (in the current configuration), the synchronization (if any) is possible and they can reach a target configuration. A transition is not enabled, if taking it would lead to a configuration where a location invariant is violated.

As auxiliary constructs, *pseudo-transitions* (⟨connector⟩s[1]) are used. At least one end of a pseudo-transition connects to a pseudo-location. Pseudo-transitions cannot be augmented with synchronizations, but in special situations may carry guards and/or assignments, as explained in Section 3 in detail.

Integer variables are shared (multi-read, multi-write), and may occur in guards and assignments. As a real-time construct, hierarchical timed automata are equipped with *clocks*. Clocks are understood as real-valued variables that change continuous and synchronous as time passes. Clocks can be reset to 0 on transitions, but not set to specific values. Clock values can occur in guards in a syntactically restricted fashion[2]. They can also occur in invariants, but only *downwards closed*, i.e., either as an expression $x < c$ or $x \leq c$, where c is an integer constant.

Hand-shake communication exits between parallel superstates by means of sending (!) or receiving (?) a signal on a *channel*. Two parallel automata can synchronize on transitions by executing them at the same instant. If they are equipped with conflicting variable assignments, the one of the transition labeled with "!" is executed first. Channels may be declared locally, restricting the potential participants in a hand-shake communication. We have to assure, that the control situation remains valid after processing both transitions. A

[1]The ⟨**ElementName**⟩ notation refers to the Element names in Appendix A.

[2]To be more precise, clocks x and y may occur in expressions $x - y \backsim c$ and $x \backsim c$, where c is an integer constant and $\backsim \in \{<, >, =, \leq, \geq\}$.

transition t originating in a superstate S cannot synchronize with a transition inside S, for processing t corresponds to rendering S inactive.

There is a top level, where a parallel composition of *fundamental superstates* is specified. They are understood as running in parallel, but *not* put together in an *AND* superstate for ease of usage: we assume, that system designers will frequently change this part, e.g., to test a controller with respect to different environments put in parallel. Therefore, this parallel composition is realized textually via the ⟨*system*⟩ tag. For the fundamental superstates, an initial entry has to be declared (⟨*globalinit*⟩). Moreover, they are allowed to *terminate* if specified so (`canexit` attribute in ⟨*globalinit*⟩), i.e., they can reach a special halt situation that can never be revoked.[3]

In the following, we describe entry and exit of superstates.

Entries Every superstate has a set of entries, that build part of its interface to the outside world. They are denoted by a stub, or alternatively, by a bullet: •. Transitions connect to superstates via entries.

Default Entry Optionally, a superstate S can have one entry, that is declared to be the *default entry*. If a transition on the next higher level ends at the border of S, without pointing to an explicit entry, it is assumed to lead to this default entry. In the case that no default entry is declared, such a transition is an error in the model.

History Entry A superstate may be declared to be a history-superstate, by equipping it with a special history entry, designated by a capital H in a circle, Ⓗ. If the superstate is entered via this, the last control location this superstate was in (before it became inactive) is restored. Additionally, all locally declared variables are restored. The locally declared clocks are *not* reset, but kept running. Only clocks explicitly declared as `forgetful clock` are set to 0 on entry via a history entry.

A history entry has to be equipped with a *default history location*, which is entered, if this is the first time the superstate becomes active. This location may be non-basic itself.

Every non-basic substate of S of a history superstate H is constrained to have either a history entry or a default-entry. If H is entered via the history entry, and the control points to S, then S is entered either via its history entry, or via the default entry, if S has no history entry.

There is no explicit *deep history entry*, that guarantees to instantiate the history of all enclosed substates as well. However, this can be expressed explicitly, by adding a history entry to all such substates and their descendants.

Forks Forks split the control to parallel substates. A fork can carry assignments and clock resets, but no guards or synchronization. Forks may trigger a cascade of other forks, that

[3] In general, this can violate deadlock freedom. However, it corresponds very much to a situation where a part of the system simply crashes. This aspect is useful, if the model explicitly specifies redundancy.

are all part of the same transition step. For simplicity, we treat here every entry of an *AND* superstate as a fork, i.e., it is required to have an outgoing transition to every substate and these transition are understood to be taken in parallel.

Local Clocks Clocks may be declared local to a superstate S. The first time S becomes active, these clocks are set to 0. On re-entry, local clocks are re-set to 0 as well, with one exception: ordinary local clocks are *not* re-set, if S is entered via an history entry. They can be thought of as *kept running* when S becomes inactive. Their value increases in accordance to the global clock. In general, it is not predictable whether it will be re-entered via a history entry or not.

The local clocks declared to be `forgetful clock` are always reset on entry, even this happens via a history entry.

Location Invariants Transitions t to locations carrying an *invariant* can only be taken, if the invariant evaluates to **true** (after possible clock resets executed along t). This generalizes in the situation, where a transition points to a non-basic location: it can only be taken, if the invariants of all reached locations (in case of a fork, there can be several) evaluate to **true** after executing the run-to-completion step.

Exits Explicit exits are denoted by a stub, or—alternatively—by a bullseye (⦿). Pseudo-transitions leading to an exit can only be taken, if the transition step associated with it can be taken as a whole. (This is in conformance with the UML notion of run-to-completion steps.) For notational convenience, various copies of explicit exits can be present in the same superstate. They are identified by sharing the same name.

As a well-formedness constraint, every exit that is reached by a transition, has to be connected to a transition or pseudo-transition on the next-higher level.

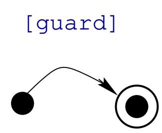


Figure 1: Default Exit.

Default Exits The understanding of a default exit is a specially designated exit, that can be reached either *unconditionally* or *guarded* from every enclosed location. This implies, that all non-basic substates are required to have default exits as well. If the guard is identical to **true**, this explicitly denotes a superstate to be *interrupt-able*, since it can be left in any case (provided it can synchronize on exit with parallel substates; typically, one of them will trigger the interrupt).

From the inside, they are *not* visible in general. But they can be indicated by an unlabeled general exit, see Figure 1.

Joins Joins are auxiliary constructs in *AND* superstates, that move control upward one level, after all substates became inactive. A join can carry guards, but no synchronization, clock resets, or assignments. Joins may be required to synchronize with other joins, that are all part of the same transition. In our notion, either all or none of them are taken.

We make the simplifying assumption, that every join can be associated with exactly one exit. Therefore we treat exits of *AND* superstates as joins.[4]

A *configuration* describes a snapshot of the system. In particular, every configuration

1. marks every location of the system as active or inactive
2. denotes one control location for every active *XOR* superstate
3. defines a value for every global variable and clock, and every local variable and clock of active superstate
4. defines a value for every local variable and local clock for every active superstate and the inactive ones, that contains a history entry

We call a configuration *proper*, if it does not contain pseudo-locations. A *run-to-completion step* is a tuple consisting of a proper source configuration, a step (that is either a proper transition or a sequence containing one proper transition and arbitrary many pseudo-transitions), and a proper target configuration (that is reached from the source configuration via execution of this step).

Dynamics of Transitions

An *execution step* of the model is either an *action step* or a *delay step*. An action step corresponds to executing one run-to-completion step, or—in case of synchronization—two synchronizing run-to-completion steps in an atomic fashion. A run-to-completion step is composed from one proper transition and arbitrary many pseudo transitions. The latter ones can, e.g., encode forks, joins, entries, or exits of substates. A run-to-completion step is only enabled, if

1. all the guards in the participating transition parts evaluate to **true**
2. the invariant(s) of the subsequent target location(s) hold after execution of assignments and clock resets

Syntactic restrictions guarantee, that 1. is always equivalent to the case, that the conjunction of these guards are true.

A delay step amounts to incrementing all clock variables by a real number $d > 0$, such that no invariant is violated.

Synchronization on Entry If an *AND* superstate is entered, every substate is entered immediately. This might trigger a cascade of entries, since substates are allowed to be *AND* superstates themselves.

[4]Our implementation deals with the more general case, that every exit can be reached by an arbitrary number of joins. However, the semantics is easier to desrcibe for this simplified setting, that does not restrict the expressiveness of our modeling formalism.

Synchronization on Exit A tree of joins is understood as an indivisible step, i.e., once it is started, it is executed, including the transition following immediately, called *root transition* of this join. There are no interleavings with other transitions or time delays. If the root transition synchronizes with another transition t, both are taken in parallel.

Pseudo-transitions to exits are allowed to have guards, but no assignment, clock-resets or synchronization labels. This guarantees that, given the conjunction of the guards evaluates to **true** in this configuration, the join can be executed to completion.

Urgency Urgency is a property of transitions and marks them as having priority over delay. If an urgent transition is enabled, the system is not allowed to delay, but must take an action transition as the next step.

Urgency cannot be only be used to resolve conflicts between action transitions and delay transition. An urgent action transition does not have priority over a non-urgent one, if both of them are enabled.

Lax Input Language

For notational convenience, it makes sense to allow a user to draw statechart diagrams in a more liberal way. In most cases, this can be safely translated to an explicit formulation. Some examples of this are given in Figure 2. Note that arrows on the left-hand side are sometimes replaced by sequences, that contain pseudo-states (stubs), pseudo-transitions, and exactly one ordinary transition. This is the one, where guards, assignments and synchronizations are attached. Following the UML notion of run-to-completion steps, the understanding of the explicit notation is identical with the (usual) interpretation of the lax notation.

In case of ambiguity, we expect a model editor to be clever enough to resolve the choices explicitly. In the following we always assume to have the explicit format, for this makes the task of formalizing the semantics easier.

3 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.

3.1 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.

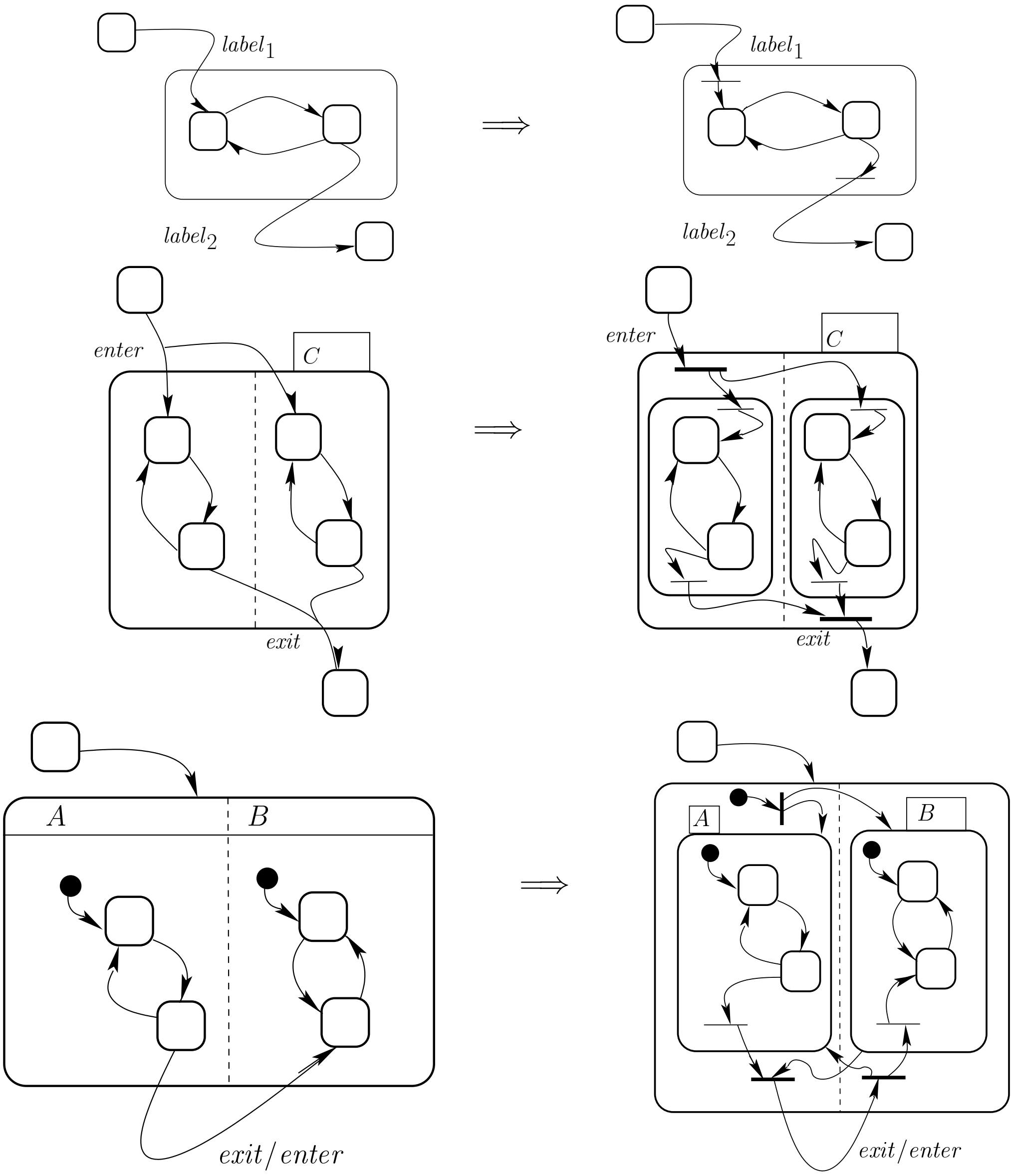


Figure 2: Translation of a lax entry formulation to the explicit form.

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.

3.2 Structural Components

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

Definition 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 3.1.
- $Inv : S \to Invariant$ 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 $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, $l \in S$. 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

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

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 $\delta^\times(l) \stackrel{def}{=} \delta^*(l) \setminus \{l\}$.
We introduce $\tilde{\delta}$ to refer to the children, that are proper locations.

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

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.

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

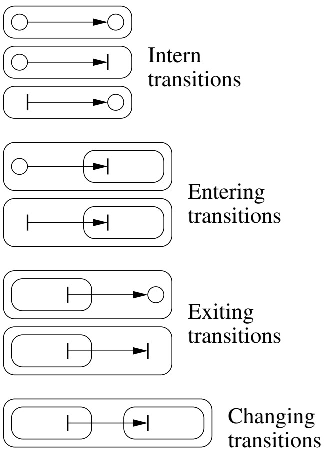
Entry constraints Let $e \in S$, *HENTRY*(e). If *XOR*$(\delta^{-1}(l))$, then T contains exactly one transition $e \xrightarrow{r} l'$. If *AND*$(\delta^{-1}(l))$, then T contains exactly one transitions $e \xrightarrow{r} e_i$ for every proper substate $l_i \in \tilde{\delta}(\delta^{-1}(l))$, and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

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 $l \rightarrow l'$.



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 $l \xrightarrow{g,s,r,u} l'$.

4 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 *Active*$(l) \stackrel{def}{=} l \in \rho^{\times}(root)$, where $\rho^{\times}(l)$ is the set of all active sub-states of l. Notice that *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 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 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 $\neg Active(l)$ 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 $\mathcal{C}(root)$.

History We capture the existence of a history entry with the predicate $\mathit{HasHistory}(l) \stackrel{def}{=} \exists n \in \delta(l).\ \mathit{HISTORY}(n)$. 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 $\mathit{Targets}_\theta : 2^S \to 2^S$ relative to θ.

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

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 *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

$\theta^1_{state} := \theta_{state}[\forall h \in H.\ h \mapsto \mathit{HEntry}(\rho(h))]$ and
$\theta^1_{var} := \theta_{var}[\forall h \in H.\ h \mapsto \mu(h)]$.

If $\neg$*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 *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 $t : l \xrightarrow{g,s,r,u} l'$:

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

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 *PreExitSets*(l). Each element X of *PreExitSets*(l) is itself a set of exits. If transitions are enabled to all exits in X, then all substates can synchronize.

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

Here, the operator $\boxtimes : (2^{2^S}) \times (2^{2^S}) \to 2^{2^S}$ is a product over families of sets, i.e., it maps $(\{A_1,\dots,A_a\},\{B_1,\dots,B_b\})$ to $\{A_1 \cup B_1, A_1 \cup B_2, \dots, A_a \cup B_b\}$ 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:

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

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

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

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.

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

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

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

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.

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

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.

5 From Hierarchical Timed Automata to Uppaal

In this section we give a detailed description of our flattening procedure, that translates our hierarchical timed automaton model to to a parallel composition of (flat) Uppaal timed automata [LPY97]. For both models we have a syntactic representation via a XML document type definition. The Document Type Definition in Appendix A describes the HTA syntax. Appendix B contains the document type definition for Uppaal timed automata syntax. The latter serves as a valid input format for the Uppaal verification tool from version 3.2 on. Our flattening procedure translates XML documents according to Appendix A to XML documents according to Appendix B. It is implemented in Java, making use of XML technology [XML] to ease lexing and parsing. Code and Java documentation can be found at `http://www.brics.dk/~omoeller/hta/vanilla-1/`. Since this amounts to 9359 lines of documented code, we strive to give a concise description here.

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 model size. 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.

Outline of the Flattening Algorithm The basic concept of the procedure is the translation of instantiated templates. For every superstate occurring in the HTA model, one Uppaal template is constructed. However, this cannot be done in an transducer fashion. Since parallel states synchronize on exit, information about exits depends on other parts, that may not have been translated yet.

Thus the translation has three phases: collection of instantiations, computation of global joins, and post-processing channel communication.

For sake of clarity, we choose to omit various thinkable optimizations. For example, *XOR* substates of *XOR* superstates or *AND* substates of *AND* superstates are not lifted, even if there are no local variables on the lower levels.

5.1 Phase I: Collection of Instantiations

In this phase, the (implicit) hierarchical instantiation tree is traversed and for every hierarchical superstate, the skeleton of a (flat) template is constructed.

Initially, the direct children of the root are on the stack, i.e., the fundamental superstates as contained in the $\langle system \rangle$ element.

How exactly the superstates I are translated is dependent on their type, that is either *XOR* or *AND*.

Algorithm: *PHASE I: instantiateTemplates*

input: Stack S of superstates to translate
output: Set T of (flat) timed automata
Set GJ of global join starting points

$T := \{$ *Global_Kickoff* automaton for $s \in S\}$
$GJ := \varnothing$
WHILE *notempty*(S)
  $I := pop(S)$
  $\mathcal{C} := \{$non-basic locations in $I\}$
  FORALL $c \in \mathcal{C}$
    push$(S, [c \text{ in } I])$
    /⋆ $[c$ in $I]$ inherits all invariants attached to I ⋆/
    create a location $\hat{c}$ in $\hat{I}$
    $E_c := \{$set of entries of c in $I\}$
    FORALL $e \in E_c$
      create a committed location $\hat{c_e}$ in $\hat{I}$
      create a transition from $\hat{c_e}$ to $\hat{c}$ in $\hat{I}$
      /⋆ this transition carries a synchronization "`enter_`$\hat{c}$`_via_`e`!`" ⋆/
    IF *type*$(I) = XOR$ THEN
      $GJ := GJ \cup \{c \text{ in } I\}$

  $T := T \cup \{$translation $\hat{I}$ of superstate I, depending on *type*$(I)\}$

XOR:
- we have basic locations and transitions
- possibly, it contains superstates (⟨*component*⟩s)
- there is at least one ⟨*entry*⟩
- there might be ⟨*exit*⟩s, possibly declared default
- entries are connected to locations or entries of substates
- exits are reached from locations or exits of substates

***AND*:**
- there are no basic locations, no transitions
- there are two or more ⟨*component*⟩s
- there is at least one ⟨*entry*⟩
- entries correspond to ⟨*fork*⟩s
- there may be ⟨*exit*⟩s
- exits correspond to ⟨*join*⟩s

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 substate S of X we introduce a location S_ACTIVE_IN_X in $\hat{X}$. In Figure 6, the *XOR* superstate **X** has only one substate **S**. **X** and **S** are translated to the two timed automatons in Figures 9 and 10.[5]

Moreover, for every entry e of S we introduce an auxiliary location in $\hat{X}$, 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 substate S.

Exits of this substate S are more complicated, for they are only possible, if all non-basic substates of S can exit. This is described as global joins, see Section 5.2.

If superstate X is inactivated, this is realized in the translation $\hat{X}$ 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 $\hat{A}$ via a transition from A_ACTIVE to A_IDLE, which carries the synchronization signal exit_A.

5.2 Phase II: Computation of Global Joins

Transitions originating from superstates are a subtle issue, for they may require a cascade of substate exits—called *global join*—in order to be taken.

[5]Vanilla-1 uses the un-prefixed component name Detail instead of the equivalent to S_ACTIVE_IN_X, because the length of names is limited.

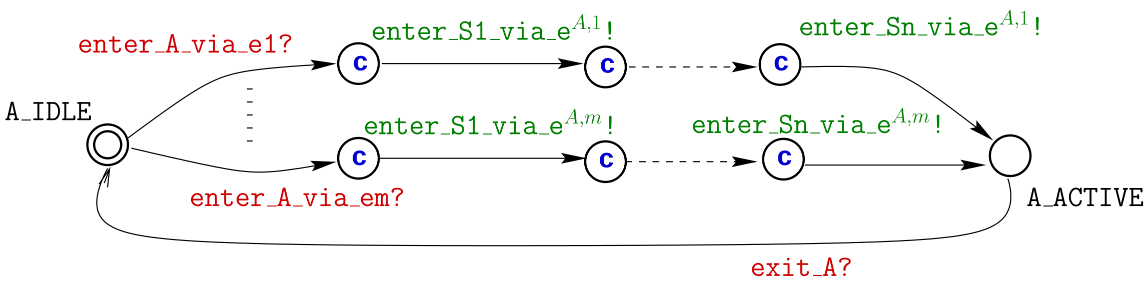
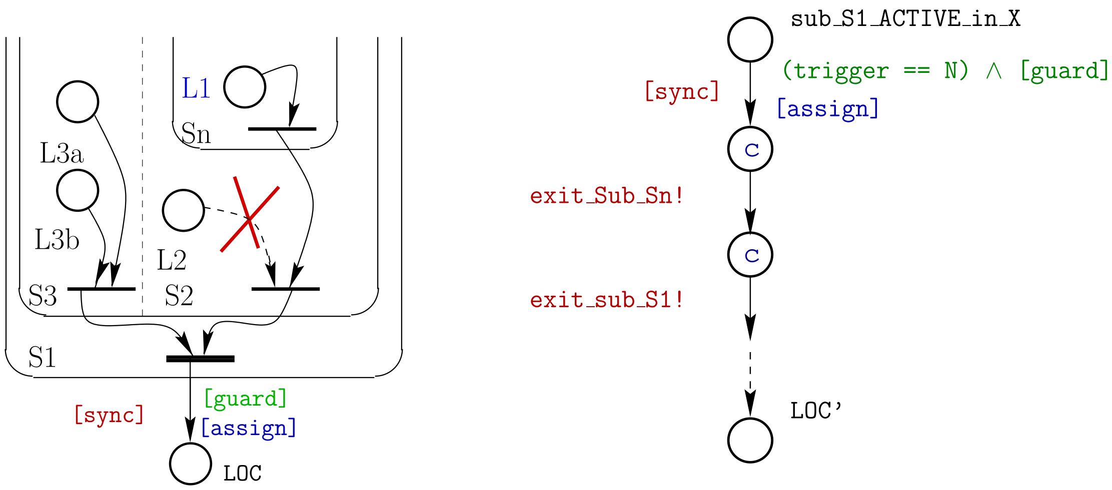


Figure 3: Translation of entering and exiting an *AND* component.



(a) Part of hierarchical timed automaton X *(b) Translation a the global join in $\hat{X}$*

Figure 4: The exit of `S1` in superstate X gives rise to a number of global joins.

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. In phase I of our flattening algorithm, the output GJ collects the topmost components, that have to be exited, if a transition (like to `LOC` in Figure 4) has to be translated. One entry in GJ 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 substate `Sn`.

Every possible global join is translated to a sequence like 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. 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, see 5.3. Roughly this can be described with the pseudo-code *expandGlobalJoins*.

5.3 Phase III: Post-processing Channel Communication

If a transition in the hierarchical timed automaton formalism starts at a non-basic state S and carries a synchronization, it cannot synchronize with a transition *inside* S. Since the substate/superstate relation is lost in the translation, we have to resolve this scope conflict explicitly. In Vanilla-1 we do this by introducing duplications of channels and transitions.

We start with a priority queue Q over transitions that possibly can cause a conflict. These elements were collected during the construction of the global joins. Q is sorted obeying the partial order introduced by the substate/superstate relation on instantiations. Then the

Algorithm: *PHASE II: expandGlobalJoins*
input: Set GJ of global join starting points
output: Auxiliary constructions: counters and guarded transitions

$JoinTrees := \varnothing$
FORALL $gj \in GJ$
  collect all trees t of control locations, that can synchronize to gj;
  the leaves of t are sets of basic locations, that share transitions to exits x.
  /⋆ These sets are singletons, if x is an ordinary exit and span over all basic locations in the superstate otherwise ⋆/
  $JoinTrees := JoinTrees \cup \{t\}$
FORALL $tree \in JoinTrees$
  let $\hat{L} := \{\hat{l} \mid l$ is element in a basic location set of $tree\}$
  declare the counter $trigger_{tree}$
  FORALL $\hat{l} \in \hat{L}$
    FORALL transitions $\hat{k} \to \hat{l}$
      add the assignment $trigger_{tree} := trigger_{tree} + 1$ to $\hat{k} \to \hat{l}$
    FORALL transitions $\hat{l} \to \hat{m}$
      add the assignment $trigger_{tree} := trigger_{tree} - 1$ to $\hat{l} \to \hat{m}$
  let $N :=$ number of leaf sets in $tree$
  let $\mathcal{S}_{tree} :=$ substates occurring in $tree$
  FORALL transition t starting at $root(tree)$
    create a chain of transitions, starting with $\hat{t}$,
      corresponding to exiting every $s \in \mathcal{S}_{tree}$
    /⋆ see Figure 4 (b); note the additional guard $trigger_{tree}$`==`N ⋆/

post-processing can be described as in the pseudo-code snip-let *postprocessChannels*.

5.4 Correctness of the Translation

Starting at the root level, we can define a correspondence between every legal global state of the HTA model and its translation into UPPAAL timed automata.

Every superstate S in the hierarchical timed automaton model corresponds exactly to one UPPAAL timed automaton $\hat{S}$. For proper configurations, we can relate ρ in the hierarchical timed automaton model to a *control vector* $\hat{\rho}$ in the UPPAAL model. For an UPPAAL automaton U, $\hat{\rho}(U)$ denotes the active location of U. 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 $\rho(A) = \bot$ and $\hat{\rho}(\hat{A}) = \{\hat{S} \mid S$ parallel substate of $A\}$ otherwise. The value of the introduced auxiliary variables is completely determined by the current control location, i.e., it is redundant for the configuration and only serves to enable or disable transitions.

Algorithm: *PHASE III: postprocessChannels*
input: *priorityQueue* Q over (*syncSignal*, *transition*, *instantiation*)

WHILE notempty(Q)
  (*syncSignal*, *transition*, *instantiation*) := *pop*(Q)
  IF $\exists$ transition t with *match*(*syncSignal*) in *instantiation*:
    create a new channel c
    replace *channel*(*syncSignal*) on *transition* by c
    FORALL transitions t' with *match*(*syncSignal*) outside *instantiation*
      create a copy of t', where *channel*(*syncSignal*) is replaced by c
      if there is an entry of t' in Q, add an entry for the created copy to Q

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 legal 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.

6 Example: Translation of a Cardiac Pacemaker

In this section we apply our flattening procedure on a hierarchical timed automaton version of a cardiac pacemaker model. This model is strongly motivated by the often-used UML design example, see e.g. [Dou99]. The pacemaker is put in parallel with a model of a human heart and a programmer, who changes operation modes on the pacemaker. We translate the hierarchical timed automaton model of this composition to an equivalent (flat) UPPAAL timed automata model and explain the obtained automata in detail. Additionally, we report on run-time data of the formal verification of this translation with respect to safety and response properties.

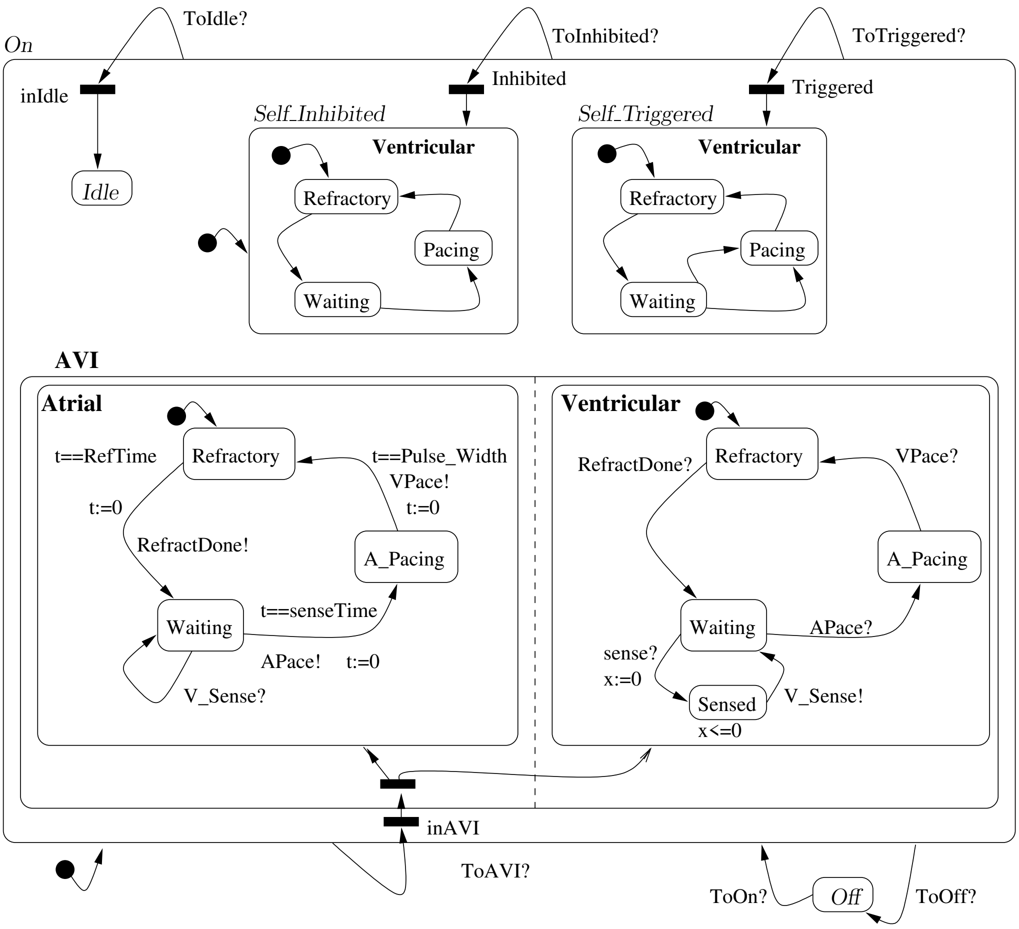


Figure 5: Overview of a HTA model describing a functional component from a cardiac pacemaker, notice the basic location *Off* and the superstate *On*. Initially, the VVI mode (ventricular, self-triggered) is entered.

6.1 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. Thus, in our example only the ventricular chamber is observed, but a pace signal may be sent either to the ventricular or articular chamber.

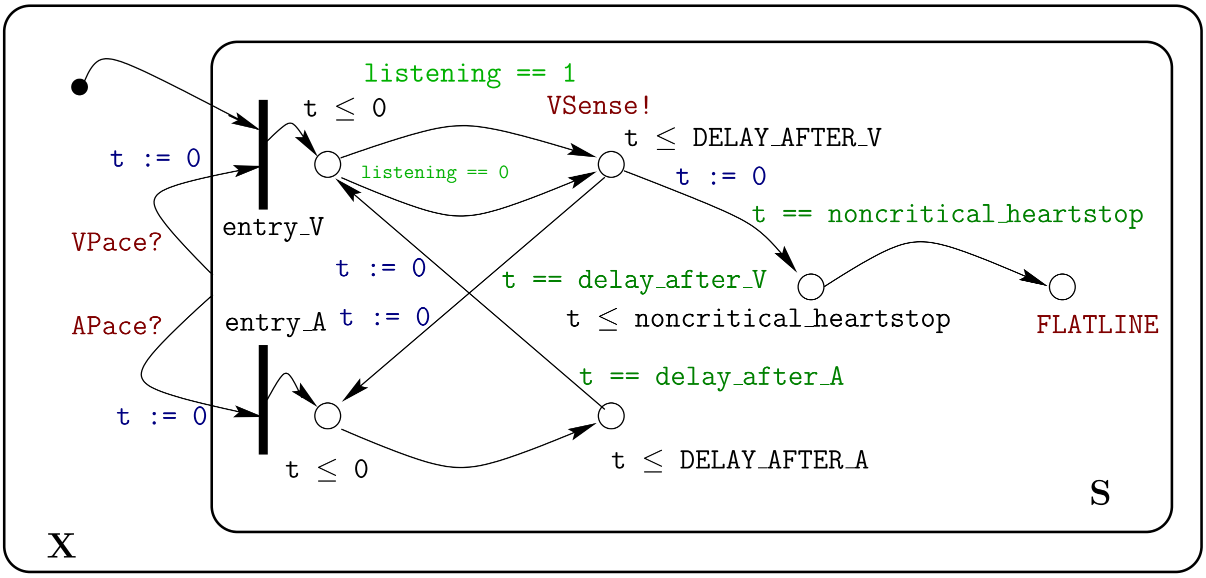


Figure 6: A simplified model of the human heart.

Heart Model. We use a simplified model of a human heart, that might require pacing (Figure 6). 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.

The 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.

Programmer Model. The signals `commandedOn!`, `commandedOff!`, `toIdle!`, `toVVI!`, `toVVT!`, and `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. The programmer is modeled by a two-state machine. In the first state, `Modeswitch`, any signal can be issued while entering the second state. The second state is left after exactly `DELAY_AFTER_MODESWITCH` time units. We introduced two extra states *Random* and *Idle*, to encode alternative behavior that is not relevant here.

Additionally, we allow the programmer to terminate at some point, i.e., reach a specific stop state that can never be revoked. We use our `global exit` construction for this. Termination is possible, whenever the programmer is in the `Modeswitch` state.

6.2 Translation to UPPAAL Timed Automata

In the HTA model, the `Programmer`, `Heart`, and `Pacemaker` are put in parallel. Only the `Programmer` is allowed to terminate.

In the translation, this yields

- one automaton to start the three parts (Figure 7)
- one automaton for the `Programmer` (Figure 8)
- two automata for the `Heart`, a top-level (Figure 9), where exit and re-entry happens and one for the substate (Figure 10), where the heart is beating
- seven automata for the `Pacemaker`, put together as
  - one automaton for the top-level (Figure 11), where the pacemaker is either *On* or *Off*
  - one automaton for the VVI operation mode (Figure 13)
  - one automaton for the VVT operation mode (Figure 14)
  - three automata for the AVI operation mode, one for the *AND* superstate (Figure 15) and two for the substates listening to the ventricular chamber (Figure 16) and pacing the articular chamber (Figure 17)

Over all, the increase in terms of model size was noticable, but moderate. Table 2 lists this data in detail. A large number of committed locations were introduced to encode entry and global joins. However, these forks and joins are triggering a deterministic sequence of actions and thus do not significantly increase the state space. A similar observation holds

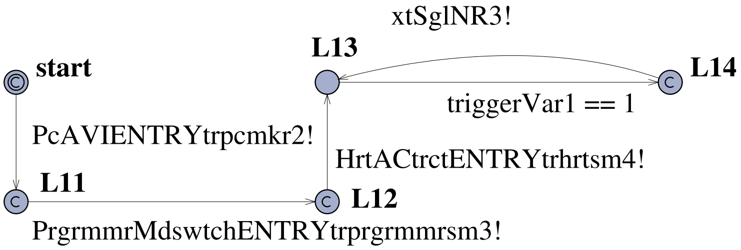


Figure 7: The `KickOff` automaton starts the three fundamental superstates. The transitions on the far right correspond to the `Programmer` becoming inactive.

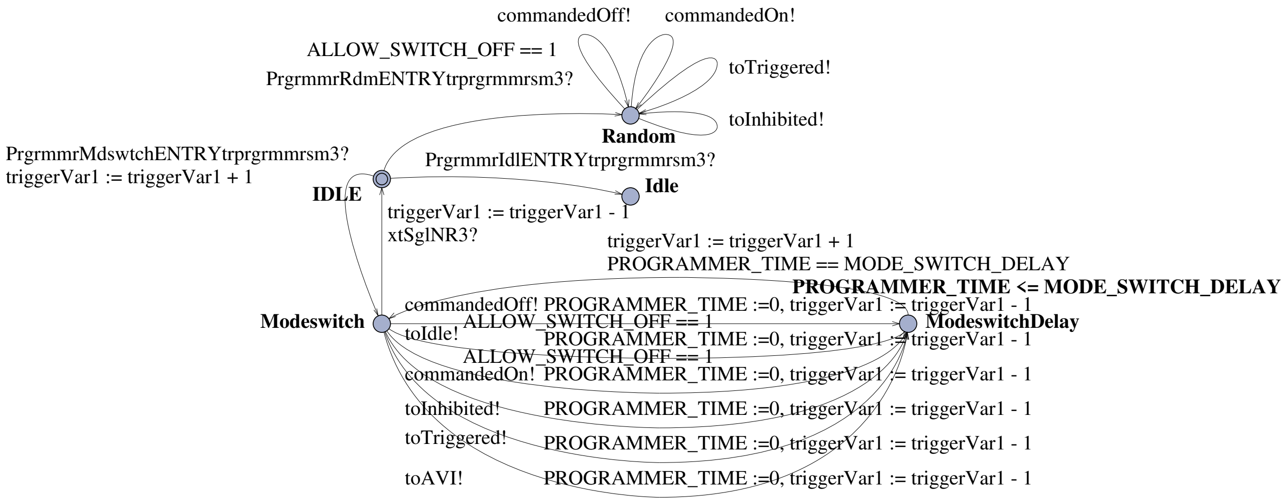


Figure 8: Translation of the *XOR* superstate `Programmer`. In our context, the entry is set to the `Modeswitch` state, where it is possible to issue control signals and move to the `ModeswitchDelay` state. The programmer can terminate itself by taking the transition to `IDLE`.

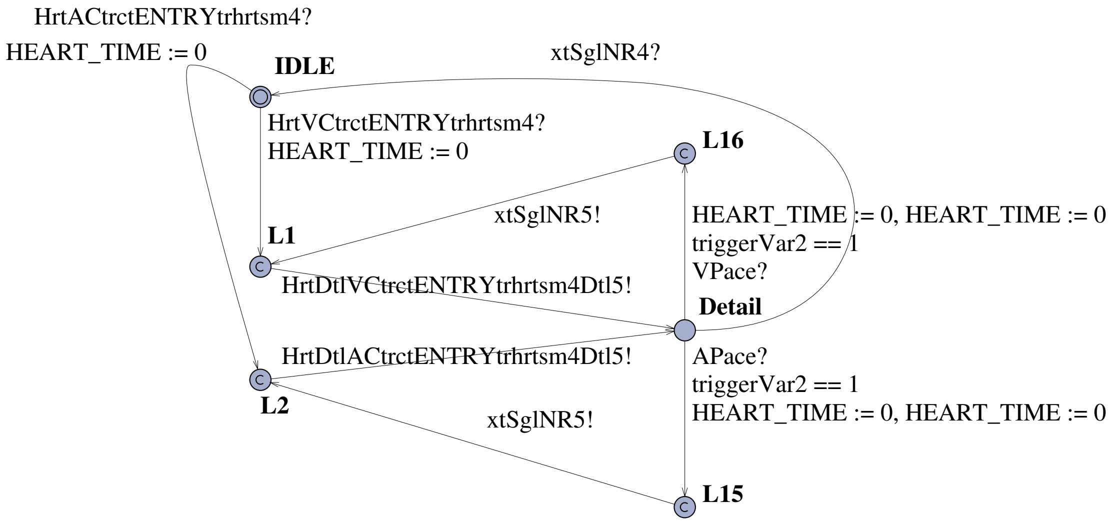


Figure 9: Translation of the *XOR* top level superstate modeling the heart. Here we find the entries and exits triggered by `APace?` and `VPace?`.

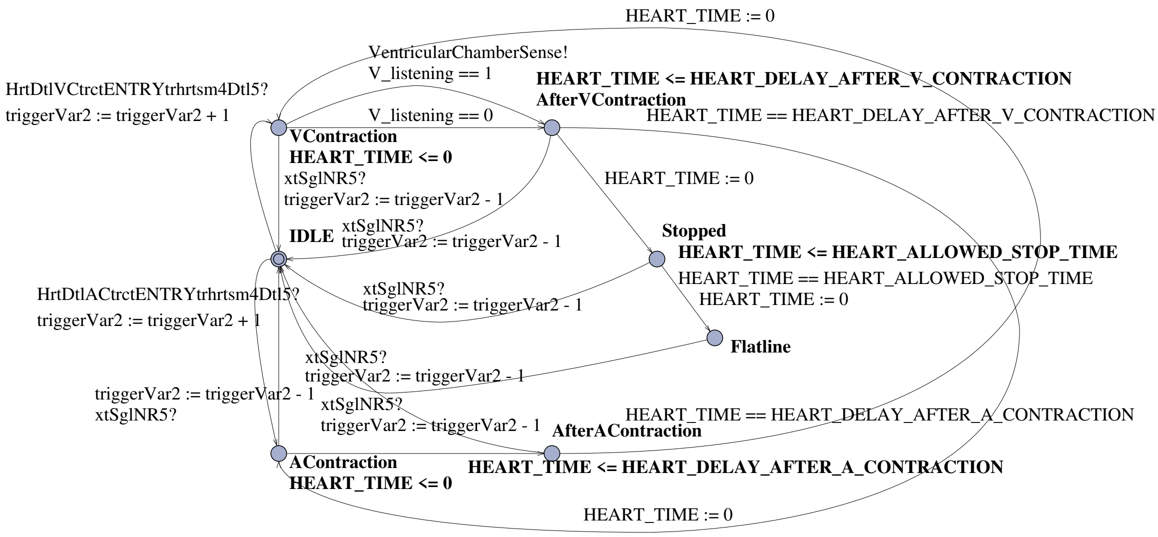


Figure 10: Translation of the *XOR* superstate, that encodes the beating heart. It can nondeterministically take a transition to *Stopped* and—after some time delay—further to `Flatline`. The exit transition to `IDLE` is triggered by a signal from the automaton in Figure 9.

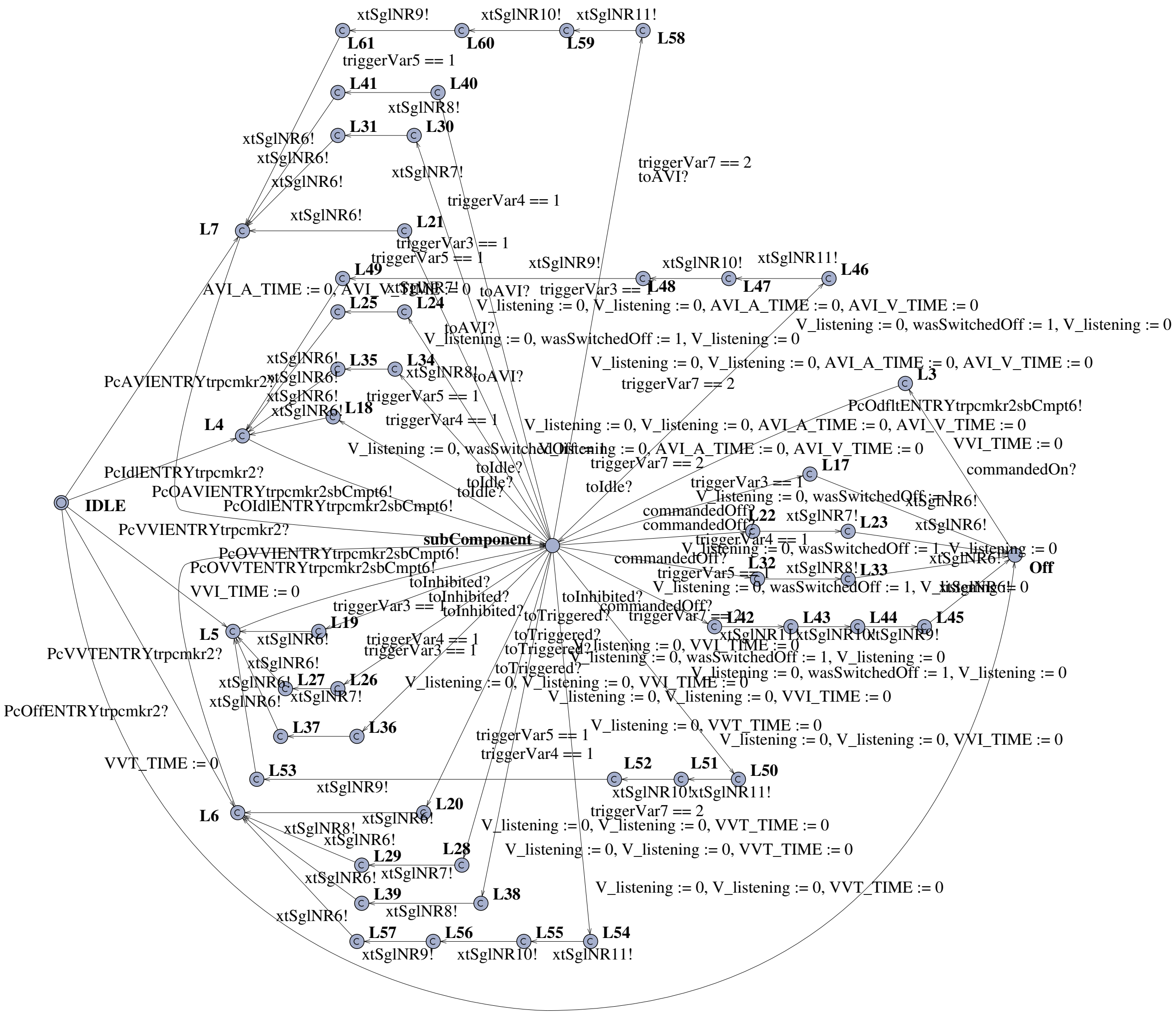


Figure 11: Translation of the *XOR* superstate, that captures the topmost level of the pacemaker. You can distinguish four entry pseudo-states on the left: the locations `L4`, `L5`, `L6`, and `L7` are correspond to entering the modes `Idle`, `VVIMode`, `VVTMode`, and `AVIMode`. The pacemaker is *on*, when it control resides in `subComponent` and *off*, when the control is at `Off` (far right). The outgoing chains from `subComponent` correspond to global joins and lead to (committed) entry pseudo-states.

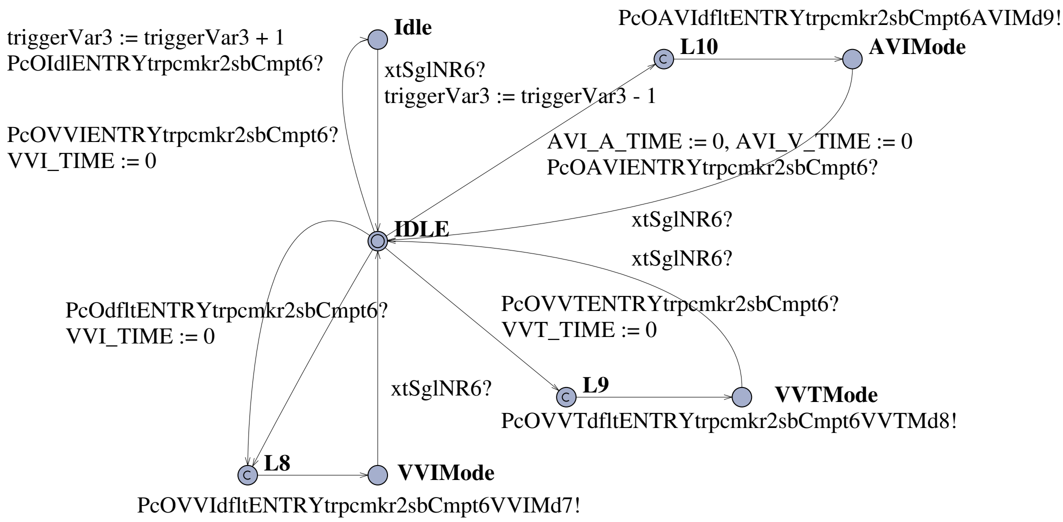


Figure 12: Translation of the *XOR* superstate, where the pacemaker is active. It can reside in the locations `Idle`, `VVIMode`, `VVTMode`, and `AVIMode`. There are no direct transitions between these modes, the superstate has to be exited to change in between them.

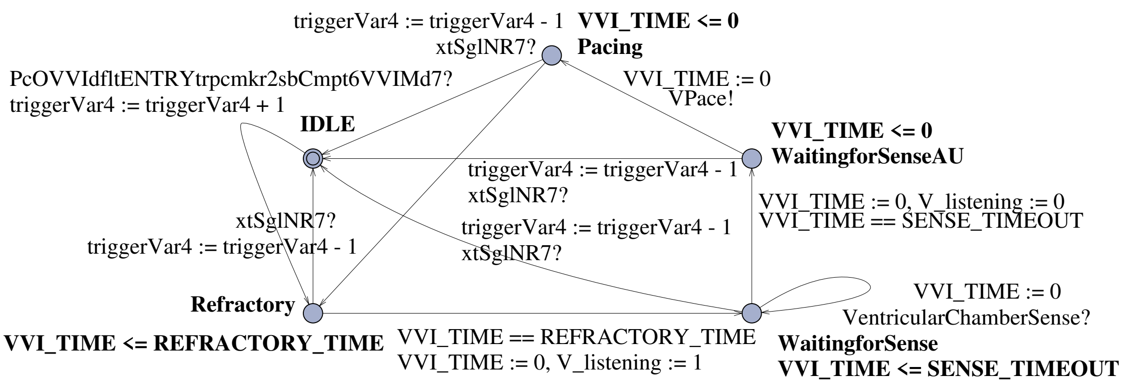


Figure 13: Translation of the *XOR* superstate corresponding to the VVI Mode.

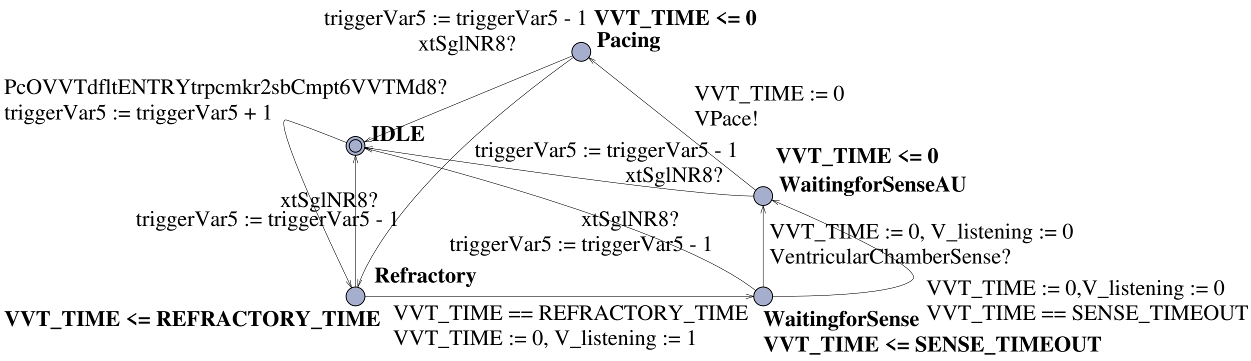


Figure 14: Translation of the *XOR* superstate corresponding to the VVT Mode.

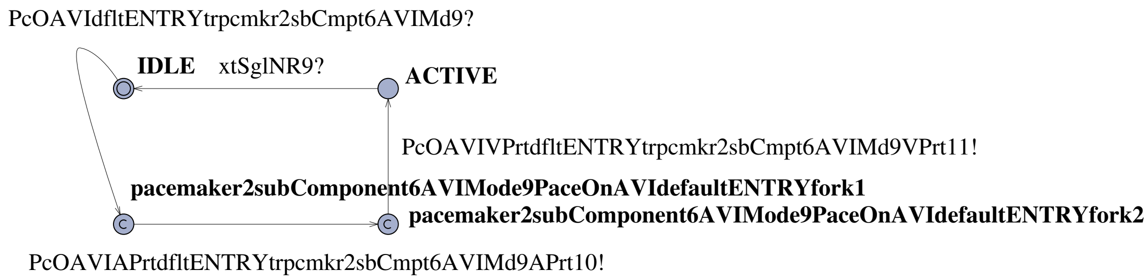


Figure 15: Translation of the *AND* superstate corresponding to the AVI mode. Merely the two substates AVI-A and AVI-V are activated.

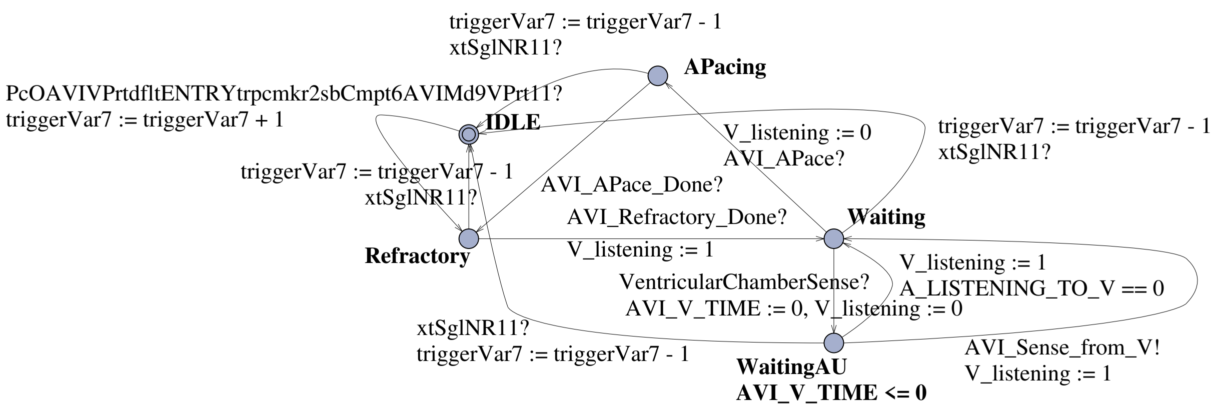


Figure 16: Translation of the *XOR* superstate AVI-V.

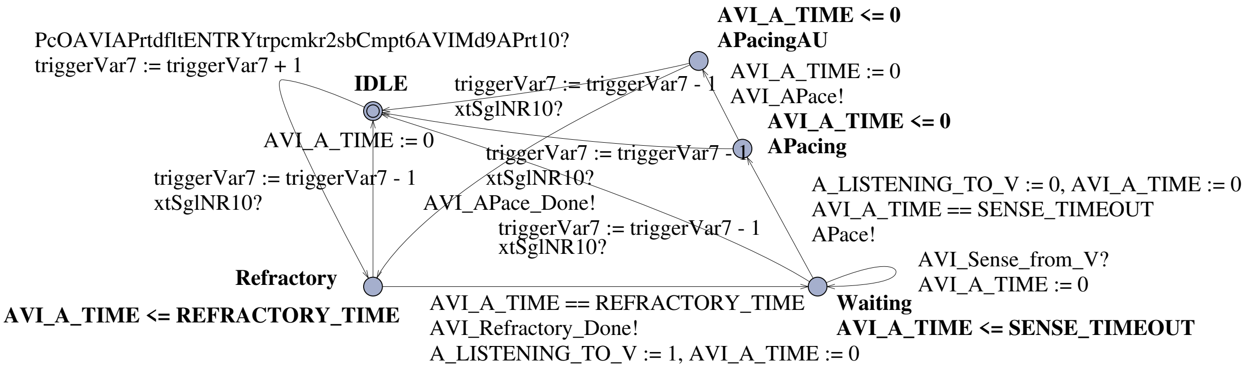


Figure 17: Translation of the *XOR* superstate AVI-A.

	HTA model	UPPAAL model
# XML tags	564	1191
# proper control locations	35	45
# pseudo-states / committed locations	33	63
# transitions	47	177
# variables and constants	33	72
# formal clocks	6	6

Table 2: Translations of a hierarchical timed automaton description to an equivalent flat UPPAAL model. Both data formats are described in terms of XML grammars.

for the introduced auxiallary variables: The values of variables triggering global joins are completely determined by the current control state. The auxillary channels introduced to switch components from `IDLE` to `ACTIVE` and vice versa does not increase the complexity significantly.

6.3 Model-Checking the UPPAAL Model

We used the translation as input to the UPPAAL tool. The system as described is not deadlock free: when the programmer terminates after switching off the pacemaker, and the heart stops beating, a configuration is reached where time can delay indefinetley. In one variation, the programmer was explicitly disallowed to exit. In a second variation, the pacemaker could not be switched off. In both variations, deadlock freedom was established via a run of the model-checking engine on a true invariant with switch settings `-Aa` (convex hull approxiation and active clock reduction switched on), and took 3.50 respectively 1.75 seconds.

We verified two desirable properties in the (non-variated) 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 states, 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
```

Figure 18: Constants that yield property *(i)*.

The latest version of the UPPAAL tool[6] is able to perform the model-checking of both properties successfully in 11.83 repectively 4.26 seconds. The verification of the typically more expensive property *(ii)* is faster, since here it is possible to apply a property preserving convex hull over-approximation, that is not preservative with respect to property *(i)*. We use 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 gives drastic improvements in model-checking time in this example.

It is worthwhile to mention, that validity of property *(i)* is strongly dependent on the parameter setting of the model. We use the constants from Figure 18. 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. In related work, an extended version of UPPAAL is used to derive parameters yieling property satisfaction automatically, see [RS-01-5].

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

7 Summary

We defined a hierarchical timed automaton formalism and equipped it with a formal semantics in terms of a transition system. We present a translation to Uppaal timed automata.

Our formalism is realized via the XML grammar in Appendix A. We implemented our translation procedure in Java as a transformation from XML documents according to Appendix A to XML documents according to Appendix B. The later is an input format to the Uppaal tool. Our experimental data indicates, that the overhead introduced in the translation is tolerable in terms of model size and run-times of the model-checking engine on the translation.

The XML document type definition in A is designed to be flexible and extensible. It is based on a template/instantiation mechanism and uses purely textual data for parts that supposedly change frequently during model design, like the elements $\langle \mathit{declaration} \rangle$ or $\langle \mathit{system} \rangle$. It is intended to eventually replace the Uppaal document format, thus the hierarchical features are optional. In fact, if a document does not contain a $\langle \mathit{component} \rangle$ element, the only difference between a document of type grammar Appendix A and a document of type grammar Appendix B is, that the former declares initial states via $\langle \mathit{globalinit} \rangle$ element, whereas the latter uses the $\langle \mathit{init} \rangle$ tag.

Future Work

Though we have a working prototype for a grammar and a translation, it is to be considered work in progress. We made the implementation accessible for future reference as frozen version at `http://www.brics.dk/~omoeller/hta/vanilla-1/`.

The translation `Vanilla-1` is documented as a milestone to make experiments with. It is not able to translate some powerful modeling constructs, though they are already present syntactically.

Unresolved Issues in Vanilla-1 are in particular local declarations, scope overriding, history entries, synchronization mechanisms other than handshake communication, and parameterized templates.

In near future, it is planed to implement an editor for the hierarchical grammar in the Uppaal tool. Simulation and verification of hierarchical models, however, are done on flat Uppaal timed automata, constructed by future versions of Vanilla-1.

There is a strong correspondence between hierarchical and flat traces. However, the imperative of introducing fresh and unambiguous names for flattened constructs makes it difficult for a human user to see this immediately, compare Section 6. One possible remedy for this is to equip the Uppaal simulator with the appropriate mapping, so it can display names as specified in the hierarchical system. We feel that it is also necessary to provide a translation of TCTL formulas to corresponding ones in the flattened version. This seems to be purely syntactical, but strongly dependent on the mapping of local and global variables.

Looking ahead, we believe that there is a great potential for exploiting the hierarchical structure directly in terms of shaping more efficient model checking algorithms. Since we want this to work in practice, we consider it crucial for this enterprise to get the hierarchical modeling formalism right.

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/`.

[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.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

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

[DY00] Alexandre David and Wany Yi. Hierarchical Timed Automata. unpublished draft, dated: April 23. Contact the authors `adavid@DoCS.uu.se`, `yi@DoCS.uu.se`, 2000.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[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.

[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.

[XML] Extensible Markup Language (XML) 1.0 (Second Edition), according to the W3C Recommendation 6 October 2000, see http://www.w3.org/TR/REC-xml.

A The XML grammar huppaal-0.6

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Tentative hierarchical document definition                   -->
<!--                                                              -->
<!-- Synopsis:                                                    -->
<!--  XML, hierarchical Uppaal                                    -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- @TABLE OF CONTENTS:                 [TOCD: 16:30 20 Feb 2001] -->
<!--                                                              -->
<!--  [1] modified elements here                                  -->
<!--  [2] classical things below                                  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      -->
<!-- @FILE:    huppaal-0.6.dtd                                    -->
<!-- @FORMAT:  XML Document Type Definition                       -->
<!-- @AUTHOR:  M. Oliver M'o'ller     <omoeller@brics.dk>         -->
<!-- @BEGUN:   Wed Oct 18 14:02:43 2000                           -->
<!-- @VERSION  V0.6   Sun Apr  8 18:18:50 2001                    -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

<!ELEMENT hta (imports?, declaration?, template+, instantiation?, system,
               globalinit*)>
<!ELEMENT imports (#PCDATA)>
<!ELEMENT declaration (#PCDATA)>
<!-- notes on template:
 1. components seem to be a generalization of locations;

-->
<!ELEMENT template (name,
                    parameter?,
                    declaration?,
                    entry+,
                    fork*,
                    exit*,
                    join*,
                    location*,
                    component*,
                    transition*
                      )>
<!ATTLIST template type CDATA #IMPLIED> <!-- NN -->
<!-- the type is intended to be "XOR" (default) or "AND"
"XOR" means:
 * we have locations and transitions
 * possibly, we have 'components' as generalisations of transitions
 * there is an <entry>
 * there might be an <exit>
```

```
 * there is no fork
 * there is no join

"AND" means:
 * there are no locations, no transitions
 * there are two or more <component> tags
 * there is a fork
 * there is a  <entry>
 * every <entry> points to a <fork>, thus there is a fork
 * every fork/join connects *every* component, that is present
 * optionally, there is an <exit> location
 * optionally, there is a join

  -->
<!-- "history" is not specified explicitly in this grammar.
     The reason for this is, that having a history is not an isolated
     property, but requires a designated <entry> that has the
     attribute type="history".
     this will serve as a defintion. -->

<!-- the instantiates attribute refers to the (textual) NAME of the template
     (this makes imports easier)
     The optional labels include invariants, and (possibly) comments.  -->
<!ELEMENT component (name, label*)>
<!ATTLIST component instantiates   CDATA #REQUIRED
                    withparameters CDATA #IMPLIED
                    id             ID    #REQUIRED
                    x              CDATA #IMPLIED
                    y              CDATA #IMPLIED>
<!-- A connection from here has no source; the target is the location of fork
     it leads to.
     Multiple connections correspond to non-deterministic branching.
     The type-attribute is uses to denote entries history (or default?)  -->
<!ELEMENT entry (name, connection*, entrypoint?)>
<!ATTLIST entry id       ID    #REQUIRED
                type     CDATA #IMPLIED
                x        CDATA #IMPLIED
                y        CDATA #IMPLIED>

<!-- if an entrypoint is defined, it *replaces* the entry graphically, if the
     inside of the component is shown
     (thus, the transition from it is nicer to display).
     The notation for this is a small bar (stub) with outgoing arrow and
     a name
     (no name means "default entry"; an alternative notation for this is a
     small bullet)
     There can be at most one entrypoint per entry -->
<!ELEMENT entrypoint EMPTY>
<!ATTLIST entrypoint  x        CDATA #IMPLIED
                      y        CDATA #IMPLIED>
```

```
<!-- connections from a fork do not have source, but only a target -->
<!ELEMENT fork (name?, connection*)>
<!ATTLIST fork  id       ID    #REQUIRED
                x        CDATA #IMPLIED
                y        CDATA #IMPLIED>

<!-- the connection to an exit does not have a target, only a source;
     the type-attribute is used to declare an exit "default-exit"
     default-exits are only allowed in XOR templates.
-->
<!ELEMENT exit (name?, connection*, exitpoint*)>
<!ATTLIST exit  id       ID    #REQUIRED
                type     CDATA #IMPLIED
                x        CDATA #IMPLIED
                y        CDATA #IMPLIED>

<!-- exitpoints are a notational convenience, and semantically identical
     with the exit they point to (exit attribute).
     there can be arbitrary many exitpoints corresponding to the same exit -->
<!ELEMENT exitpoint EMPTY>
<!ATTLIST exitpoint  id       ID    #REQUIRED
                     x        CDATA #IMPLIED
                     y        CDATA #IMPLIED>

<!-- connections to joins do not have a target, only a source -->
<!ELEMENT join (name?, connection*)>
<!ATTLIST join  id       ID    #REQUIRED
                x        CDATA #IMPLIED
                y        CDATA #IMPLIED>

<!-- point to an entry in a global system component, i.e., an instantiation
     and an entry (ref) of it.
     the attibute CDATA is by default "no", other values are
     "all" or "specified".
     In the "all" case, every exit is a possible starting point.
     In the "specified" case, the connection elements point to the relevant
     exits and can also carry synchronisations/guards/assignments.
-->
<!ELEMENT globalinit (connection)*>
<!ATTLIST globalinit instantiationname CDATA #REQUIRED
                     ref               IDREF #IMPLIED
                     canexit           CDATA #IMPLIED>

<!-- ~~ connections, aka pseudo-transitions ~~~~~~~~~~~~~~~~ -->
<!ELEMENT connection (source?, target?, label*, nail*)>
<!ATTLIST connection type CDATA #IMPLIED
                     x    CDATA #IMPLIED
                     y    CDATA #IMPLIED>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      -->
<!-- [1] modified elements here                                       -->
```

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      -->


<!-- if source points to a component, then
     exitref: points to an exit or exitpoint
     of the template the component instantiates -->
<!ELEMENT source EMPTY>
<!ATTLIST source ref      IDREF #REQUIRED
                 exitref  IDREF #IMPLIED>

<!-- if target points to a component, then
     entryref: points to an entry or entrypoint
     of the template the component instantiates -->
<!ELEMENT target EMPTY>
<!ATTLIST target ref      IDREF #REQUIRED
                 entryref IDREF #IMPLIED>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      -->
<!-- [2] classical things below                                           -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      -->

<!ELEMENT name (#PCDATA)>
<!ATTLIST name x   CDATA #IMPLIED
               y   CDATA #IMPLIED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter x   CDATA #IMPLIED
                    y   CDATA #IMPLIED>
<!ELEMENT location (name, label*, urgent?, committed?)>
<!ATTLIST location id ID #REQUIRED
                   x  CDATA #IMPLIED
                   y  CDATA #IMPLIED>
<!ELEMENT urgent EMPTY>
<!ELEMENT committed EMPTY>

<!-- kind: "assignment", "guard", "synchronisation", "invariant" -->
<!ELEMENT label (#PCDATA)>
<!ATTLIST label kind CDATA #REQUIRED
                x    CDATA #IMPLIED
                y    CDATA #IMPLIED>
<!ELEMENT nail EMPTY>
<!ATTLIST nail x   CDATA #REQUIRED
               y   CDATA #REQUIRED>
<!ELEMENT instantiation (#PCDATA)>
<!ELEMENT system (#PCDATA)>

<!ELEMENT transition (source, target, label*, nail*)>
<!ATTLIST transition x   CDATA #IMPLIED
                     y   CDATA #IMPLIED>
```

B The XML grammar uppaal-1.4

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- dtd distributed by Gerd on 19 Feb 2001 as part of             -->
<!-- Uppaal-3.1.39                                                 -->
<!--                                                               -->
<!-- Now with <label> tag, using JAXP 1.1  (final version)         -->
<!--                                                               -->
<!-- Synopsis:                                                     -->
<!--  XML, hierarchical Uppaal                                     -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- @FILE:    uppaal-1.4.dtd                                      -->
<!-- @FORMAT:  XML Document Type Definition                        -->
<!-- @AUTHOR:  Gerd Behrmann    <behrmann@cs.auc.dk>               -->
<!-- @BEGUN:   Wed Feb 19 14:52:05 2001                            -->
<!-- @VERSION: Mon Feb 19 20:45:53 2001                            -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

<!ELEMENT nta (imports?, declaration?, template+, instantiation?, system)>
<!ELEMENT imports (#PCDATA)>
<!ELEMENT declaration (#PCDATA)>
<!ELEMENT template (name, parameter?, declaration?, location*, init?,
                    transition*)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name x   CDATA #IMPLIED
               y   CDATA #IMPLIED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter x   CDATA #IMPLIED
                    y   CDATA #IMPLIED>
<!ELEMENT location (name?, invariant?, urgent?, committed?)>
<!ATTLIST location id ID #REQUIRED
                   x  CDATA #IMPLIED
                   y  CDATA #IMPLIED>
<!ELEMENT init EMPTY>
<!ATTLIST init ref IDREF #IMPLIED>
<!ELEMENT invariant (#PCDATA)>
<!ATTLIST invariant x   CDATA #IMPLIED
                    y   CDATA #IMPLIED>
<!ELEMENT urgent EMPTY>
<!ELEMENT committed EMPTY>
<!ELEMENT transition (source, target, label*, nail*)>
<!ATTLIST transition x   CDATA #IMPLIED
                     y   CDATA #IMPLIED>
<!ELEMENT source EMPTY>
<!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target ref IDREF #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ATTLIST label kind CDATA #REQUIRED
                x    CDATA #IMPLIED
```

```
                  y     CDATA #IMPLIED>
<!ELEMENT nail EMPTY>
<!ATTLIST nail x    CDATA #REQUIRED
               y    CDATA #REQUIRED>
<!ELEMENT instantiation (#PCDATA)>
<!ELEMENT system (#PCDATA)>
```

Glossary

configuration A configuration is a snapshot of the system, where every location is either active or inactive and every variable and clock is set to *one specific* value. A configuration is *proper*, if all active basic locations are proper.

entry A pseudo-location, that is passed to activate the corresponding superstate.

entry point Copy of an entry, displayed as bullet (•), annotated with the name of the entry.
This is a notational/graphical convenience with the semantics of an alias.

exit A pseudo-location, that is passed to inactivate the corresponding superstate.

exit point Copy of an exit, displayed as bullseye (⦿).
This is a notational/graphical convenience with the semantics of an alias.

fork Auxiliary structure used in *AND* superstates.
A fork connects an entry of the superstate with the entries of the parallel substates, thus activating them.

global join Synchronous exit of various parallel superstates.
A global join gives rise to a *tree* of joins (connected via exits), that is specific to a *root transition* (the one that is executed immediately after the join).
A global join can only be started, if all participants can synchronize on their exit. It is executed without interruption, including the execution of the root transition. (In the UPPAAL translation, this requires special constructions, see Section 5.2.)

join Auxiliary structure used in *AND* superstates.
A join connects exits from each of the enclosed superstates with an exit of the superstate itself.

location The basic unit of control.
A location can be *basic* or a *superstate*, i.e., itself a hierarchical timed automaton. Basic locations are either *proper* or *pseudo-locations*. At any time, a location is either *active* or *inactive*.

pseudo-location Auxiliary location to encode complex transitions.
Though physically a (committed) location, this does usually not correspond to a state the modeled system can be in and exists solely for modeling purposes, typically to encode forks, joins, or multi-synchronization.

pseudo-transitions Auxiliary transition to encode a part of a run-to-completion step, e.g., to encode entry, exit, or multi-synchronization.
Pseudo-transitions are connected to at least one committed location. Restrictions on allowed guards, assignments, and synchronizations apply.

pre-exit A location that has a transition to an exit.

run-to-completion step A sequence of transitions, containing one proper transition and arbitrary many pseudo-transitions.
This amounts to a macro-transition leading from one proper configuration to to a subsequent proper configuration.

superstate A non-basic location.
We distinguish *XOR* superstates (exactly one of the substates is active, if the superstate is active) and *AND* superstates (parallel composition: all substates are active, if the superstate is active).

transition A transition connects two locations, carrying guards, assignments, and synchronization.
If these are non-basic, the transition connects to specific entries or exits. A transition is either *proper* or *pseudo*.

Recent BRICS Report Series Publications

RS-01-11 Alexandre David and M. Oliver Möller. *From* HUPPAAL *to* UPPAAL*: A Translation from Hierarchical Timed Automata to Flat Timed Automata*. March 2001. 40 pp.

RS-01-10 Daniel Fridlender and Mia Indrika. *Do we Need Dependent Types?* March 2001. 6 pp. Appears in *Journal of Functional Programming*, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.

RS-01-9 Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *Static Validation of Dynamically Generated HTML*. February 2001. 18 pp.

RS-01-8 Ulrik Frendrup and Jesper Nyholm Jensen. *Checking for Open Bisimilarity in the π-Calculus*. February 2001. 61 pp.

RS-01-7 Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo. *On the Number of Quasi-Kernels in Digraphs*. January 2001. 11 pp.

RS-01-6 Gregory Gutin, Anders Yeo, and Alexey Zverovich. *Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*. January 2001. 7 pp.

RS-01-5 Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. *Linear Parametric Model Checking of Timed Automata*. January 2001. 44 pp. To appear in Margaria and Yi, editors, *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.

RS-01-4 Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. *Efficient Guiding Towards Cost-Optimality in* UPPAAL. January 2001. 21 pp. To appear in Margaria and Yi, editors, *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.