Verification of UML Statechart with Real-time Extensions

Alexandre David
M. Oliver Möller
Wang Yi

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2003-009
Februari 2003
ISSN 1404-3203

VERIFICATION OF UML STATECHART WITH REAL-TIME EXTENSIONS

Alexandre David[*] M. Oliver Möller[†]
Wang Yi[*]
[*]*Department of Information Technology, Uppsala University, Sweden*
{adavid,yi}@docs.uu.se,

[†]BRICS *Department of Computer Science, Aarhus University, Denmark*
omoeller@brics.dk.

Abstract. We develop a formal model for hierarchical timed systems. The statechart-like hierarchy features parallelism on any level and connects superstate and substate via explicit entries and exits. Time is represented by clocks, invariants, and guards. For this formalism we give an operational semantics that is appropriate for the verification of universal timed computation tree logic (TCTL) properties.

Our model is strongly related to the timed automata dialect as present in the model checking tool UPPAAL. Here networks of timed automata are enriched with shared variables, hand-shake synchronization, and urgency.

We describe a flattening procedure that translates our formalism into a network of UPPAAL timed automata. This flattening preserves a correspondence of the sets of legal traces. Therefor the translation can be used to establish properties in the hierarchical model.

As a case study, we use the standard UML modeling example of a cardiac pacemaker. We model it in our hierarchical language, flatten it to UPPAAL input, and use the latter for a formal analysis.

Our formalism remains decidable with respect to TCTL properties. In general the encoding of statecharts requires an abstraction step, which is not covered by this article.

1. Introduction

The correct both concurrent and real-time. Any one of these features already complicates the design, for basic descriptions may entail unforeseen behaviors. This suggests to include concerns for correctness a priori, before a prototype of a system is built.

In model-based development this requires appropriate modeling languages that describe the system under development on a high level. If this model should be used for an analysis of the system, it needs a formal semantics and machinery to support this analysis. Since early design model undergo frequent changes, automation in the analysis is not only desirable but a prerequisite.

For most reasonably expressive modeling languages, even basic properties are undecidable, which prevents fully automated treatment in general.

However, under appropriate safe abstractions the analysis might be able to establish or refute a relevant sub-set of the actual system properties.

For reactive systems, Harel and Pnueli suggest to use hierarchical state-machines with parallelism on various levels as an appropriate modeling language [HP85]. The properties could be expressed in dialects of temporal logics.

However, standard statechart formalisms typically use event queues for communication, which renders reachability an undecidable problem. Moreover, the timing properties are usually a second class citizen in the sense, that timeout events are used to generate timing conditions.

What we propose is to include time prominently in a formalism that is structurally close to statecharts and features a less powerful synchronization mechanism than events. Properties of this formalism, which are chosen from a real-time version of temporal logics, should remain decidable. The necessary abstraction step from a "real" statechart design model then could be carried out on the level of data-abstraction, where a rich tradition in the framework of abstract interpretation exists [CC77].

Following this idea we define a formalism for timed systems that is halfway between UML statecharts and UPPAAL timed automata. Basically we extend timed automata with a statechart-style hierarchy and parallelism on any level. The resulting language is described by a formal syntax and given a operational semantics. Considering the rich set of existing formal statechart-like languages—including several timed variations—, the introduction of yet another formalisms might come as a surprise. It is motivated along two dimensions.

First, we are primarily concerned with the *formal analysis* of models in our language. In particular, we plan to pursue a model checking approach that is powerful enough to capture the complete behavior of a system with respect to a timed logic. To deal with the high computational complexity, we strive to benefit from the intensive research on the timed automata model. This dictates to restrict our formalism to decidable primitives that moreover allow for reasonable efficiency in the exhaustive analysis of a system.

Second, the multitude of variations in the statechart formalism makes the choice of one formalism not easier. No two variations we know of are comparable. We note a trend to treat statecharts as a programming language close formalism, e.g., by attaching **C**++ code to states and transitions. It is conceivable that algorithmic treatment of this requires an abstraction step. The anchor of our formalism is the possibility for fully automatic analysis. As a price, the translation of other formalisms into it might have to be an *abstraction function*. This still allows for a faithful analysis with respect to, e.g., safety properties.

Thus our language is structurally close to full-featured statechart formalisms and conceptually close to timed automata. The former is incorporated, e.g., by the RHAPSODY tool, and the latter by the real-time model checking tool UPPAAL.

Plan. This article is organized as follows. In Section 2 we introduce our timed statechart-like formalism, called hierarchical timed automata. In Section 3 we give the (flat) timed automata formalism that can serve as an input to the model checking tool Uppaal. In Section 4 we define a subset of TCTL that can be effectively used for model-checking. This is appropriate both for the hierarchical and for the flat timed model. In Section 5 we give a description of a flattening procedure that translates hierarchical timed automata into an equivalent flattened network. In Section 6 we sketch the correctness of this translations, in the sense that both hierarchical and flattened model satisfy a common set of TCTL properties. We implemented this flattening procedure for a XML representation of both formalisms. In Section 7 we use the model of a cardiac pacemaker as a case study. In Section 8 we give concluding remarks.

2. Hierarchical Timed Automata

We fist give an informal introduction and then define the syntax of our formalism. Next we present the operational semantics.

2.1 Syntax of Hierarchical Timed Automata

Hierarchical Timed Automata (HTAs) are motivated by the statechart formalism of [Har87]. As the main syntactic restriction the event communication is replaced by a less expressive hand-shake synchronization. This is necessary to maintain decidability.

We introduce the syntax of HTAs first intuitively and then by a formal definition.

2.1.1 A Restricted Statechart Formalism

Since we are primarily interested in formal verification, we restrict the rich and expressive UML statechart formalism. Timed behavior is reflected by (formal) clocks, timed guards, and invariants. Our goal is to tailor a formalism where essential properties remain decidable.

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

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

What we preserve is the essence of the statechart formalism: hierarchical structure, parallel composition at any level, synchronization of remote parts, and history.

2.1.2 Data Components

We introduce the data components of hierarchical timed automata that are used in guards, synchronizations, resets, and assignment expressions. Some of this data is kept local to a superstate S.

Integer variables. Let *Var* be a finite set of integer variables. $\mathit{Var}(S) \subseteq \mathit{Var}$ is the set of integer variables local to a superstate S.

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

Channels. Let *Chan* a finite set of synchronization channels. $\mathit{Chan}(S) \subseteq \mathit{Chan}$ is the set of channels that are local to a superstate S, i.e., there cannot be synchronization along a channel $c \in \mathit{Chan}(S)$ between one transition inside S and one outside S.

Synchronizations. *Chan* gives rise to a finite set of channel synchronizations, called *Sync*. For $c \in \mathit{Chan}$, $c?$, $c! \in \mathit{Sync}$.

Guards and invariants. A data constraints is a boolean expressions of the form $E \bowtie E$, where E is an arithmetic expression over *Var* and $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraints is an expressions of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in \mathit{Clocks}$ and $n \in \mathbb{Z}$ with $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraint is downward closed, if $\bowtie \in \{<, =, \leq\}$. 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 $x \in \mathit{Clocks}$. A data assignment is of the form $v := E$, where $v \in \mathit{Var}$ and E an arithmetic expression over *Var*. *Reset* is the set of clock resets and data assignments.

2.1.3 Structural Components

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

Definition 1. (Hierarchical Timed Automaton (HTA))
A hierarchical timed automaton is a tuple $\langle \mathcal{S}, \mathcal{S}_0, \eta, \mathit{type}, \mathit{Var}, \mathit{Clocks}, \mathit{Chan}, \mathit{Inv}, T \rangle$ *where*

- $\mathcal{S}$ *is a finite set of locations.*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is a set of initial locations.*
- $\eta : \mathcal{S} \to \wp(\mathcal{S})$. η *maps* S *to all possible substates of* S. η *is required to give rise to a tree structure where a special superstate* $\mathit{root} \in \mathcal{S}$ *is the root. We readily extend* η *to operate on sets of locations in the obvious way.*

- *$type : \mathcal{S} \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ is the type function for locations. Superstates are of type AND or XOR.*
- *Var, Clocks, Chan are sets of variables, clocks, and channels. They give rise to* Guard, Reset, Sync, *and* Invariant *as described in Section 2.1.2.*
- *$Inv : \mathcal{S} \to$ Invariant maps every locations S to an invariant expression, possibly to the constant* **true***.*
- *$T \subseteq \mathcal{S} \times (Guard \times Sync \times Reset \times \{\mathbf{true}, \mathbf{false}\}) \times \mathcal{S}$ is the set of transitions. A transition connects two locations S and S', has a guard g, an assignment r (including clock resets), and an urgency flag u. S is called the* source *and S' is called the* target *of the transition. We use the notation $S \xrightarrow{g,s,r,u} S'$ 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 $TYPE(S)$ for $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, $S \in \mathcal{S}$. E.g., $AND(S)$ is true, exactly if $type(S) = AND$. The type $HISTORY$ is a special case of an entry. We use $HENTRY(S)$ to capture simple entry or history entry, i.e., $HENTRY(S)$ stands for $ENTRY(S) \vee HISTORY(S)$.

We define the parent function

$$\eta^{-1}(S) := \begin{cases} b, \text{ where } S \in \eta(b) & \text{if } S \neq root \\ \bot & \text{otherwise} \end{cases}$$

We readily extend η^{-1} to operate on sets of locations, i.e., for $\mathcal{S}' \subseteq \mathcal{S}$: $\eta^{-1}(\mathcal{S}') := \{\eta^{-1}(S) \mid S \in \mathcal{S}'\}$. Furthermore, we use $\eta^*(S)$ to denote the set of all nested locations of a superstate S, including S. $\eta^{-*}(S)$ is the set of all ancestors of S, including S. Moreover we use $\eta^+(S) := \eta^*(S) \setminus \{S\}$.

We introduce $\tilde{\eta}$ to refer to the children, that are proper locations.

$$\tilde{\eta}(S) := \{b \in \eta(S) \mid BASIC(b) \vee XOR(b) \vee AND(b)\}$$

We use $Var^+(S)$ to denote the variables in the scope of superstate S: $Var^+(S) = \bigcup_{b \in \eta^{-*}(S)} Var(S)$. $Clocks^+(S)$ and $Chan^+(S)$ are defined analogously.

2.1.4 Well-Formedness Constraints

We give a set of well-formedness constraints to ensure consistency, grouped as for the syntactic categories locations, initial locations, variables, entries, and transitions.

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

(1) The function η gives rise to a proper tree rooted at $root$, and $\mathcal{S} = \eta^*(root)$.

(2) Only superstates contain other locations: $AND(S) \vee XOR(S) \Leftrightarrow \eta(S) \neq \varnothing$.

(3) Substates of *AND* superstates are not basic: $\mathit{AND}(S) \wedge b \in \eta(S) \Rightarrow \neg \mathit{BASIC}(b)$.

(4) No invariants on pseudo-locations: $\mathit{HENTRY}(S) \vee \mathit{EXIT}(S) \Rightarrow \mathit{Inv}(S) = \mathbf{true}$.

(5) For every superstate S, at most one exit can be declared to be the *default exit*. If existent, the default exit is reachable from every location in S.

Initial location constraints. $\mathcal{S}_0$ has to correspond to a consistent and proper control situation, i.e., $\mathit{root} \in \mathcal{S}_0$ and for every $S \in \mathcal{S}_0$ the following holds:

(1) $\mathit{BASIC}(S) \vee \mathit{XOR}(S) \vee \mathit{AND}(S)$,

(2) $S = \mathit{root} \vee \eta^{-1}(S) \in \mathcal{S}_0$,

(3) $\mathit{XOR}(S) \Rightarrow |\eta(S) \cap \mathcal{S}_0| = 1$, and

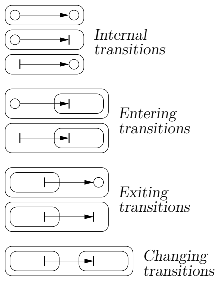
(4) $\mathit{AND}(S) \Rightarrow \eta(S) \cap \mathcal{S}_0 = \tilde{\eta}(S)$.

Variable constraints. We explicitly disallow conflict in assignments in synchronizing transitions:
It holds that $S_1 \xrightarrow{g,c!,r,u} S_2$, $S_1' \xrightarrow{g',c?,r',u'} S_2' \in T \Rightarrow \mathit{vars}(r) \cap \mathit{vars}(r') = \varnothing$, where $\mathit{vars}(r)$ is the set of integer variables occurring in r. We require an analogous constraint to hold for the pseudo-transitions originating in the entry of an *AND* superstate.

Static scope: For $S_1 \xrightarrow{g,s,r,u} S_2 \in T$, g, r are defined over $\mathit{Var}^+(\eta^{-1}(S_1)) \cup \mathit{Clocks}^+(\eta^{-1}(S_1))$ and s is defined over $\mathit{Chan}^+(\eta^{-1}(S_1))$.

Entry constraints. Let $e \in \mathcal{S}$, $\mathit{HENTRY}(e)$. If $\mathit{XOR}(\eta^{-1}(S))$, then T contains exactly one transition $e \xrightarrow{r} S'$. If $\mathit{AND}(\eta^{-1}(S))$, then T contains exactly one transition $e \xrightarrow{r} e_i$ for every proper substate $B_i \in \tilde{\eta}(\eta^{-1}(S))$, and $e_i \in \eta(B_i)$.
In case of $\mathit{HISTORY}(e)$, outgoing transitions declare the *default history locations*.
At most one entry of a superstate can be declared to be the *default entry*. If a superstate S has a history entry, then every substate B of S has to provide a history entry or a default entry.

Transition constraints. Transitions have to respect the structure given in η and cannot cross levels in the hierarchy, except via connecting to entries or exits. The set of legal transitions is given in Table 2.1. Note that transitions cannot lead directly from entries to exits. The internal transitions are those made inside one superstate: from a state to a state, from a state to an exit or from an entry to a state. The constraint expresses that the parent state must be the same. The entering transition is from a state to an entry and the fork transition is from an entry to an entry. The constraints express the transition to a nested state. The exiting and join transitions are symmetric to entering and fork. The changing transition is from the exit of a superstate to the entry of another superstate. The constraint states that both superstates must have a common parent.



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

Fig. 2.1: Overview on Legal Transitions $S \xrightarrow{g,s,r,u} S'$.

Transitions $S \xrightarrow{g,s,r,u} S'$ with $\mathit{HENTRY}(S)$ or $\mathit{EXIT}(S')$ are called *pseudo-transitions*. They are restricted in the sense that they cannot carry synchronizations or urgency flags, and only either guards or assignments. For $\mathit{HENTRY}(S)$, only pseudo-transition of the form $S \xrightarrow{r} S'$ are allowed. For $\mathit{EXIT}(S')$, only pseudo-transition of the form $S \xrightarrow{g} S'$ are allowed. For $\mathit{EXIT}(S) \wedge \mathit{EXIT}(S')$, this is further restricted to be of the form $S \rightarrow S'$.

2.2 Operational Semantics of HTAs

We define now the operational semantics of the hierarchical timed automaton formalism. Legal steps between configurations of a HTA give rise to a set of traces.

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

$\rho : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ captures the control situation. ρ can be understood as a partial, dynamic version of η that maps every superstate S to the set of active substates. If a superstate S is not active, $\rho(S) = \varnothing$. We define $\mathit{Active}(S) := S \in \rho^{+}(\mathit{root})$, where $\rho^{+}(S)$ is the set of all active substates of S. Notice that $\mathit{Active}(S) \Leftrightarrow S \in \rho(\eta^{-1}(S))$.

$\mu : \mathcal{S} \rightarrow (\mathbb{Z})^{*}$. μ gives the valuation of the local integer variables of a superstate S as a finite tuple of integer numbers. If $\neg\mathit{Active}(S)$ then $\mu(S) = \lambda$ (the empty tuple). If $\mathit{Active}(S)$ then we require that $|\mu(S)| = |\mathit{Var}(S)|$ and μ is consistent with respect to the value of shared variables (i.e., always maps to the same value). We use $\mu(S)(a)$ to denote the value of $a \in \mathit{Var}(S)$. 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 $0^{\mathit{Var}(S)}$ for the tuple $(0, 0 \ldots 0)$ with arity $|\mathit{Var}(S)|$.

$\nu : \mathcal{S} \rightarrow (\mathbb{R}_{\geq 0})^{*}$. ν gives the real valuation of the clocks $\mathit{Clocks}(S)$ defined locally to the superstate S, thus $|\nu(S)| = |\mathit{Clocks}(S)|$. If $\neg\mathit{Active}(S)$ then $\nu(S) = \lambda$.

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

$\theta_{state}(S)$ returns the last visited substate of S—or an entry of the substate, in the case where the substate is not basic—(to restore $\rho(S)$), and $\theta_{var}(S)$ 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 $\mathit{Clocks}(\mathit{root})$.

We call a configuration where all S in $\rho^+(\mathit{root})$ are of type *BASIC*, *XOR*, or *AND* a *proper configuration.*

History. We capture the existence of a history entry with the predicate $\mathit{HasHistory}(S) := \exists b \in \eta(S).\ \mathit{HISTORY}(b)$. If $\mathit{HasHistory}(S)$ holds, the term $\mathit{HEntry}(S)$ denotes the unique history entry of S. If $\mathit{HasHistory}(S)$ does not holds, the term $\mathit{HEntry}(S)$ denotes the default entry of S. If S is basic $\mathit{HEntry}(S) = S$. If none of the above is the case, then $\mathit{HEntry}(S)$ is undefined.

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

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

$$\mathit{Targets}_\theta(L) :=$$
$$L \cup \bigcup_{S \in L} \{b \mid b \in \theta_{state}(S) \ \wedge\ \mathit{HISTORY}(S)\} \cup \{b \mid S \xrightarrow{r} b \ \wedge\ \mathit{ENTRY}(S)\}$$

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

Configuration vector transformation. Taking a transition $t : S \xrightarrow{g,s,r,u} S'$ entails in general 1. executing a join to exit S, 2. taking the proper transition t itself, and 3. executing a fork at S'. If S (respectively S') is a basic location, part 1. (respectively 3.) is trivial. Together, 1–3 define a *proper step*. We represent a proper 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 $\rho^1 := \rho[\forall b \in \rho^+(S).\ b \mapsto \varnothing]$.
μ is updated to $\mu^1 := \mu[\forall b \in \rho^+(S).\ b \mapsto \lambda]$.
ν is updated to $\nu^1 := \nu[\forall b \in \rho^+(S).\ b \mapsto \lambda]$.

If $\mathit{EXIT}(S)$, the history is recorded. Let H be the set of superstates $h \in \rho^+(\eta^{-1}(S))$, 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 $\neg\mathit{EXIT}(S)$ or $H = \varnothing$, then $\theta^1 := \theta$.

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

and $r(\nu^1)$ the updated clock evaluation 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 $\mathit{Targets}^*_{\theta^2}(S')$. Note that $\rho^2(b) = \varnothing$ for all $b \in \eta^+(S')$. Thus we can compute ρ^3 as follows:

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

μ^3 is derived from μ^2 by first initializing all local variables of the superstates B in $\mathit{Targets}^*_{\theta^2}(S')$, i.e., $\mu^3(\mathit{Var}(B)) := 0^{\mathrm{Var}(B)}$. If $\mathit{HasHistory}(B)$, $\theta_{var}(B)$ is used instead of $0^{\mathrm{Var}(B)}$. 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 S and S' are basic locations. We define the configuration vector transformation $\mathcal{T}_t$ for a transition $t : S \xrightarrow{g,s,r,u} S'$:

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

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 $e \in \eta(S)$ we recursively define the family of sets of exits $\mathit{PreExitSets}(e)$. Each element E of $\mathit{PreExitSets}(e)$ is itself a set of exits. If transitions are enabled to all exits in E, then all substates can synchronize.

$$\mathit{PreExitSets}(e) :=$$

$$\begin{cases} \left.\begin{array}{l} \displaystyle\bigcup_{b_1,\dots,b_k} \boxtimes_{1\le i\le k} \mathit{PreExitSets}(b_i), \text{ where} \\ \qquad k = |\tilde{\eta}(\eta^{-1}(e))|,\ \{b_1,\dots,b_k\} \subseteq \eta^+(\eta^{-1}(e)), \\ \qquad \forall i.\mathit{EXIT}(b_i) \ \wedge \ b_i \rightarrow e \in T \\ \qquad \eta^{-1}(\{b_1,\dots,b_k\}) = \tilde{\eta}(e) \end{array}\right\} & \text{if } \begin{array}{l}\mathit{EXIT}(e) \wedge \\ \mathit{AND}(\eta^{-1}(e))\end{array} \\ \left.\begin{array}{l} \displaystyle\bigcup_{m\in\eta(\eta^{-1}(e))} \mathit{PreExitSets}(m), \text{ where } m \xrightarrow{g,r} e \in T \\ \qquad\qquad \cup\ \{\{e\}\} \end{array}\right\} & \text{if } \begin{array}{l}\mathit{EXIT}(e) \wedge \\ \mathit{XOR}(\eta^{-1}(e))\end{array} \\ \{\,\{\}\,\} & \text{if } \mathit{BASIC}(e) \end{cases}$$

Here, the operator $\boxtimes : (2^{2^{\mathcal{S}}}) \times (2^{2^{\mathcal{S}}}) \to 2^{2^{\mathcal{S}}}$ 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 S:

$$\mathit{Leaves}(\rho, S) \quad := \quad \{b \in \rho^+(S) \mid \rho(b) = \varnothing\}$$

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

$$\begin{aligned}\mathit{JoinEnabled}(\rho, \mu, \nu, S) := \ & \mathit{BASIC}(S) \vee \\ & \exists E \in \mathit{PreExitSets}(S).\ \forall b \in \mathit{Leaves}(\rho, S). \\ & \qquad \exists b' \in E.\ b \xrightarrow{g} b' \ \wedge \ g(\mu, \nu)\end{aligned}$$

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

For the invariants of a location we use a function $\mathit{Inv}_\nu : \mathcal{S} \to \{\mathbf{true}, \mathbf{false}\}$, 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.

$$\mathit{Inv}(\rho, \nu) := \bigwedge_{b \in \rho^+(\mathit{root})} \mathit{Inv}_\nu(b)$$

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

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

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.

$$\begin{aligned}\mathit{UrgentEnabled}(\rho, \mu, \nu) := \ & \exists t : S \xrightarrow{g,r,u} S'.\ \mathit{TransitionEnabled}(t, \rho, \mu, \nu) \ \wedge \ u \\ & \vee \exists t_1 : S_1 \xrightarrow{g_1,s,r_1,u_1} S_1', t_2 : S_2 \xrightarrow{g_2,\bar{s},r_2,u_2} S_2'. \\ & \quad \mathit{TransitionEnabled}(t_1, \rho, \mu, \nu) \ \wedge \\ & \quad \mathit{TransitionEnabled}(t_2, \rho, \mu, \nu) \ \wedge \ (u_1 \vee u_2)\end{aligned}$$

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.

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

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 latter case, this includes the appropriate joins and/or fork operations.

The delay transition rule is:

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

where $\nu + d$ stands for the current clock assignment plus the delay $d \in \mathbb{R}_{\geq 0}$ 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.

$$\frac{\begin{array}{ll}\mathit{TransitionEnabled}(t_1 : S_1 \xrightarrow{g_1,c!,r_1,u_1} S_1',\ \rho, \mu, \nu) & S_1 \notin \eta^+(S_2) \\ \mathit{TransitionEnabled}(t_2 : S_2 \xrightarrow{g_2,c?,r_2,u_2} S_2',\ \rho, \mu, \nu) & S_2 \notin \eta^+(S_1)\end{array}}{(\rho, \mu, \nu, \theta) \xrightarrow{t_1,t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho, \mu, \nu, \theta)}\ \mathit{sync}$$

We choose the order first t_1, then t_2 here. This could be inverted, since the well-formedness constraints ensure that the assignments cannot conflict with each other. The side conditions $S_1 \notin \eta^+(S_2)$ and $S_2 \notin \eta^+(S_1)$ prevent synchronization of a superstate with its own descendants. For example, in Fig. 2.2 The a? transition exiting SUB cannot synchronize with the a! transition in P.

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 an invariant prevents time 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.

Similar to Def. 9, we define a set of timed traces for an HTA that capture its behavior. We explicitly exclude sequences that are zeno or not maximally extended.

Definition 2. (HTA Timed Trace Semantics)
Let $M = \langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Chan, Inv, T \rangle$ be an hierarchical timed automaton. A timed trace *of M is a sequence of configurations $\{(\rho, \mu, \nu, \theta)\}^K = (\rho, \mu, \nu, \theta)^0, (\rho, \mu, \nu, \theta)^1, \ldots$ of length $K \in \mathbb{N} \cup \{\infty\}$ if*

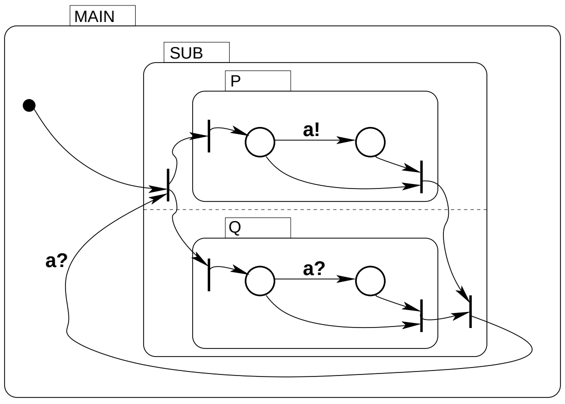


Fig. 2.2: The a? Transition Exiting SUB Cannot Synchronize with a! in P.

(i) *It starts at the initial configuration, i.e, for* $(\rho, \mu, \nu, \theta)^0$*:*
$\mathcal{S}_0 = (\rho^0)^*(root)$, $\mu = [Var \mapsto (0)^*]$, *and* $\nu = [Clocks \mapsto 0\,]$,

(ii) *Every step from* $(\rho, \mu, \nu, \theta)^k$ *to from* $(\rho, \mu, \nu, \theta)^{k+1}$ *is derived from the rules action, delay, and sync,*

(iii) *(maximally extended finite sequences)*
If $K < \infty$, *then for* $(\rho, \mu, \nu, \theta)^K$ *no further step is enabled, and*

(iv) *(non-zeno)*
If $K = \infty$ *and* $\{(\rho, \mu, \nu, \theta)\}^K$ *contains only a finitely many* k *such that* $(\rho^k, \mu^k) \neq (\rho^{k+1}, \mu^{k+1})$, *then eventually every clock value exceeds every bound* $(\forall x \in Clocks \forall c \in I\!N\, \exists k.\ \nu^k(x) > c)$.

The set of timed traces, denoted by $\mathcal{T}r(M)$, *is the* timed trace semantics *for* M.

3. The Timed Automata Model of Uppaal

Uppaal [LPY97] is a tool box for modeling, verification and simulation or real-time systems. It has been developed jointly by Uppsala University and Aalborg University throughout the last seven years. It is appropriate for systems that can be described as collection of non-deterministic parallel processes.

The modeling language used in Uppaal is an enriched dialect of the well studied timed automaton formalism [AD94], i.e., it features real-valued clocks over a finite control structure. Additionally the language allows for networks of timed automata that communicate through channels and/or shared variables. The usability and scalability of this formalism has been demonstrated by successfully application in various case studies, e.g., [LPY98, LP97, HSLL97].

In this Chapter we formally introduce the modeling language of Uppaal and equip it with a trace-based (formal) semantics. We use this semantics to specify the specification language of the tool, that allows for (timed) safety, reachability, inevitability, potentially always, and unbounded response.

3.1 Informal Description

An Uppaal model consists of a network of timed automata with clocks, invariants, variables over basic data types, guards, handshake synchronization, urgency, and committed locations.

The basic unit is one process, that consists of a directed control graph with labels on locations and transitions. One location is marked as initial, indicated by the notation ◎.

Data components. The data part of the model consists of discrete integer variables and (formal) clocks, that can take any non-negative real value. In Uppaal, integers are constrained to have values in the interval `[-32767; 32767]`. Exceeding the limits wraps around to this finite domain. Variables and clocks can be local to one process or global. If they are local, standard scoping rules apply and they cannot be accessed by other processes.

We note that for integer variables, Uppaal allows for some useful constructs. It is possible to declare integers with limited range, construct arrays of fixed width, and deal with integer expressions containing constants and the operators `+`, `-`, `*`, and `/`. For simplicity, we treat variables here always as integers and do not describe the full range of valid integer expressions. For the details we refer to [LPY97] and the online help.

Control structure.

Every location can be equipped with an *invariant*. This is constrained to be a conjunction of expressions $\texttt{x} \leq \texttt{const}$ and $\texttt{x} < \texttt{const}$, where `x` is a clock and `const` is an integer constant.

Locations can be equipped with one of the attributes *urgent* or *committed*. If a location is urgent, no time delay is possible before this location is left. A committed location also has to be left immediately, but leaving this location has precedence over other possible transitions. We use the graphical notations ⓤ and ⓒ for urgent or committed locations respectively.

Transitions are directed arcs between locations called the *source* and the *target*. Transitions can carry guards, assignments, and synchronization signals. We assume that guards and assignments are always given, in case of absence they are considered constant **true** or empty respectively.

Attributes for transitions.

For a location l, all transitions with source l are called *outgoing transitions* of l.

A *guard* is a conjunction of boolean expressions over variables and clock constraints of the form $\texttt{x}\sim\texttt{const}$ or $\texttt{x-y}\sim\texttt{const}$, where `x,y` are clocks, $\sim\in\{<,\leq,>,\geq\}$, and `const` is an integer constant.

Outgoing transitions without synchronization signals are *enabled*, if their guard evaluates to **true** and the invariant of the target location holds after

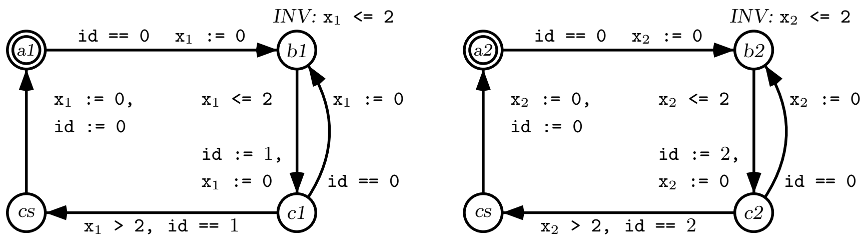


Fig. 3.3: Fischer's Protocol for Mutual Exclusion (2 Processes).

execution of the assignment.

An outgoing transition t_1 with synchronization signal `b!` is enabled, if there exists an outgoing transition t_2 in a parallel process with matching synchronization signal `b?`, and for both t_1 and t_2 the guards evaluate to **true** and the location invariants of the target locations hold after executing the corresponding assignments.

An *assignment* is a sequence of expressions that are either clock resets or of the form `v := expr`, where `v` is an integer variable or element of an array of integers, and `expr` is an arithmetic expression over integers.

Clock resets are of the form `x := 0`, where `x` is a clock.

Example 1. (Fischer's Mutex)
Fig. 3.3 shows of Fischer's mutual exclusion protocol for two Uppaal processes. The processes share the integer variable `id` (initially set to 0). Each process owns a clock $\mathtt{x}_i$, i.e., has exclusive read and reset operations on it. This clock is used to time the progress to the critical section (*cs*). The mutual exclusion property requires, that always at most one process in the critical section.

The processes, call them `P1` and `P2`, start at *a1* and *a2* with `id == 0` and clocks set to 0. Further progress in action and time delay is nondeterministic, as long as it obeys the restrictions of guards and invariants of the model. For example, an arbitrary amount of time can elapse (delay step) before any of the two processes takes a transition (action step). As a possible first action step, the first process can pass the guard `id == 0`, reset its clock $\mathtt{x}_i$ to 0, and move control to the location *a2*. The invariant *INV:* $\mathtt{x}_i$ `<= 2` requires, that *a2* is left again before clock $\mathtt{x}_i$ exceeds 2, i.e., within 2 time units. The only option to do so is taking the transition to *c1*, that writes the process number (1) to the shared variable `id` and resets the clock $\mathtt{x}_1$. Now in order to progress to the critical section *cs*, time has to elapse for *more* than 2 time units (guard $\mathtt{x}_i$ `> 2`). The guard `id == 1` makes sure, that no other process i has taken the transition b_i to c_i in the meantime. As it turns out, this suffices to establish mutual exclusion.

Behavior.

A *configuration* is a snapshot of the system with one designated control location for every process and values for all variables and clocks. An execution of the model starts in the implicit initial configuration, where every process is in its initial location, all clocks are 0 and all variables (global as local) are set to their initial value (integers are 0, arrays are filled with 0).

A configuration evolves in *action steps* and *delay steps*. Action steps are either isolated of synchronized. A simple action step amounts to taking one enabled transition of one process, execute assignments and clock resets and move control for this process to the new location. A synchronized action step means that two processes with enabled transitions, that carry matching synchronization signals (e.g, `b!` and `b?`) both take these transitions. Both associated assignments and clock resets are executed—the one corresponding to the `!`-transitions first—and control is updated for both processes.

If one of the processes is in a committed location, then all action steps not starting in committed location are blocked. In case of a synchronized action step, at least one of the two participating processes is required to be in a committed location, otherwise the step is blocked.

A delay step increases the value of all clocks by a real value $d > 0$. Delay is only enabled, if several conditions hold true.

(1) No process is in an urgent location,

(2) No process is in a committed location,

(3) No synchronized action on an urgent channel is enabled, and

(4) No location invariants are violated after the delay d.

We note that the real-valued nature of the delay steps is not directly observable, since clocks are always compared to integer values (in guards, invariants, and formulas). The possibility of real-valued delays basically allows for any order of the fractional part of clocks, which is not possible if the granularity of time is fixed in advance [Alu91].

A *trace* is a sequence of configurations, starting with the initial configuration. For every two consecutive configurations c_i and c_{i+1} in a trace, there has to exist an action or delay step that transforms c_i into c_{i+1}. For safety properties, it suffices it suffices to consider only finite traces, since every safety property can be violated (if at all) after a finite number of steps. For liveness, we have to consider both infinite and maximally extended finite (deadlocked) traces, since liveness properties can fail in the later case.

3.2 Formal Syntax

We define the formal syntax of Uppaal models as a parallel composition of processes.

For simplicity, we assume a set of labels *Labels*, that ranges over syntactically correct invariants, assignments, guards and synchronization labels. As a well-formedness condition, labels are constrained to occur only in appropriate places, contain only declared variables, and have to respect the variable types.

Definition 3. (Uppaal Process)
An Uppaal process A *is a tuple* $\langle L, T, \mathit{Type}, l^0 \rangle$*, where*

- L *is a set of locations,*
- T *is a set of transitions* $l \xrightarrow{g,s,a} l'$*, where* $l, l' \in L$*,* g *is a guard,* s *is a synchronization label (optional), and* a *is an assignment (possibly empty),*
- $\mathit{Type} : L \rightarrow \{\mathrm{o}, \mathrm{u}, \mathrm{c}\}$ *is a type function for locations, and*
- $l^0 \in L$ *is the initial location.*

We use the following access functions to refer to invariants, guards, synchronizations, and assignments.

- $\mathrm{Inv} : L \rightarrow \mathrm{Labels}$ *maps to the invariant of a location (possibly constant* **true***),*
- $\mathit{Guard} : T \rightarrow \mathrm{Labels}$ *maps to the guard of a transition (possibly constant* **true***),*
- $\mathit{Sync} : T \rightarrow \mathrm{Labels} \cup \{\varnothing\}$ *maps to the synchronization label of a transition (if any), and*
- $\mathit{Assign} : T \rightarrow \mathrm{Labels} \cup \{\varnothing\}$ *maps to the assignment associated with a transition (possibly the empty assignment).*

Definition 4. (Uppaal Model)
An Uppaal model *is a tuple* $\langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \rangle$*, where*

- $\vec{A}$ *is a vector of processes* $A_1, \ldots, A_n$*;*
  We use the index i *to refer to* A_i*-specific parts* L_i*,* T_i*,* Type_i*, and* l_i^0*,*
- Vars *is a set of variables, i.e., (bounded) integers and arrays,*
- Clocks *is a set of clocks,* $\mathit{Clocks} \cap \mathit{Vars} = \varnothing$*,*
- Chan *is a set of synchronization channels,* $\mathit{Chan} \cap \mathit{Vars} = \varnothing$*, and* $\mathit{Chan} \cap \mathit{Clocks} = \varnothing$*,*
- Type *is a polymorphic type function extending the* Type_i*, i.e.,* Type *maps*
  - *locations to* $\{\mathrm{o}, \mathrm{u}, \mathrm{c}\}$ *(according to the functions* Type_i*),*
  - *channels to* $\{\mathrm{o}, \mathrm{u}\}$*, and*
  - *variables to* $\{\mathrm{int}, \mathrm{array}\}$*.*

  We use o*,* u*,* c*,* int*, and* array *as predicates, i.e., for a channel* b *the expression* $\mathrm{u}(b)$ *evaluates to* **true***, if and only if* $\mathit{Type}(b) = \mathrm{u}$*.*

Definition 5. (Configuration)
A configuration *of an* Uppaal *model* $\langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \rangle$ *is a triple* $(\vec{l}, \mathrm{e}, \nu)$*, where* $\vec{l}$ *is a vector of locations,* e *is the environment for discrete variables, and* ν *is the clock evaluation, i.e.:*

- $\vec{l} = (l_1, \ldots, l_n)$*, where* $l_i \in L_i$ *is a location of process* A_i*,*
- $\mathrm{e} : \mathit{Vars} \rightarrow (\mathbb{Z})^*$ *maps every variable* v *to either a value (if* $\mathrm{int}(v)$*) or a tuple of values (in case of* $\mathrm{array}(v)$*), and*
- $\nu : \mathit{Clocks} \rightarrow \mathbb{R}_{\geq 0}$ *maps every clock to a non-negative real number. For* $d > 0$*, the notation* $(\nu + d) : \mathit{Clocks} \rightarrow \mathbb{R}_{\geq 0}$ *describes the function "*ν

shifted by d" in the following sense:
$\forall x \in Clocks.\ (\nu(x) + d) \; = \; \nu(x) + d.$

Sometimes it is necessary to refer to certain parts of a configuration. We call $\vec{l}$ the *control situation* the pair $(\vec{l}, \mathrm{e})$ the *discrete part*, and ν the *continuous part* of a configuration.

3.3 Trace Semantics of the UPPAAL *Model*

UPPAAL models evolve according to legal steps, that are either delays or actions. The compendium of all legal steps defines the behavior of the model.

We start by formulating simple actions, synchronized action, and delay steps. To modify the control situation $\vec{l}$, we use the notation $\vec{l}[l_i'/l_i]$ to indicate, that at position i, l_i was replaced by l_i', and the other positions did not change. We readily use assignments a as transformers on the function e (and ν) and write $a(e)$ (and $a(\nu)$) for the resulting evaluations. Furthermore we use the notation $\mathrm{e}, \nu \models_{loc} \varphi$ to indicate, that a boolean expression φ holds true under the evaluations e, ν for the contained variables and clocks, and $(\vec{l}, \mathrm{e}, \nu) \models_{loc} \varphi$ analogously in the case that φ contains expressions of the form $A_i.l_i$ (denoting that process A_i is in location l_i). We defer a formal definition of $\models_{loc}$ to Section 4.1.

DEFINITION 6. (SIMPLE ACTION STEP) *For a configuration* $(\vec{l}, \mathrm{e}, \nu)$*, a* simple action step *is enabled, if there is a transition* $l_i \xrightarrow{g,a} l_i' \in T_i$*,* l_i *in* $\vec{l}$*, such that*

(1) $\mathrm{e}, \nu \models_{loc} g$*,*

(2) $a(\mathrm{e}), a(\nu) \models_{loc} \mathrm{Inv}(l_i')$*, and*

(3) if $\exists l_c$ *in* $\vec{l}$ *with* $\mathrm{c}(l_c)$*, then* $\mathrm{c}(l_i)$*.*

We abbreviate this with $(\vec{l}, \mathrm{e}, \nu) \stackrel{a}{\Longrightarrow} (\vec{l}[l_i'/l_i], a(\mathrm{e}), a(\nu))$

DEFINITION 7. (SYNCHRONIZED ACTION STEP) *For a configuration* $(\vec{l}, \mathrm{e}, \nu)$*, a* synchronized action step *is enabled if and only if for a channel b there exist two transitions* $l_i \xrightarrow{g_i, b!, a_i} l_i' \in T$ *and* $l_j \xrightarrow{g_j, b?, a_j} l_j' \in T$*,* l_i, l_j *in* $\vec{l}$*,* $i \neq j$*, such that*

(1) $\mathrm{e}, \nu \models_{loc} g_i \wedge g_j$*,*

(2) $a_j(a_i(\mathrm{e})), a_j(a_i(\nu)) \models_{loc} \mathrm{Inv}(l_i') \wedge \mathrm{Inv}(l_j')$*, and*

(3) if $\exists l_c$ *in* $\vec{l}$ *with* $\mathrm{c}(l_c)$*, then* $\mathrm{c}(l_i) \vee \mathrm{c}(l_j)$*.*

We abbreviate this with $(\vec{l}, \mathrm{e}, \nu) \stackrel{\tau}{\Longrightarrow} (\vec{l}[l_i'/l_i][l_j'/l_j], a_j(a_i(\mathrm{e})), a_j(a_i(\nu))$

DEFINITION 8. (DELAY STEP) *For a configuration* $(\vec{l}, \mathrm{e}, \nu)$*, a* delay step *with delay d is enabled, if and only if all of the following holds.*

(1) $\forall l_i$ *in* $\vec{l}.\ \neg \mathrm{u}(l_i)$,

(2) $\forall l_i$ *in* $\vec{l}.\ \neg c(l_i)$,

(3) $\neg\exists l_i \xrightarrow{g_i, b!, a_i} l'_i \in T_i,\ l_j \xrightarrow{g_j, b?, a_j} l'_j \in T_j$, *with* l_i, l_j *in* $\vec{l}$, $i \neq j$, *such that* $\mathrm{u}(b)$, $\mathrm{e}, \nu \models_{loc} g_i$, $\mathrm{e}, \nu \models_{loc} g_j$, $a_j(a_i(\mathrm{e})) \models_{loc} \mathrm{Inv}(l'_i) \wedge \mathrm{Inv}(l'_j)$, *and*

(4) $\mathrm{e}, (\nu + d) \models_{loc} \bigwedge_i \mathrm{Inv}(l_i)$.

We denote this by $(\vec{l}, \mathrm{e}, \nu) \overset{d}{\Longrightarrow} (\vec{l}, \mathrm{e}, (\nu + d))$.

Definition 9. (Well-Formed Sequence/Timed Trace)
Let $M = \langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \rangle$ *be a* Uppaal *model. A sequence of configurations* $\{(\vec{l}, \mathrm{e}, \nu)\}^K = (\vec{l}, \mathrm{e}, \nu)^0, (\vec{l}, \mathrm{e}, \nu)^1, \ldots$ *of length* $K \in \mathbb{N} \cup \{\infty\}$ *is called a* well-formed sequence for M, *if*

(i) $(\vec{l}, \mathrm{e}, \nu)^0 = \big((l_1^0, \ldots, l_n^0),\ [\mathit{Vars} \mapsto (0)^*],\ [\mathit{Clocks} \mapsto 0]\big)$,

(ii) (maximally extended finite sequences)
If $K < \infty$, *then for* $(\vec{l}, \mathrm{e}, \nu)^K$ *no further step is enabled,*

(iii) (non-zeno)
If $K = \infty$ *and* $\{(\vec{l}, \mathrm{e}, \nu)\}^K$ *contains only finitely many* k *such that* $(\vec{l}^{\,k}, \mathrm{e}^k) \neq (\vec{l}^{\,k+1}, \mathrm{e}^{k+1})$, *then eventually every clock value exceeds every bound* ($\forall x \in \mathit{Clocks}\, \forall c \in \mathbb{N}\, \exists k.\ \nu^k(x) > c$).

A well-formed sequence for M *is called a* timed trace for M, *if in addition the following holds.*

(iv) For every $k < K$, *the two subsequent configurations* k *and* $k+1$ *are connected via a simple action step, a synchronized action step, or a delay step, i.e.,*

$$\begin{array}{ll} (\vec{l}, \mathrm{e}, \nu)^k \overset{a}{\Longrightarrow} (\vec{l}, \mathrm{e}, \nu)^{k+1} & \textit{or} \\ (\vec{l}, \mathrm{e}, \nu)^k \overset{\tau}{\Longrightarrow} (\vec{l}, \mathrm{e}, \nu)^{k+1} & \textit{or} \\ (\vec{l}, \mathrm{e}, \nu)^k \overset{d}{\Longrightarrow} (\vec{l}, \mathrm{e}, \nu)^{k+1}. & \end{array}$$

Condition *(iii)* weeds out those traces, where time converges towards a finite value in an infinite number of steps. These traces are also called *zeno traces* and correspond to a degenerated behavior of the model, i.e., they have no counterpart in the physical world where time always progresses.

We note that according to this definition, an infinite trace may yield an infinite loop of (synchronized) action steps. This also prevents time from progressing, but is rather a failure of the model than a flaw of the modeling language. These degenerated traces are kept in semantics to make it possible to detect failures of this type.

Example 2. (Zeno Traces) Consider a Uppaal model consisting of one Uppaal process A and one clock x. A has only one (initial) location l with the invariant $x \leq 2$. Now one can construct a sequence of delay steps with

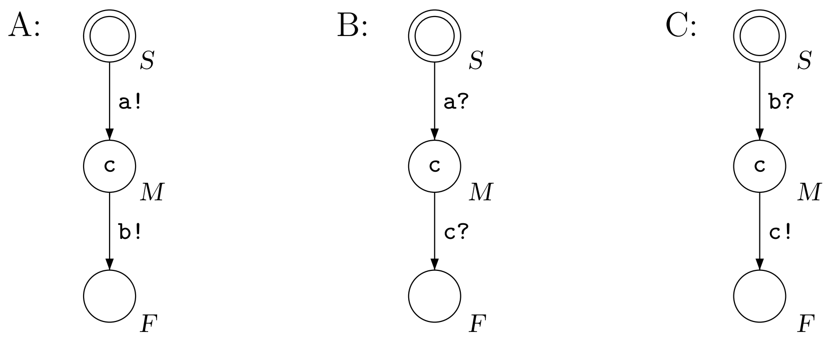


Fig. 3.4: The Control Situation `A.F and B.F and C.F` Can be Reached Via the Trace `(A.S B.S C.S)` $\overset{\tau}{\Longrightarrow}$ `(A.M B.M C.S)` $\overset{\tau}{\Longrightarrow}$ `(A.F B.M C.M)` $\overset{\tau}{\Longrightarrow}$ `(A.F B.F C.F)`.

delay values 1, 1/2, 1/4, 1/8, ect. This sequence can be infinite without ever reaching a configuration with $\nu(x) = 2$.

According to Def. 9 (iii), this sequence is not a valid trace. For this UPPAAL model every trace is finite and ends, due to (ii), in the configuration where A is at l and $\nu(x) = 2$. There are uncountably many such traces.

We now associate an UPPAAL model M with an (typically uncountable) set $\mathcal{T}(M)$ of timed traces that are either infinite or maximally extended (deadlocked).

DEFINITION 10. (TRACE SEMANTICS) *Let M be an* UPPAAL *model. Then the* trace semantics *of M, written $\mathcal{T}(M)$, is the set of timed traces according to Def. 9.*

Note that timed traces are memoryless in the sense that the possible futures do only depend on a configuration and not on the history. If two traces $\sigma_1, \sigma_2 \in \mathcal{T}(M)$ contain the same configuration $\mathbf{s}$, the prefixes leading to $\mathbf{s}$ can be interchanged and the resulting sequences are both again timed traces in $\mathcal{T}(M)$. This property is sometimes called *fusion closure*.

We note that the UPPAAL timed automata model has been equipped with semantics before, in particular in [Pet99]. However, the latter does not correspond to the implementation of committed locations as implemented in UPPAAL 3.0.x, 3.2.x, and later. In Fig. 3.4 the control situation `A.F and B.F and C.F` can not be reached according to [Pet99] p. 140 (second bullet point). In the implementation it can be reached, and our semantics reflects this.

4. The Logic Language of UPPAAL

The UPPAAL model checking engine allows to automatically establish or refute properties that are expressed in a specification language. This language is a subset of *timed computation tree logic* (TCTL, [ACD93]), where primitive expressions are location names, variables, and clocks from the modeled system.

We define validity of formulas in the specification language relative to the semantics given in the previous section.

4.1 Local Properties

A local property is a condition, that for a specific configuration is either **true** or **false**. The basic building blocks are expressions over locations, variables, and clocks. It is crucial for the efficiency of property verifications that clocks can only be compared to integer values.

Definition 11. (Local Property)
Given an Uppaal *model* $\langle\vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type}\rangle$. *A formula* φ *is a* local property *iff it is formed according to the following syntactic rules.*

$\varphi ::=$	`deadlock`	
$\mid$	$A.l$	*for* $A \in \vec{A}$ *and* $l \in L_A$
$\mid$	$x \bowtie \mathtt{c}$	*for* $x \in \mathit{Clocks}$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$, $\mathtt{c} \in \mathbb{Z}$
$\mid$	$x - y \bowtie \mathtt{c}$	*for* $x, y \in \mathit{Clocks}$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$, *and* $\mathtt{c} \in \mathbb{Z}$
$\mid$	$a \bowtie b$	*for* $a, b \in \mathit{Vars} \cup \mathbb{Z}$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{!=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$
$\mid$	`(`φ_1`)`	*for* φ_1 *a local property*
$\mid$	`not` φ_1	*for* φ_1 *a local property*
$\mid$	φ_1 `or` φ_2	*for* φ_1, φ_2 *local properties (logical OR)*
$\mid$	φ_1 `and` φ_2	*for* φ_1, φ_2 *local properties (logical AND)*
$\mid$	φ_1 `imply` φ_2	*for* φ_1, φ_2 *local properties (logical implication)*

The truth value of a local property can effectively be evaluated in a configuration $\mathtt{s}$.

Definition 12. (Validity of a Local Property) *A local property* φ *is* valid *in a configuration* $\mathtt{s} = (\vec{l}, e, \nu)$, *in symbols* $\mathtt{s} \models_{loc} \varphi$, *iff it is valid according to the following structural definitions.*

$\mathtt{s} \models_{loc}$ `deadlock`	*iff*	*no delay or action steps are enabled in* $\mathtt{s}$
$\mathtt{s} \models_{loc} A.l$	*iff*	$l = l_i$ *in* $\vec{l}$ *for* $A = A_i$ *in* $\vec{A}$
$\mathtt{s} \models_{loc} x \bowtie \mathtt{c}$	*iff*	$\nu(x) \bowtie \mathtt{c}$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$
$\mathtt{s} \models_{loc} x - y \bowtie \mathtt{c}$	*iff*	$\nu(x) - \nu(y) \bowtie \mathtt{c}$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$
$\mathtt{s} \models_{loc} a \bowtie b$	*iff*	$e(a) \bowtie e(b)$, $\bowtie \in \{\mathtt{<}, \mathtt{<=}, \mathtt{!=}, \mathtt{==}, \mathtt{>=}, \mathtt{>}\}$
$\mathtt{s} \models_{loc}$ `(`φ_1`)`	*iff*	$\mathtt{s} \models_{loc} \varphi_1$
$\mathtt{s} \models_{loc}$ `not` φ_1	*iff*	$\neg\,(\mathtt{s} \models_{loc} \varphi_1)$
$\mathtt{s} \models_{loc} \varphi_1$ `or` φ_2	*iff*	$\mathtt{s} \models_{loc} \varphi_1$ *or* $\mathtt{s} \models_{loc} \varphi_2$
$\mathtt{s} \models_{loc} \varphi_1$ `and` φ_2	*iff*	$\mathtt{s} \models_{loc} \varphi_1$ *and* $\mathtt{s} \models_{loc} \varphi_2$
$\mathtt{s} \models_{loc} \varphi_1$ `imply` φ_2	*iff*	$\neg(\mathtt{s} \models_{loc} \varphi_1)$ *or* $\mathtt{s} \models_{loc} \varphi_2$

Above, φ_1 *and* φ_2 *stand for local properties.*

`E<>` φ	reachability of φ
`A[]` φ	safety (invariantly φ)
`E[]` φ	possibly always φ
`A<>` φ	inevitably φ
φ `-->` ψ	unbounded response
	(corresponds to `A[]` ($\varphi \Rightarrow$ `A<>` ψ))

φ, ψ: local properties

Fig. 4.5: The Classes of TCTL Formulas, that UPPAAL can Model Check.

This notion of locality must not be confused with locality in the sense of "local to one process." The UPPAAL language allows also to declare variables and clocks locally to one process P and uses the syntax `P.var` to identify the `var` that is local to P. Note that every locally declared variable or clock can be equivalently replaced by a global one under appropriate renaming of labels. For simplicity we therefore treat all variables and clocks as global.

4.2 Temporal Properties

The five classes of temporal properties that UPPAAL can effectively verify are summarized in Fig. 4.5. We define the validity of temporal properties via our trace semantics (Def. 10). We chose to give the direct definition of three of the classes and define the remaining two classes as syntactic duals.

DEFINITION 13. (TEMPORAL PROPERTIES)
Let $M = \langle \vec{A}, Vars, Clocks, Chan, Type \rangle$ be an UPPAAL *model and let φ and ψ be local properties. The* validity of temporal properties *is defined for the classes* `A[]`, `A<>`, *and* `-->` *as follows.*

$$M \models \texttt{A[]}\ \varphi \quad \textit{iff} \quad \forall \{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \ \forall k \leq K.\ \ (\vec{l}, e, \nu)^k \models_{loc} \varphi$$

$$M \models \texttt{A<>}\ \varphi \quad \textit{iff} \quad \forall \{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \ \exists k \leq K.\ \ (\vec{l}, e, \nu)^k \models_{loc} \varphi$$

$$M \models \varphi\ \texttt{-->}\ \psi \quad \textit{iff} \quad \forall \{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \ \forall k \leq K.$$
$$(\vec{l}, e, \nu)^k \models_{loc} \varphi \ \Rightarrow\ \exists k' \geq k.\ (\vec{l}, e, \nu)^{k'} \models_{loc} \psi$$

The two temporal property classes dual to `A[]` *and* `A<>` *are defined below.*

$$M \models \texttt{E<>}\ \varphi \quad \textit{iff} \quad \neg(M \models \texttt{A[] not}(\varphi))$$

$$M \models \texttt{E[]}\ \varphi \quad \textit{iff} \quad \neg(M \models \texttt{A<> not}(\varphi))$$

EXAMPLE 3. (FISCHER'S MUTEX, CONTINUED)
The mutual exclusion property of the UPPAAL model in Example 1 can be expressed by the local property `not ( P1.cs and P2.cs )`. This is a local property that has to hold invariantly, i.e., it should be true that $\mathit{Fischer}_2 \models$ `A[] not ( P1.cs and P2.cs )`.

Other temporal properties that should hold include, e.g., that every process *can* reach the critical section: `E<> P1.cs` and `E<> P2.cs` .

Uppaal is not a modeling tool for design. The timed automata model is much more restricted than a formalism that a system developer would use. One of the important missing features is hierarchical structure.

Most interesting properties in a real-world design language can be expected to be undecidable. Automated analysis then requires an abstraction step. To establish soundness of this step, it has to be clear *what* gets abstracted. In compiler optimization, for example, safe over-approximation by replacing data domains by Boolean values has been very successful (e.g., [NNH99]). Here data is abstracted, but control structure is preserved.

There is a gap between a design tool and a formalism for automated analysis. The former tends to have rich data types, powerful synchronization mechanisms, and hierarchical organization. The latter has the strong obligation to remain in a decidable fragment.

5. Flattening Hierarchical Timed Automata

We now address the algorithmic verification of the hierarchical timed automata (HTA) model from Section 2. Our claim is that presence of the hierarchies does merely complicate the verification part, but not hinder it. In particular we consider the specification language of Uppaal suitable for specifying properties.

The foundation for establishing properties of HTAs is the trace-based formal semantics. We do not have a model checking engine for HTAs. Instead we *flatten* a HTA model to a Uppaal model and make use of the well-engineered implementation of that tool. This translation is complicated mainly by the implicit synchronization on exit. We give first a high-level description and subsequently elaborate to the relevant details.

5.1 Overview on the Flattening Procedure

Flattening of statechart-like languages is complicated mainly by the presence of transitions that result in a cascade of entries and exits. In particular the synchronization on exit gives rise to complex auxiliary constructs.

In this Section we give an overview description of our flattening procedure. It is subsequently elaborated in Section 5.2.

Flattening a hierarchical timed automaton.On the topmost level of an HTA we find a parallel composition of superstates, conceptually under an implicit root. Each can be of type *AND* or *XOR* and can itself contain superstates. The complete collection of superstates is called the *instantiation tree*. In Section 2.1 this corresponds to η. At any point in time the behavior of a HTA depends on the sub-tree of this instantiation tree that is currently active.

Every superstate S in the instantiation tree is translated to one Uppaal process $\widehat{S}$. All those processes are put in parallel. An auxiliary location in $\widehat{S}$ is added for the configurations where S is not active (i.e., is idle). The translation proceeds in three main phases.

I. *Collection of instantiations:* The instantiation tree is traversed and for every superstate S the skeleton of a (flat) process $\widehat{S}$ is constructed. This contains basic locations, transitions, and the auxiliary initial location $\widehat{S}$_IDLE. Entries to S are translated to guarded transitions from $\widehat{S}$_IDLE.

II. *Computation of global joins:* Transitions originating from superstates can require a cascade of substate exits, called *global join*. All configurations that can synchronize to such a global join are computed. This yields a guard condition that evaluates to **true** if an only if one such cascade can be taken to completion.

III. *Post-processing channel communication:* If a transition in the HTA starts at a superstate S and carries a synchronization, it cannot synchronize with a transition *inside* S. Since the sub-state/superstate relation is lost in the translation, we resolve this conflict explicitly by duplicating channels and transitions.

Correspondence of hierarchical and flattened model. A configuration in the HTA model M corresponds to one configuration in the flattened version $\widehat{M}$. All other configurations of $\widehat{M}$ are either intermediate to this or unreachable. This correspondence allows us to associate every trace of M with one in $\widehat{M}$.

This association dictates the property language for hierarchical timed automata. We sketch this only conceptually. Of main interest are the classes of properties that can be model checked with UPPAAL, see Section 4. Consequently, the syntax of properties for hierarchical timed automata is like in Fig. 4.5. The difference is that the local properties are required to identify (super)locations, variables, and clocks uniquely. It is necessary to trace back every identifier to the point in the instantiation tree where it is declared. Note that scoping rules allow to override a declarations of x in an ancestor superstate in the instantiation tree. Thus the identifier x can be associated with a different variable, and even a different type, depending on where it occurs.

These scoping problems can be solved via *renaming*. All ambiguities introduced by name duplications can be consistently resolved by prefixing a path of instantiation names to identifiers, starting at the implicit *root*. For simplicity we omit this renaming in our description and treat all variables, clocks, and channels as global. This way for every property φ in the HTA we can compute a corresponding property $\widehat{\varphi}$ for the flattened model, where the identifiers and names of superstates are replaced accordingly.

The subsequent Section 5.2 contains a more detailed description of the flattening procedure. In Section 7 we use a cardiac pacemaker as a case study.

5.2 Flattening in More Detail

We now give a detailed description our flattening procedure. This is organized in three phases: Translation of superstates and their entries, transla-

```
Algorithm: PHASE I: instantiateTemplates
  input:   Stack $\mathfrak{S}$ of superstates to translate
  output:  Set $\mathbf{P}$ of (flat) timed automata
           Set $\mathbf{G}$ of global join starting points
  $\mathbf{P} := \{$Global_Kickoff automaton for $s \in \mathfrak{S}\}$
  $\mathbf{G} := \varnothing$
  WHILE $notempty(\mathfrak{S})$
      $S := pop(\mathfrak{S})$
      $\mathcal{C} := \{$non-basic locations $B$ in $S\}$
      FORALL $B \in \mathcal{C}$
          $push([B \text{ in } S], \mathfrak{S})$
          /⋆ $[B$ in $S]$ inherits all invariants attached to $S$ ⋆/
          create a location $\widehat{B}$ in $\widehat{S}$
          $E_B := \{$set of entries of $B$ in $S\}$
          FORALL $e \in E_B$
              create a committed location $\widehat{B_e}$ in $\widehat{S}$
              create a transition from $\widehat{B_e}$ to $\widehat{B}$ in $\widehat{S}$
              /⋆ this transition carries a synchronization enter_$B$_in_$S$_via_$e$! ⋆/
          IF $type(S)$ = $XOR$ THEN
              $\mathbf{G} := \mathbf{G} \cup \{B$ in $S\}$

      $\mathbf{P} := \mathbf{P} \cup \{$translation $\widehat{S}$ of superstate $S$, depending on $type(S)\}$
```

Fig. 5.6: Algorithm for Translation of the Instantiation Tree.

tion of exits, and post-processing of channels.

In their syntactic representation via XML files, both the hierarchical timed automata model and then UPPAAL model rely on a *template mechanism*. Templates for superstates (processes) are instantiated to create the concrete superstates (processes) that constitute the actual model. This works very much like instantiation of classes to objects, and the motivation is also similar. It should be easy to make small consistent modifications, e.g., via setting parameters. Parts that are (nearly) identical should not be described twice but derived as two instantiations of the same template. The implementation of our flattening procedure therefore in fact translates a set of HTA templates plus an instantiation at root level to a set of flat timed automata templates where each is instantiated exactly once.

Conceptually, however, the translation works on instantiation level. If a superstate template is instantiated twice, the two instantiations are translated separately. This makes it easier to take the context into account. At template level, e.g., no parent superstate can be attributed to a template. To construct translations of entries or exits, knowledge about this context is crucial. For simplicity we therefore describe the translation as if all superstates and processes were primitives.

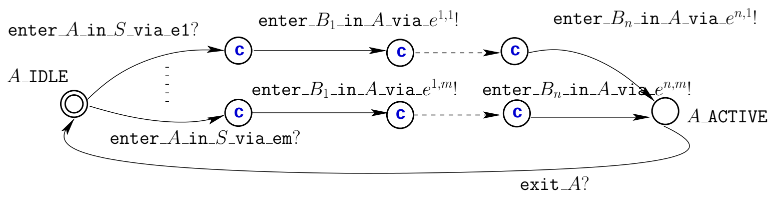

enter_A_in_S_via_e1?
enter_B_1_in_A_via_$e^{1,1}$!
enter_B_n_in_A_via_$e^{n,1}$!
A_IDLE
c
c
c
c
c
c
enter_B_1_in_A_via_$e^{1,m}$!
enter_B_n_in_A_via_$e^{n,m}$!
A_ACTIVE
enter_A_in_S_via_em?
exit_A?

Fig. 5.7: Translation of Entries and Exits an *AND* Superstate.

5.2.1 Translation of Superstates and Entries — Phase I

We sketch now the translation of a superstate S to a process $\widehat{S}$, the pseudocode is given in Fig. 5.6.

For every location l in S, $\widehat{l}$ is created in $\widehat{S}$. Additional $\widehat{S}$ contains the location S_IDLE, which is the initial location. Every entry of S corresponds to a transition in $\widehat{S}$ originating from S_IDLE. Some auxiliary constructions are necessary to mimic the behavior of hierarchical machines adequately. They depend on the type (*XOR* or *AND*) of S.

Translation of *XOR* superstates.

In a hierarchical *XOR* superstate X, at most one location is active at a given time. For every substate B of X we introduce a location B_ACTIVE_IN_X in $\widehat{X}$. Moreover, for every entry e of B we introduce an auxiliary location in $\widehat{X}$, called X_AUX_B_e. These are declared committed and are connected to B_ACTIVE_IN_X with a transition that synchronizes on a channel enter_B_in_X_via_e. Transitions leading originally to a B-entry e in X are represented in the translation by leading to X_AUX_B_e and trigger—without interleaving with other processes—the activation of the substate B.

Exits of substates B are translated similarly by transitions from B_ACTIVE_IN_X. They give rise to additional complications since leaving an *AND* substate B is only possible if all descendants of B can exit. So in fact a chain of exit transitions starting at B_ACTIVE_IN_X can be necessary, see Section 5.2.2.

If the *XOR* superstate X is inactivated (exited), this corresponds in the translation $\widehat{X}$ to transitions to X_IDLE. This transition carries the synchronization exit_X?. If the superstate X has a default exit, every non-auxiliary location in $\widehat{X}$ has such a transition to B_IDLE.

Translation of *AND* superstates. An *AND* superstate A is a parallel composition of superstates. Either non of them is active or all of them are. In the translation $\widehat{A}$ (Fig. 5.7), this corresponds to locations A_IDLE and A_ACTIVE. If A is activated, this is specific to an entry e_i of A. The substates B_j of A are entered one after another. Which entry is used for each B_j is dependent on e_i. Thus for every entry e_i of A there is a separate chain of transitions leading from A_IDLE to A_ACTIVE. The choice of entries of B_j is reflected by appropriate signals enter_B_j_in_A_via_$e^{j,i}$. The auxiliary locations in the chain are declared committed, so no time can elapse before A_ACTIVE is reached and interleaving with other processes is blocked.

Kickoff. Since the root of the instantiation tree is implicit, one special process is needed to trigger the entry of the topmost superstates. This process is called `Global_Kickoff` and also initializes all variables.

We note that the topmost superstates S_i are considered special, since they do not synchronize on exit. Instead, they can be enabled to become in-active via following a special exit transition. Once one of these S_i 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 is intended to be able to be both inactivated and activated again, it cannot nest at the root level but must be itself contained in a superstate.

History. History amounts to record the status of an *XOR* superstate X when it is exited. Since we assume all variables and clocks to be global, this amounts to storing the last control location. This can be encoded via an auxiliary integer variable *hist* that is updated along each transition in $\widehat{X}$. Each value corresponds exactly to one location $\widehat{l_i}$ in $\widehat{X}$. The history entry then has a transition to each location $\widehat{l_i}$ guarded by the expression $hist{==}\ i$. If *hist* has its initial value 0, then then the only guard evaluating to **true** leads to the default history location.

The clocks local to superstates with history entry are not frozen on exit but kept running. Reachability for automata with stopwatches is undecidable [CL00]. If local clocks are declared to be forgetful, then they are reset along every entry. Otherwise they resume with the accumulated value.

For simplicity we do not treat history in our flattening procedure.

Urgent transitions. In the HTA formalism transitions can be declared urgent. The corresponding concept in the UPPAAL model is to declare channels urgent, i.e., channels where synchronization has preference over time delay. An urgent transition t can be encoded by this as follows.

a) If t does not carry synchronization:
   Add a dummy synchronization `Hurry?` on the transition and add one parallel process `HurryDummy` that constantly offers synchronization on this channel.

b) If t synchronizes on channel c:
   Declare c urgent. If there are situations where two non-urgent transitions can synchronize on c, then it is necessary to introduce a urgent and non-urgent copy of c and duplicate all transitions where both urgent and non-urgent synchronizations are possible.

For simplicity we do not treat urgency in our flattening procedure.

5.2.2 Exit of Superstates via Global Joins — Phase II

The exit of a superstate S is represented in $\widehat{S}$ by a transition to S`_IDLE` which carries the synchronization signal `exit_`S?. These exits do not necessarily happen in isolation, but might happen as part of a cascade of exits from superstates and non-basic substates. Thus it is necessary

```
Algorithm: PHASE II: expandGlobalJoins
  input:    Set $\mathbf{G}$ of global join starting points
  output:   auxiliary constructions: counters and guarded transitions

  $JoinTrees := \varnothing$
  FORALL $\mathbf{g} \in \mathbf{G}$
      collect all trees $\mathbf{T}$ of control locations that can synchronize to $\mathbf{g}$;
      the leaves of $\mathbf{T}$ are sets of basic locations that share transitions to the
      same exit $e$.
      /* These sets are singletons, if $e$ is an ordinary exit
         and span over all basic locations in the superstate otherwise          */
      $JoinTrees := JoinTrees \cup \{\mathbf{T}\}$
  FORALL $\mathbf{T} \in JoinTrees$
      let $\widehat{L} := \{\widehat{l} \mid l$ is element in a basic location set of $\mathbf{T}\}$
      declare the counter $trigger_{\mathbf{T}}$
      FORALL $\widehat{l} \in \widehat{L}$
          FORALL transitions $\widehat{k} \to \widehat{l}$
              add the assignment $trigger_{\mathbf{T}} := trigger_{\mathbf{T}} + 1$   to   $\widehat{k} \to \widehat{l}$
          FORALL transitions $\widehat{l} \to \widehat{m}$
              add the assignment $trigger_{\mathbf{T}} := trigger_{\mathbf{T}} - 1$   to   $\widehat{l} \to \widehat{m}$
      let $N :=$ number of leaves of $\mathbf{T}$
      let $\mathcal{S}_{\mathbf{T}} :=$ superstates $S$ occurring in $\mathbf{T}$
      FORALL transitions $t$ starting at $root(\mathbf{T})$
          create a chain of transitions, starting with $\widehat{t}$,
                corresponding to exiting every $S \in \mathcal{S}_{\mathbf{T}}$
```

Fig. 5.8: Pseudo-code for the Encoding of All Global Joins.

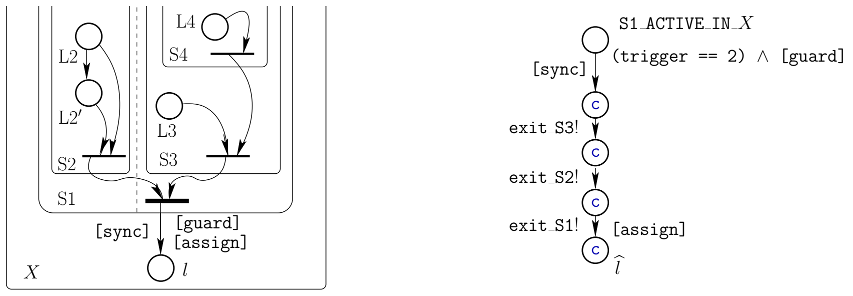
(1) to derive conditions that allow a set of superstates to exit, and
(2) to make sure that always the complete set of exits is performed.

We call the process of performing a legal set of exits a *global join*.

EXAMPLE 4. (GLOBAL JOIN)
Consider Fig. 5.9 (i) with control at (L2,L3). Then the superstates S3, S2, and S1 have to be left, in order to reach l. The same holds for control situation (L2′, L3). This cascade of exits is encoded the sequence of in Fig. 5.9 (ii). However, if control is at (L2,L4), then S4 must be left as well, this would correspond to a *different* sequence of substate exits than displayed in (ii), i.e., a different global join.

One transition leaving a superstate B can give rise to a number of global joins, possibly exponential in the depth of hierarchical structure.

For every global join there is exactly one proper transition that does not lead to an exit. In Example 4 this is the transition to l. An auxiliary variable `trigger` keeps track of the number of active basic locations, that can participate in this join. In a transition from L2 to L2′, for example, the value of `trigger` does not change. `trigger` has to reach the threshold



(i) Part of an XOR Superstate X　　*(ii) Exits in $\widehat{X}$, starting at (*L2,L3*)*

Fig. 5.9: Translation of a Global Join That is Rooted at *XOR* Superstate *X*.

```
Algorithm: PHASE III: postprocessChannels
  input:   Queue $Q$ over ($syncSignal$, $transition$, $S$)
  WHILE notempty($Q$)
      ($syncSignal$, $transition$, $S$) :=pop($Q$)
      IF ∃ transition $t$ with $match(syncSignal)$ in $S$:
          create a new channel $c$
          replace $channel(syncSignal)$ on $transition$ by $c$
          FORALL transitions $t$ with $match(syncSignal)$ outside $S$
              create a copy $t'$ of $t$, where $channel(syncSignal)$ is replaced by $c$
              if $\exists(s',t,S') \in Q$ then $push\big((s',t',S'),\ Q\big)$
```

Fig. 5.10: Pseudo-code for Post-processing Synchronization Channels.

value—here: 2—to enable the global join. It is crucial that the proper transition terminating the global join—here: `S1` to l—can be taken, i.e., that the guard (if any) evaluates to **true**. Likewise the synchronization with other transitions (if any) has to be possible at this point in time.

Thus, in the sequence of substate exits in Fig. 5.9 (ii), `[guard]` and `[sync]` are attached to the *first* transition, while `[assign]` is executed along the last transition.

5.2.3 Post-Processing of Channels — Phase III

Transitions that cause the same location to be exited are in conflict, i.e., they cannot be executed simultaneously. The only case where two transitions in the HTA model are taken truly simultaneous (and not interleaved) is the synchronization along channels. E.g., in Fig. 2.2, the a? transition exiting SUB cannot synchronize with the a! transition in P.

In the flattening the structural relation of ancestor/descendant is lost. Therefor we have to prevent synchronization between the processes $\widehat{\mathsf{SUB}}$

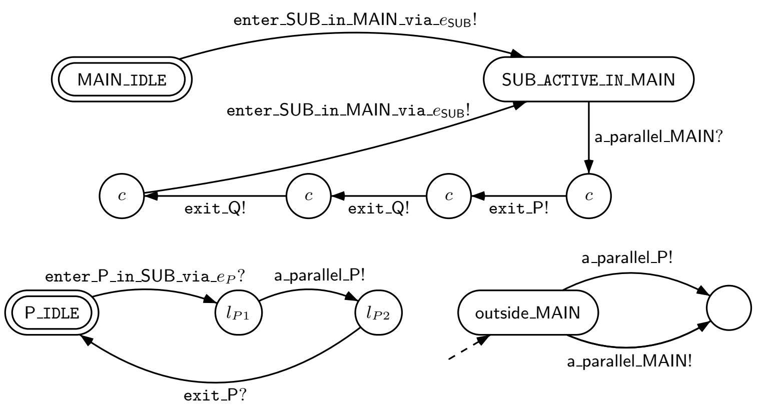


Fig. 5.11: Part of the Flattened Model of the HTA in Fig. 2.2 After Phase III.

and $\widehat{\mathsf{P}}$ explicitly. We achieve that by introducing *duplications* of channels such that synchronization is guaranteed to happen between processes that correspond to parallel superstates. This can make it necessary to also introduce duplications of transitions.

For example, the HTA in Fig. 2.2 is flattened such that channel a is replaced by two copies a_parallel_P and a_parallel_MAIN. One can synchronize with superstates parallel to P and one with superstates parallel to MAIN. The signals a! and a? along channel a have to be replaced accordingly.

Parts of the flattened model are drawn in Fig. 5.11. If a superstate is both parallel to P and to MAIN, a transition originally carrying a! is replaced by two transitions, one carrying a_parallel_P! and one carrying a_parallel_MAIN!. The pseudo-code for this post-processing is given in Fig. 5.10.

6. Semantic Correspondence of HTAs and TAs

Hierarchical and flattened model are related in that with every hierarchical configuration we can associate a flat one. We show that every hierarchical trace corresponds to a projection of a trace in the flattened version. A similar connection holds in the other direction. It follows that both models are equivalent with respect to the TCTL properties checkable with Uppaal.

6.1 Hierarchical and Flat Configurations

Conceptually we can relate a configuration of a HTA M to a configuration of the flattened Uppaal model $\widehat{M}$. The reverse direction is not possible in general; some configurations of the Uppaal model do not make sense from the HTA point of view, e.g., if a process corresponding to a substate is active but not the process corresponding to its superstate. Our construction guarantees that those configurations are not reachable. Other configurations in the UPPAAL model are intermediate steps in the encoding of an exit or entry. We call those configurations of the UPPAAL model that have a counterpart in the hierarchical model *stable*.

DEFINITION 14. (STABLE/UNSTABLE CONFIGURATION)
Given a HTA M and a corresponding UPPAAL *model $\widehat{M}$, where every superstate S in M corresponds to the process $\widehat{S}$ in $\widehat{M}$. A* stable configuration *of $\widehat{M}$ then is a configuration $(\vec{l}, e, \nu)$, where*

- *No $l \in \vec{l}$ is committed, i.e., $\forall i.\ \neg c(l_i)$,*
- *If X is a XOR superstate and for some S X*_`ACTIVE_IN`_*$S \in \vec{l}$, then X*_`IDLE` *$\notin \vec{l}$, and*
- *If A is a AND superstate and for some S A*_`ACTIVE_IN`_*$\widehat{S} \in \vec{l}$, then for every substate B_i of A: B_i*_`IDLE` *$\notin \vec{l}$.*

Every consistent UPPAAL *model configuration that is not stable is called* unstable.

We can define a relation of configurations of a HTA M to stable configuration of $\widehat{M}$.

DEFINITION 15. (MATCHING CONFIGURATION)
Given a HTA M and a proper configuration $\mathbf{c} := (\rho, \mu, \nu, \theta)$ of it. A configuration $\mathbf{s} := (\vec{l}, e, \nu)$ of $\widehat{M}$ is a matching configuration, *in symbols $\mathbf{c} \sim^M \mathbf{s}$ if the following holds.*

(i) $\forall S \in \rho^+(root).\ BASIC(S) \ \Rightarrow\ \widehat{S} \in \vec{l}$,

(ii) $\forall S \in \rho^+(root).\ XOR(S) \vee AND(S) \ \Rightarrow\ S\_\texttt{ACTIVE\_IN}\_(\eta^{-1}(S)) \in \vec{l}$, *and*

(iii) $\forall v \in Var(root).\ \mu(v) = e(v)$

It is easy to see that the flat configuration in the above definition is necessarily stable. The relation $\sim^M$ ignores history and the values of auxiliary variables. In general $\sim^M$ is an injection. By construction of the steps, however, for every reachable hierarchical configuration $\mathbf{c}$ only one flat configuration $\mathbf{s}$ is reachable.

6.2 Correspondence of Steps

The flattened version $\widehat{M}$ of a HTA M is a *refinement* in the sense that every step in M corresponds to a finite sequence of steps in $\widehat{M}$. If an ordinary transition or a delay is mimicked this sequence is of length 1. The exit and entry of superstates require a larger number of steps to be taken in the flattened version.

Delay. A delay step of duration d is possible if no urgent transition is enabled and all invariants remain **true** throughout this delay. In phase I, all invariants of superstates are inherited, i.e., every location in the flattened model carries a conjunction of the invariants of all ancestor superstates it is derived from. Thus, a duration step from a HTA configuration $\mathbf{c}$ is possible if and only if it is possible in a corresponding flat $\mathbf{s}$ with $\mathbf{c} \sim^M \mathbf{s}$.

Join. The computation of *PreExitSets*(e) in Section 2.2 corresponds to the sets of locations that are computed in *expandGlobalJoins*. Recall that *PreExitSets*(e) is a family of sets of basic locations. The global join can be taken if control is such that one location in each set is active. These sets are locations in the same *XOR* superstate, thus not more than one can be active. For the global join $\mathbf{g}_i$ the auxiliary variables ($\texttt{trigger}_i$) reflects the number of locations that are in the sets of $\mathbf{g}_i$, i.e. *PreExitSets*(e). If this number reaches the threshold $|\mathit{PreExitSets}(e)|$, the global join can be taken.

Every such performed global join relies on one proper transition t that does not lead to an exit. t is necessarily part of a *XOR* superstate X. The encoding of the global join is a chain of transitions (like in Fig. 5.9 (b)). The first transition carries guard and synchronization of t. The subsequent transitions signal the substates B_i of X to become idle, i.e., the processes $\widehat{B_i}$ corresponding to these substates take a transition to $B_i\texttt{\_IDLE}$. Since the intermediate locations of the chain are declared committed, this sequence cannot be disturbed by ordinary transitions or time delays.

If t synchronizes (with a transition parallel to t) this can entail two simultaneous executions of global joins and, possibly, also entries of substates. Since the transitions are necessarily parallel (or: *independent*), this does not cause problems. There might be several legal sequences of transitions that lead to the same next stable configuration.

Transition. A simple action step that does not exit or enter any superstates corresponds naturally to taking one transition in a (flat) process. In the flattened model, auxiliary variables (`trigger`) are updated along this transition. This is merely housekeeping and does not enable or block transitions. The invariants of locations are inherited. Thus the transition part of the HTA is directly mimicked in the translation.

The analogous argument holds for the synchronization of two transitions along a channel. The renaming in phase III guarantees that synchronizations are only possible between transitions that correspond to parallel transitions in the HTA.

Fork. Entries of *XOR* superstates activate one location that can be basic or a superstate. Entries of *AND* superstates activate all substates; those are necessarily superstates again. Thus every entry can result in the activation of a set of superstates. This set is given by the (static) structure.

In the flattened version this set of superstates is activated by adding auxiliary locations and synchronizing via $\texttt{enter\_}B\texttt{\_in\_}S\texttt{\_via\_}e!$. There are no guards allowed and the auxiliary locations are declared committed. Thus this sequence of synchronizations takes place without interleaving with ordinary transitions and without time delay.

It is important that all parts, once started, can execute to completion. Thus we can relate one step in a HTA M to a sequence of steps in $\widehat{M}$, where only the first and the last configurations are stable.

Lemma 1. (Step Encoding)
For a HTA M there exist a step between two configurations (ρ, μ, ν, θ) and $(\rho', \mu', \nu', \theta')$ according to rules action and sync (see Section 2.2) if and only if for the Uppaal *model $\widehat{M}$ there exists a corresponding sequence*

$$(\vec{l}, \mathrm{e}, \nu) \overset{\alpha}{\Longrightarrow} (\vec{l}_1, \mathrm{e}_1, \nu_1) \overset{\tau}{\Longrightarrow} \cdots \overset{\tau}{\Longrightarrow} (\vec{l}_k, \mathrm{e}_k, \nu_k) \overset{\tau}{\Longrightarrow} (\vec{l}', \mathrm{e}', \nu')$$

where $(\rho, \mu, \nu, \theta) \sim^M (\vec{l}, \mathrm{e}, \nu)$, $(\rho', \mu', \nu', \theta') \sim^M (\vec{l}', \mathrm{e}', \nu')$, all $(\vec{l}_i, \mathrm{e}_i, \nu_i)$ are unstable configurations, $\alpha \in \{a, \tau\}$ and the remaining synchronizations τ are along channels `exit_`B *and* `enter_`B`_in_`S`_via_`e.

Other modeling elements. We do not address history or urgency in our argumentation. This is for the sake of clarity; they are not causing complications.

History amounts to the assignment of special variables that direct control on re-entry. In the flattened version this yields a mutual exclusive choice of the transitions from the history entry to exactly one location (which can be in fact a superstate; then either the history entry or default entry is used). Along this transitions only those clocks declared as forgetful are reset to 0 and all others remain untouched.

Urgency can be completely replaced by Uppaal's mechanism for synchronization on urgent channels as explained earlier.

6.3 Correspondence of Traces

After asserting that the step relation of a HTA M is indeed refined to the step relation of the flattened $\widehat{M}$, we can relate the sets of traces. The key observation is that for every timed trace in M there exits at least one corresponding timed traces for $\widehat{M}$. For every timed trace for $\widehat{M}$ there exists exactly one timed trace for M.

The trace relation is not a bijection, since in $\widehat{M}$ interleavings between the intermediate transitions are possible. This is only the case for synchronized action steps, which are guaranteed to connect only independent transitions. Thus all such interleavings lead to the same stable configuration.

Proposition 1. (Correspondence of Hierarchical and Flattened Model)

Given a HTA M and the flattened Uppaal *model $\widehat{M}$ of it. For every timed trace $\sigma = \{(\rho, \mu, \nu, \theta)_i\}_{i \geq 0}$ of M there exists a corresponding timed trace $\widehat{\sigma} = \{(\vec{l}, \mathrm{e}, \nu)_j\}_{j \geq 0}$ of $\widehat{M}$ such that*

$$\begin{array}{lll} \forall i.\ \exists k, k',\ k < k'. & (\rho, \mu, \nu, \theta)_i \sim^M (\vec{l}, \mathrm{e}, \nu)_k & \wedge \\ & (\rho, \mu, \nu, \theta)_{i+1} \sim^M (\vec{l}, \mathrm{e}, \nu)_{k'} & \wedge \\ & \forall k < j < k'.\ (\vec{l}, \mathrm{e}, \nu)_j \textit{ is unstable.} & \end{array}$$

Conversely, for every timed trace $\widehat{\sigma} = \{(\vec{l}, \mathrm{e}, \nu)_j\}_{j\geq 0}$ *of* $\widehat{M}$ *there exists a corresponding timed trace* $\sigma = \{(\rho, \mu, \nu, \theta)_i\}_{i\geq 0}$ *of* M *such that*

$$\begin{aligned}\forall k, k',\ k < k'.\ \ & \textit{if } (\vec{l}, \mathrm{e}, \nu)_k \textit{ and } (\vec{l}, \mathrm{e}, \nu)_{k'} \textit{ are stable} \\ & \textit{and all } (\vec{l}, \mathrm{e}, \nu)_j \textit{ with } k < j < k' \textit{ are unstable, then} \\ & \exists i.\ (\rho, \mu, \nu, \theta)_i \sim^M (\vec{l}, \mathrm{e}, \nu)_k \quad \wedge \\ & \quad (\rho, \mu, \nu, \theta)_{i+1} \sim^M (\vec{l}, \mathrm{e}, \nu)_{k'}.\end{aligned}$$

Observe also that by construction the entries and exits cannot get "stuck" in the middle of the transition. Thus $\widehat{M}$ does not yield maximally extended finite traces that terminate in unstable configurations. This entails that all trace properties, that Uppaal can establish for $\widehat{M}$, also hold for M.

Corollary 1. (Flattening Sound and Complete)
A timed property φ *from the TCTL fragment in Section 4 holds in an hierarchical model* M *if and only if the the corresponding property* $\widehat{\varphi}$ *holds in* $\widehat{M}$*.*

Proof. **(Sketch)**
By Proposition 1 the sets of traces match modulo the unstable configurations contained in the traces of $\widehat{M}$. Local properties of M cannot refer to the auxiliary variables in the unstable configurations and by our well-formedness conditions the values of variables in *Var*(*root*) change at most once along a sequence of unstable configurations.

For the TCTL fragment in Section 4 it suffices to quantify over traces. The hierarchical and the flat traces are only distinguishable by the names of identifiers. Those we assume to be translated properly in $\widehat{\varphi}$. □

7. Case Study: A Cardiac Pacemaker

We exemplify our flattening procedure on the model of a cardiac pacemaker. The flattened version is model checked with Uppaal for a safety and a liveness property.

The pacemaker is put in parallel with a model of a human heart and a programmer. 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. Then we report on run-time data of the formal verification of this translation with respect to safety and response properties.

7.1 The Hierarchical Timed Automaton Model

The hierarchical model is a parallel composition of three *XOR* superstates: the human heart, the cardiac pacemaker itself, and a programmer setting up the pacemaker.

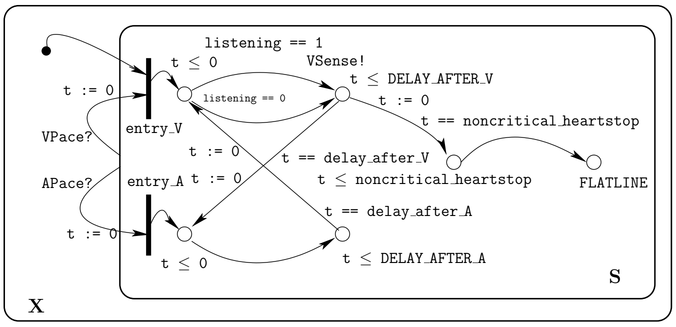


Fig. 7.12: Model of a Human Heart That Might Require Pacing.

Heart model. The human heartbeat is in fact a complex sequence of chamber contractions, where two *atrial* and two *ventricular* chambers collaborate to establish blood circulation. We use a simplified model of a human heart, that might require pacing (Fig. 7.12). We consider only two chambers, namely the (left) atrial and ventricular ones. A healthy heart contracts those in a steady rhythm. We mimic this by the time delays `DELAY_AFTER_V` and `DELAY_AFTER_A` and the local clock `t`. In our example we only monitor the ventricular chamber. The part after `entry_V` synchronizes on `VSense`,

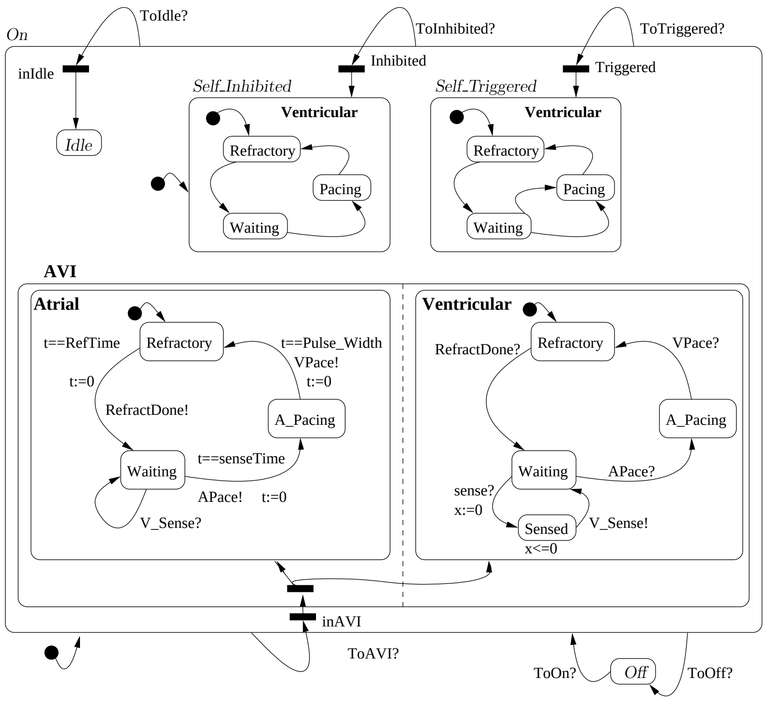


Fig. 7.13: Model of the Pacemaker. Initially *Self_Inhibited* is Entered.

in case that anybody is listening (indicated by `listening == 1`).

After the contraction of the ventricular chamber, our heart 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, regardless on when these signals occur. This is modeled by using the default exit and re-entering at one of the leftmost locations.

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 a signal 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.

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. In the *Self_Triggered* (T) modes, a pacing pulse will always occur, triggered either 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. In our example only the ventricular chamber is observed, but a pace signal may be sent either chamber.

Programmer model. A medical person—here called the *programmer*—is responsible for switching the pacemaker on/off and for selecting the operation mode. This the programmer does via the signals `commandedOn!`, `commandedOff!`, `toIdle!`, `toVVI!`, `toVVT!`, and `toAVI!`. We do not make assumptions, on how or in which order she issues the signals. However, we 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 *XOR* superstate with two locations. In the initial location, `Modeswitch`, any signal can be issued while entering the second location. The second location is left after exactly `DELAY_AFTER_MODESWITCH` time units. We include two additional locations, *Random* and *Idle*, to encode

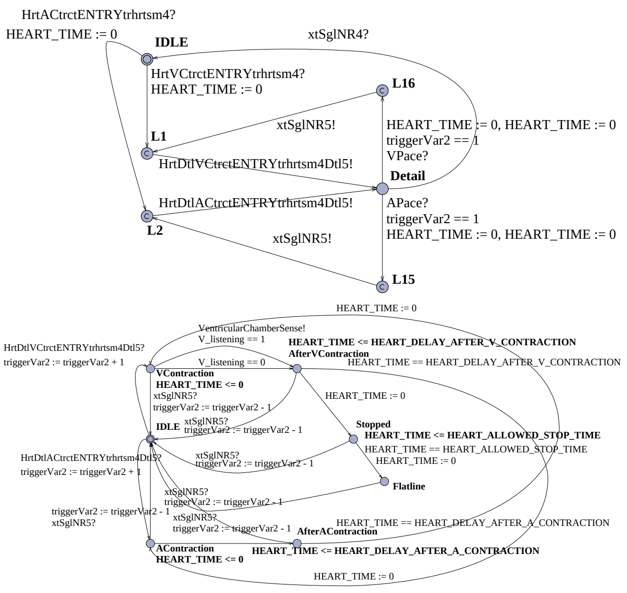

HrtACtrctENTRYtrhrtsm4?
HEART_TIME := 0
IDLE
xtSglNR4?
HrtVCtrctENTRYtrhrtsm4?
HEART_TIME := 0
L16
xtSglNR5!
HEART_TIME := 0, HEART_TIME := 0
triggerVar2 == 1
VPace?
L1
HrtDtlVCtrctENTRYtrhrtsm4Dtl5!
Detail
HrtDtlACtrctENTRYtrhrtsm4Dtl5!
APace?
triggerVar2 == 1
HEART_TIME := 0, HEART_TIME := 0
L2
xtSglNR5!
L15
HEART_TIME := 0
VentricularChamberSense!
V_listening == 1
HrtDtlVCtrctENTRYtrhrtsm4Dtl5?
triggerVar2 := triggerVar2 + 1
HEART_TIME <= HEART_DELAY_AFTER_V_CONTRACTION
AfterVContraction
V_listening == 0
HEART_TIME == HEART_DELAY_AFTER_V_CONTRACTION
VContraction
HEART_TIME <= 0
xtSglNR5?
triggerVar2 := triggerVar2 - 1
HEART_TIME := 0
IDLE
xtSglNR5?
triggerVar2 := triggerVar2 - 1
Stopped
HEART_TIME <= HEART_ALLOWED_STOP_TIME
HEART_TIME == HEART_ALLOWED_STOP_TIME
HEART_TIME := 0
HrtDtlACtrctENTRYtrhrtsm4Dtl5?
triggerVar2 := triggerVar2 + 1
xtSglNR5?
triggerVar2 := triggerVar2 - 1
Flatline
xtSglNR5?
triggerVar2 := triggerVar2 - 1
triggerVar2 := triggerVar2 - 1
xtSglNR5?
xtSglNR5?
triggerVar2 := triggerVar2 - 1
HEART_TIME == HEART_DELAY_AFTER_A_CONTRACTION
AfterAContraction
AContraction
HEART_TIME <= 0
HEART_TIME <= HEART_DELAY_AFTER_A_CONTRACTION
HEART_TIME := 0

Fig. 7.14: Flattened Version of the Heart Model.

alternative behavior of the programmer. They are not relevant here.

7.2 Translation to UPPAAL *Timed Automata*

The three superstates `Heart`, `Pacemaker`, and `Programmer` are flattened to a network of UPPAAL processes. In particular this translation yields

- ◦ two processes for the `Heart`: a top-level, where exit and re-entry happens and one for the substate where the heart is beating (Fig. 7.14),
- ◦ seven processes for the `Pacemaker`, put together as
  - – one process for the top-level where the pacemaker is either *On* or *Off* (Fig. 7.15),
  - – one process for superstate where the pacemaker is on (Fig. 7.16),
  - – one process for the VVI operation mode (Fig. 7.17),
  - – one process for the VVT operation mode (Fig. 7.18),
  - – three processes for the AVI operation mode, one for the *AND* superstate (Fig. 7.19) and two for the substates listening to the ventricular chamber (Fig. 7.20) and pacing the articular chamber (Fig. 7.21),
- ◦ one process for the `Programmer` (Fig. 7.22), and
- ◦ one process to start the three parts (Fig. 7.23).

Translation of heart (Fig. 7.14). The *XOR* superstate **X** and the *XOR* substate **S** are translated to the two processes. The translation of **X** (upper part of Figure) is responsible for selecting the entry `VContraction` or

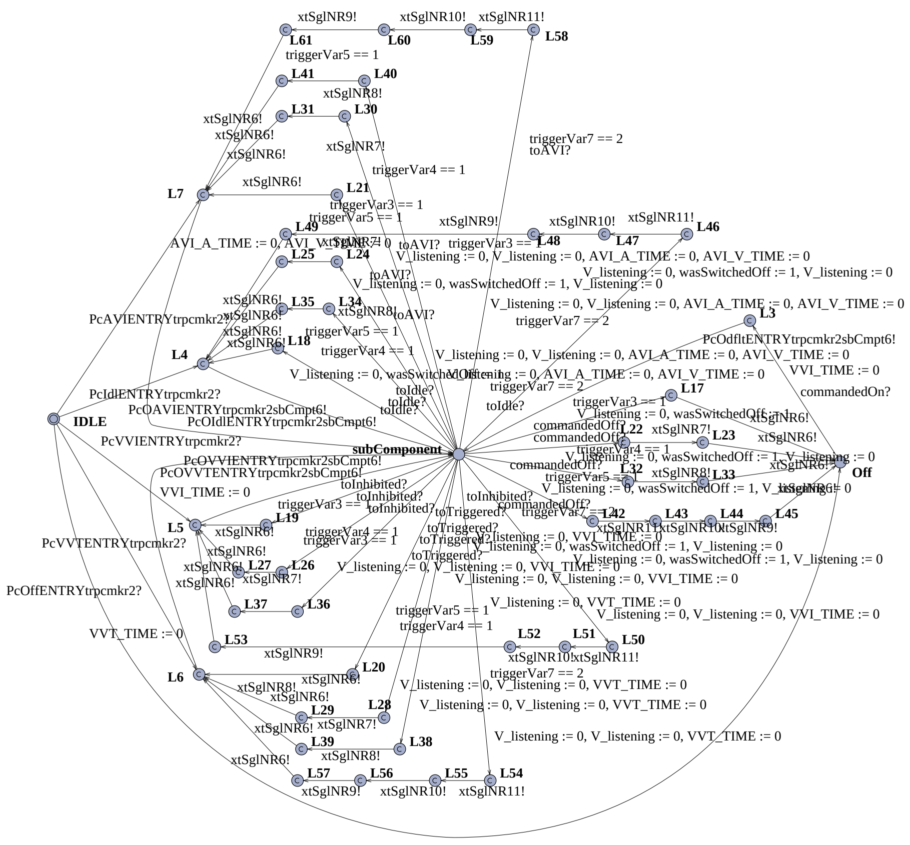
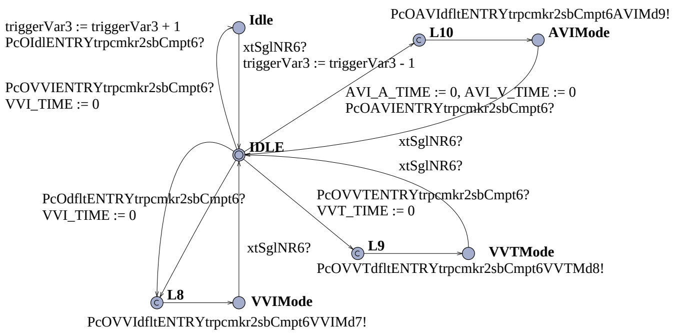


Fig. 7.15: Translation of the Topmost *XOR* Superstate of the Pacemaker.

`AContraction`. The translation of **S** (lower part of Figure) encodes the behavior. Note that from every location there is a transition to `IDLE`; this corresponds to the default exit of **S**.

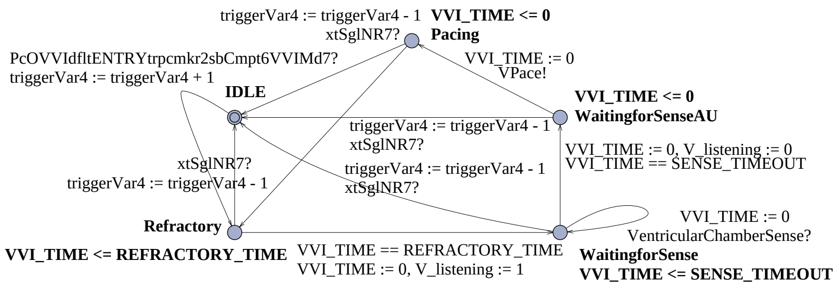
Flattened pacemaker (Figures 7.15, 7.16, 7.17, 7.18, 7.19, 7.20, 7.21). The most complicated process is the translation of the topmost *XOR* superstate. The basic locations are `IDLE` (far left), `subComponent` (center), and `Off` (far right). The pacemaker is *on*, when it control resides in `subComponent` and *off*, when the control is at `Off`.

The committed locations serve to encode the entry of the single substate and the global joins originating from it. For example, the four locations on the left `L4`, `L5`, `L6`, and `L7` correspond to entering the modes `Idle`, `VVIMode`, `VVTMode`, and `AVIMode`. Control of the pacemaker 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.



Idle
triggerVar3 := triggerVar3 + 1
PcOIdlENTRYtrpcmkr2sbCmpt6?
xtSglNR6?
triggerVar3 := triggerVar3 - 1
PcOAVIdfltENTRYtrpcmkr2sbCmpt6AVIMd9!
L10
AVIMode
PcOVVIENTRYtrpcmkr2sbCmpt6?
VVI_TIME := 0
AVI_A_TIME := 0, AVI_V_TIME := 0
PcOAVIENTRYtrpcmkr2sbCmpt6?
xtSglNR6?
IDLE
xtSglNR6?
PcOdfltENTRYtrpcmkr2sbCmpt6?
VVI_TIME := 0
PcOVVTENTRYtrpcmkr2sbCmpt6?
VVT_TIME := 0
xtSglNR6?
L9
VVTMode
PcOVVTdfltENTRYtrpcmkr2sbCmpt6VVTMd8!
L8
VVIMode
PcOVVIdfltENTRYtrpcmkr2sbCmpt6VVIMd7!

Fig. 7.16: Translation of the *XOR* Superstate *On*.

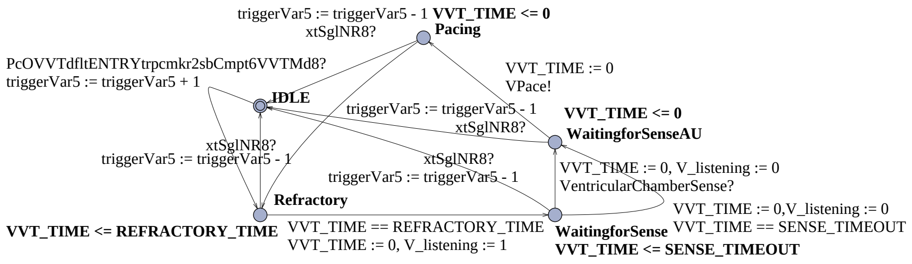


triggerVar4 := triggerVar4 - 1
xtSglNR7?
VVI_TIME <= 0
Pacing
PcOVVIdfltENTRYtrpcmkr2sbCmpt6VVIMd7?
triggerVar4 := triggerVar4 + 1
IDLE
VVI_TIME := 0
VPace!
VVI_TIME <= 0
WaitingforSenseAU
triggerVar4 := triggerVar4 - 1
xtSglNR7?
VVI_TIME := 0, V_listening := 0
VVI_TIME == SENSE_TIMEOUT
xtSglNR7?
triggerVar4 := triggerVar4 - 1
triggerVar4 := triggerVar4 - 1
xtSglNR7?
VVI_TIME := 0
VentricularChamberSense?
Refractory
VVI_TIME <= REFRACTORY_TIME
VVI_TIME == REFRACTORY_TIME
VVI_TIME := 0, V_listening := 1
WaitingforSense
VVI_TIME <= SENSE_TIMEOUT

Fig. 7.17: Translation of the *XOR* Superstate Corresponding to the VVI Mode.

The AVI mode is modeled by a *AND* superstate with two parallel *XOR* substates. In the translation this is reflected by a process with two non-committed locations `IDLE` and `ACTIVE` (Fig. 7.19) that synchronizes with two other processes AVI-A and AVI-V (Figures 7.21,7.20).

Translation of programmer (Fig. 7.22). Since the programmer is a *XOR* superstate with only basic locations, the translation is very similar. It contains the additional location `IDLE`.



triggerVar5 := triggerVar5 - 1
xtSglNR8?
VVT_TIME <= 0
Pacing
PcOVVTdfltENTRYtrpcmkr2sbCmpt6VVTMd8?
triggerVar5 := triggerVar5 + 1
IDLE
VVT_TIME := 0
VPace!
triggerVar5 := triggerVar5 - 1
xtSglNR8?
VVT_TIME <= 0
WaitingforSenseAU
xtSglNR8?
triggerVar5 := triggerVar5 - 1
xtSglNR8?
triggerVar5 := triggerVar5 - 1
VVT_TIME := 0, V_listening := 0
VentricularChamberSense?
VVT_TIME := 0,V_listening := 0
VVT_TIME == SENSE_TIMEOUT
Refractory
VVT_TIME <= REFRACTORY_TIME
VVT_TIME == REFRACTORY_TIME
VVT_TIME := 0, V_listening := 1
WaitingforSense
VVT_TIME <= SENSE_TIMEOUT

Fig. 7.18: Translation of the *XOR* Superstate Corresponding to the VVT Mode.

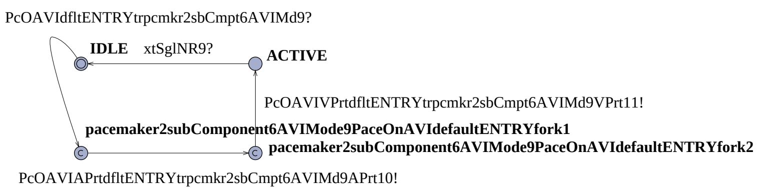

PcOAVIdfltENTRYtrpcmkr2sbCmpt6AVIMd9?
IDLE
xtSglNR9?
ACTIVE
PcOAVIVPrtdfltENTRYtrpcmkr2sbCmpt6AVIMd9VPrt11!
pacemaker2subComponent6AVIMode9PaceOnAVIdefaultENTRYfork1
pacemaker2subComponent6AVIMode9PaceOnAVIdefaultENTRYfork2
PcOAVIAPrtdfltENTRYtrpcmkr2sbCmpt6AVIMd9APrt10!

Fig. 7.19: Translation of the *AND* Superstate Corresponding to the AVI Mode.

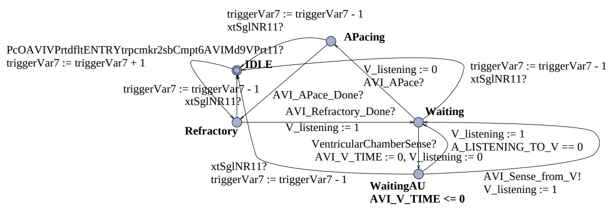


Fig. 7.20: Translation of the *XOR* Superstate AVI-V.

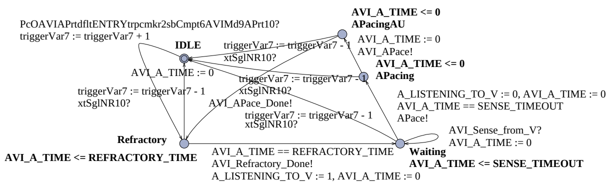

AVI_A_TIME <= 0
APacingAU
PcOAVIAPrtdfltENTRYtrpcmkr2sbCmpt6AVIMd9APrt10?
triggerVar7 := triggerVar7 + 1
IDLE
triggerVar7 := triggerVar7 - 1
xtSglNR10?
AVI_A_TIME := 0
AVI_APace!
AVI_A_TIME <= 0
APacing
AVI_A_TIME := 0
triggerVar7 := triggerVar7 - 1
xtSglNR10?
triggerVar7 := triggerVar7 - 1
xtSglNR10?
A_LISTENING_TO_V := 0, AVI_A_TIME := 0
AVI_A_TIME == SENSE_TIMEOUT
APace!
AVI_APace_Done!
triggerVar7 := triggerVar7 - 1
xtSglNR10?
AVI_Sense_from_V?
AVI_A_TIME := 0
Refractory
AVI_A_TIME <= REFRACTORY_TIME
AVI_A_TIME == REFRACTORY_TIME
AVI_Refractory_Done!
A_LISTENING_TO_V := 1, AVI_A_TIME := 0
Waiting
AVI_A_TIME <= SENSE_TIMEOUT

Fig. 7.21: Translation of the *XOR* Superstate AVI-A.

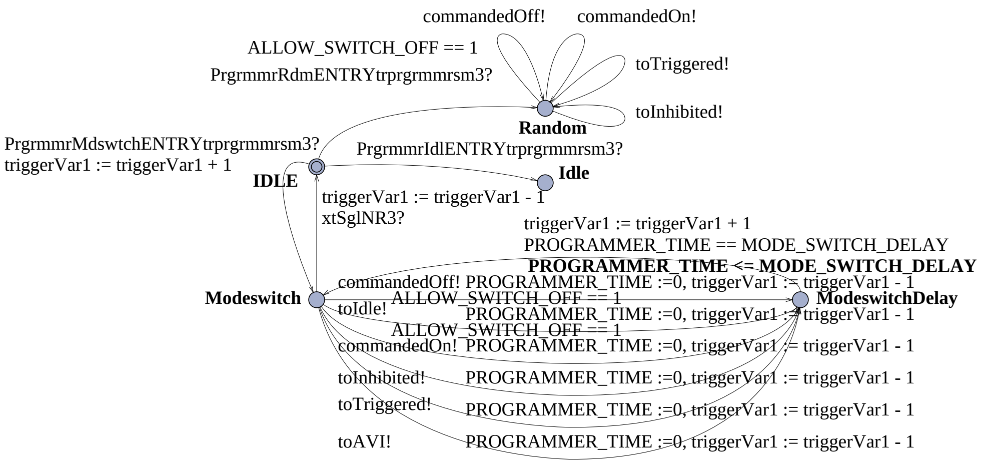

commandedOff!
commandedOn!
ALLOW_SWITCH_OFF == 1
PrgrmmrRdmENTRYtrprgrmmrsm3?
toTriggered!
toInhibited!
Random
PrgrmmrMdswtchENTRYtrprgrmmrsm3?
triggerVar1 := triggerVar1 + 1
PrgrmmrIdlENTRYtrprgrmmrsm3?
IDLE
Idle
triggerVar1 := triggerVar1 - 1
xtSglNR3?
triggerVar1 := triggerVar1 + 1
PROGRAMMER_TIME == MODE_SWITCH_DELAY
PROGRAMMER_TIME <= MODE_SWITCH_DELAY
commandedOff!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1
Modeswitch
ALLOW_SWITCH_OFF == 1
ModeswitchDelay
toIdle!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1
ALLOW_SWITCH_OFF == 1
commandedOn!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1
toInhibited!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1
toTriggered!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1
toAVI!
PROGRAMMER_TIME :=0, triggerVar1 := triggerVar1 - 1

Fig. 7.22: Translation of the *XOR* Superstate `Programmer`.

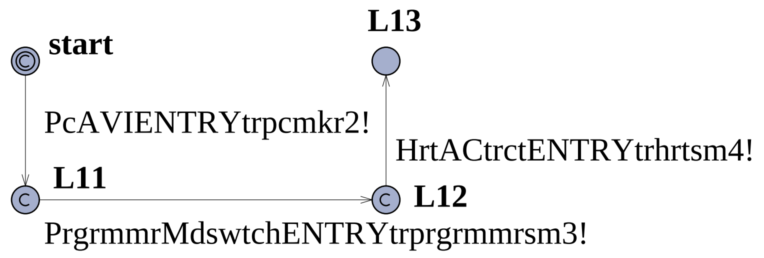


Fig. 7.23: The Additional `KickOff` Process.

	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 I: Size of the HTA Model and the Corresponding Uppaal Model.

Kickoff (Fig. 7.23). This process starts the three superstates `Heart`, `Pacemaker`, and `Programmer`. In the only process of the Uppaal model where in the initial configuration a transition is enabled.

Increase in Model Size

Both data formats, HTA and Uppaal timed automata, are described in terms of XML grammars. The flattening of the HTA yields an moderate increase in terms of model size. Table I 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 for the introduced auxiliary variables: The values of variables triggering global joins are completely determined by the current control state. The auxiliary channels introduced to switch components from `IDLE` to `ACTIVE` and vice versa does not increase the complexity significantly.

7.3 Model Checking the Uppaal *Model*

The translation of the HTA model can serve as input to the Uppaal tool. The system is not deadlock free. When the programmer switches off the pacemaker and the heart stops beating, a configuration is reached where unbounded delay is possible. 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 approximation 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.

(1) `A[] ( heart_sub.FLATLINE => (wasSwitchedOff == 1) )`

(2) `A[] ( heart_Sub.AfterAContraction =>`
`A<> heart_Sub.AfterVContraction )`

Property (1) 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 (2) 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. 7.24: Parameters That Yield Property (1).

Version 3.1.57 of the Uppaal tool is able to perform the model checking of both properties successfully in 11.83 respectively 4.26 seconds. The verification of the typically more expensive property (2) is faster, since here it is possible to apply a property preserving convex hull over-approximation, that is not preservative with respect to property (1). 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 (1) is strongly dependent on the parameter setting of the model. We use the constants from Fig. 7.24. If the programmer is allowed to switch between modes very fast, it is possible that she prevents the pacemaker from doing its job. E.g., for `MODE_SWITCH_DELAY = 65` the property (1) 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 yielding property satisfaction automatically, see [HRSV01].

Hierarchical structures are powerful formalisms; one indication for this is that there are many options on how to fill the details. This has been subject to intensive research [vdB94, Har97]. As we see it, the crucial choice in our semantics for HTAs is to treat cascades of entries and exits of superstates monolithically. This is somewhat clumsy, but allows for a conceptually simple correspondence between configurations of the hierarchical model and the flattened version.

Partially due to this decision, the reference implementation turned out to be surprisingly complicated. The source consists of more than 9000 lines of documented **Java** code, see `http://www.brics.dk/~omoeller/hta/vanilla-1/`. The high-level description given in this Chapter is a way to increase trust in our procedure and to allow for future maintenance.

The global join construction is a side effect of treating exit steps monolithically. We point out that entries and exits do *not* behave fully symmetric here. This is not an introduced problem; exiting more than one superstate implicitly requires synchronization. Giving conditions under which parts of a system to be entered is simpler than specifying at what point in time they can be left or interrupted. To the best of our knowledge this hat not been addressed before in the literature and we belief there is room for further elaboration on this topic.

In the pacemaker case-study, the increase in size of the generated model seems acceptable. Mainly entries and exits complicate Since we use committed locations to encode this it probably does not contribute significantly to the model checking time. The medium-sized model is sufficiently complicated to render the properties we model check non-trivial. The parameters that yield the safety property, e.g., were found experimentally. As for the usability of the flattened model, a lay-outer is desirable. The processes of the pacemaker case study are layouted by hand.

An alternative approach for model checking HTAs is to implement a model checking engine that operates directly on the hierarchical model. The configuration vector is more complicated to encode, but the sets of clock evaluations is not different from other dense-time formalisms. The algorithmic challenge is the implementation of superstate exits; basically the same computations as used in the global joins have to be performed. We consider it interesting to compare the run-times of model checking HTA models directly with those obtained after a flattening step. This would give an impression on how much overhead is really introduced by the flattening. There are plans in the DoCS group at Uppsala to address this, and we refer to their web-pages[1] for further information.

8. Conclusion

It is perceivable that there is a gap between industrial tools and academic tools. Industrial tools aim to support the design and production activity of their customers. The user interface has to be friendly; employees are going to interact with it for weeks and months. Academic tools aim to support research activity. Implementation is carried out by student programmers or

[1] `http://www.docs.uu.se/docs/index.eng.shtml`

PhD students. The user interface can be anything, even textual, since the typical user is either a researcher or a student.

The hierarchical timed automata formalism is neither the first nor the first timed variation of statecharts. A number of related approaches are compared and classified in [vdB94]. According to this classification, our formalism would be described by the column `g/t - (-) + - + - + - + o - + - i c + + + - - - + - - d`: graphical/textual, no negated trigger event, no (implicit) timeout event, timed transitions, no disjunction of trigger events, trigger conditions, no state reference, assignments to variables, no inter-level transition, history mechanism, operational semantics, not compositional, with synchrony hypothesis, not deterministic, interleaved concurrency, continuous time, globally consistent, causal, instantaneous states, no finiteness restriction in number of transitions, no priorities, no non-preemptive interrupt, preemptive interrupt, no distinction of internal and external events, no local events, discrete events.

We substitute "hand-shake synchronization" for "events" in van der Beeck's classification. The main motivation to construct this new formalism is the closeness to the UPPAAL model; a translation to UPPAAL exists, see Section 5. We found no existing statechart variant readily appropriate for this purpose. The major omission in HTAs with respect to UML statecharts are events.

There are two main difficulties with events. First, the *precise* notion of events has not (yet) been given in the UML, though version 1.4 is more specific than its predecessors. As a side effect some UML tools (e.g., RHAPSODY) do no longer correspond to this definition. Not all the holes are filled. In particular it is not specified yet if events are instantaneous or are queued and resolve at some later time. An unambiguous definition is a prerequisite for a formal treatment.

Second, if the event queue can grow without bound model checking is undecidable in general. This presents a serious problem, since no complete algorithm can be formulated any more. We argue that this is rather an introduced than an inherent problem. Due to constrained resources in running applications, the event queue usually has a bounded size. The exact bound, however, might not be known a priori. The approach of limiting the size of the event queues is followed in [Vot02].

Another possibility is to reason about event queues that have a certain regular structure. Sets of queue situations can have a finite encoding, though their cardinality is not finite. Here we refer to the work of Abdulla and Jonsson [AJ96, AJ01].

The work on the HTA formalism is continuing. A graphical editor for the language is currently under development at Aalborg University. It uses an XML representation of the described syntax. For practical reasons superstates are not constructed as primitives but generated from parameterized templates. More on this representation can be found in [DM01].

To assert the usability of the HTA formalism bigger examples are needed. However those are tedious to construct without an appropriate editor. We expect that the HTA formalism further evolves once the generation of examples has been made easier.

In the context of the AIT-WOODDES project the HTA formalism is planned to be used as an intermediate format. UML statechart models as constructed by the tool RHAPSODY are to be translated to UPPAAL via the HTA representation. This requires clearly an abstraction step. For once to safely omit code that is part of the model, and second to approximate events.

AIT-WOODDES: Advanced Information Technology—Workshop on Object-Oriented Design and Development of Embedded Systems. This is a project founded by the European Union, No IST-1999-10069. See `http://wooddes.intranet.gr`.

References

Rajeev Alur, Costas Courcoubetis, and David Dill. Model Checking in Dense Real-Time. *Information and Computation*, 104(1):2–34, 1993. A preliminary version appeared in the Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990).

Rajeev Alur and David L. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

Parosh Aziz Abdulla and Bengt Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, June 1996.

Parosh Aziz Abdulla and Bengt Jonsson. Ensuring Completeness of Symbolic Verification Methods for Infinite-State Systems. *Theoretical Computer Science*, 256(1–2), 2001.

Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

Patrik Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

Franck Cassez and Kim G. Larsen. The Impressive Power of Stopwatches. In *Proc. of CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science (LNCS)*, pages 138–152. Springer–Verlag, 2000.

Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001.

David Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987.

David Harel. Some Thoughts on Statecharts, 13 Years Later. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, pages 226–231. Springer–Verlag, 1997.

David Harel and Amir Pnueli. On the Development of Reactive Systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI*, pages 477–498, New York, 1985. Springer–Verlag.

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.

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 18th IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, December 1997.

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.

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.

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 (LNCS)*, pages 281–297. Springer–Verlag, 1998.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer–Verlag, 1999.

Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

Michael von der Beeck. A Comparison of Statechart Variants. In H. Langmaack, W. de Roever, and J. Vytopil, editors, *Formal Techniques in RealTime and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science (LNCS)*, pages 128–148. Springer–Verlag, 1994.

Angelika Votintseva. Specification-Based Test Generation for UML. to appear: Technical report, Universität Oldenburg (Abteilung Technische Informatik), 2002.

Februari 2003
ISSN 1404-3203
http://www.it.uu.se/