Chapter 6

Tools for Model-Checking Timed Systems

6.1. Introduction

In this chapter we present different tools for verification of timed systems. UPPAAL [LAR 97a, BEH 04b] is a tool for model-checking real-time systems developed jointly by Uppsala and Aalborg Universities. The first version of UPPAAL was released in 1995 [LAR 97a] and has been in constant development since then [BEN 98, AMN 01, BEH 01a, BEH 02, DAV 02, DAV 03, DAV 06]. It has been applied successfully to case-studies ranging from communication protocol to multimedia applications [HAV 97, LON 97, DAR 97, BOW 98, HUN 00, IVE 00, DAV 00, LIN 01]. The tool is designed to verify systems that can be modeled as networks of timed automata [ALU 90a, ALU 90b, HEN 92, ALU 94] extended with integer variables, structured data types, user defined functions, and channel synchronisation. UPPAAL-CORA is a specialized version of UPPAAL that implements guided and minimal cost reachability algorithms [BEH 01b, BEH 01c, LAR 01]. It is suitable in particular to cost-optimal schedulability problems [BEH 05a, BEH 05b]. UPPAAL-TIGA [BEH 07] is a specialization of UPPAAL designed to verify systems modeled as timed game automata where a controler plays against an environment. The tool synthesizes code represented as a strategy to reach control objectives [DEA 01, ASA 98, MAL 95, TRI 99]. The tool is based on a recent on-the-fly algorithm [CAS 05] and has already be applied to an industrial case study [JES 07]. The tool can also handle timed games with partial observability [CAS 07] and has been extended more recently to check for

Chapter written by Alexandre DAVID, Gerd BEHRMANN, Peter BULYCHEV, Joakim BYG, Thomas CHATAIN, Kim G. LARSEN, Paul PETTERSSON, Jacob Illum RASMUSSEN, Jiří SRBA, Wang YI, Kenneth Y. JOERGENSEN, Didier LIME, Morgan MAGNIN, Olivier H. ROUX and Louis-Marie TRAONOUEZ .

simulation of timed automata and timed game automata. TAPAAL [BYG 09] is an editor, simulator and verifier for timed-arc Petri nets and for the verification task it translates nets into networks of timed automata and uses the UPPAAL engine for the actual verification. It is developed at the Aalborg University.

ROMÉO [GAR 05b, LIM 09] is a tool for model-checking time Petri-nets [ZUB 80, ZUB 85, RAZ 85, BER 91b, ABD 01a, ABD 01b] and Petri nets with stopwatches[ROU 04]. It is developed by IRCCyN in Nantes. Since its first version in 2001, the software has benefited from regular improvements, both theoretical and experimental. Theoretical researches aim to widen the classes of models and properties on which the tool can perform model-checking. From an experimental point of view, developments focus on the use of up-to-date efficient libraries (e.g. the Parma Polyhedra Library [BAG 02]). The first releases of ROMÉO were mostly based on translations to other tools. The tool not only allows state space computation of TPN and on-the-fly model-checking of reachability properties, but it also performs translations from TPNs to Timed Automata (TA) that preserve the behavioural semantics (timed bisimilarity) of the TPNs. Recent research stresses on the emergence of autonomous model-checking algorithms. ROMÉO now provides an integrated TCTL model-checker and has gained in expressivity with the addition of parameters. Although there exists other tools to compute the state-space of stopwatch models, Romeo is the first one that performs TCTL model-checking on stopwatch models [MAG 08]. Moreover, it is the first tool that performs TCTL model-checking on timed parametric models [TRA 08]. Indeed, Romeo now features an efficient model-checking of time Petri nets using the Uppaal DBM Library, the model-checking of stopwatch Petri nets and parametric stopwatch Petri nets using the Parma Polyhedra Library and a graphical editor and simulator of these models. Furthermore, its audience has led to several industrial contracts, such as *DGA*, *SODIUS*, *Dassault Aviation* and *EADS*.

In this chapter we present the architecture of the tools, the basic model-checking algorithms, and the main techniques developped over the past years to improve performance both in time and space.

6.2. UPPAAL

6.2.1. *Timed Automata and Symbolic Exploration*

UPPAAL is based on an extension of timed automata. A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. A system is modeled as a network of such timed automata in parallel. Furthermore, the model is extended with (bounded) integer variables. The query language used to specify properties to be checked is a subset of TCTL (timed computation tree logic) [ALU 90a, HEN 94, BAI 08].

A state of the system is defined by the locations of all automata, the clock values, and the values of the integer variables. The system changes state by firing a transition. A transition may consist of one edge in any of the automata that can take such an edge, or several edges when a synchronization is involved (hand-shake or broadcast synchronization).

A timed automaton is a finite directed graph annotated with conditions over and resets of non-negative real valued clocks. We recall here the definition of a timed automaton (Definition 6.1). We omit here F and R.

DEFINITION 6.1 (TIMED AUTOMATON).– A *Timed Automaton* $\mathcal{A}$ is a tuple $(L, l_0, X, \Sigma_\varepsilon, E, Inv)$ where: L is a finite set of *locations*; $l_0 \in L$ is the *initial location*; X is a finite set of positive real-valued *clocks*; $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ is a finite set of *actions* and ε is the *silent* action; $E \subseteq L \times \mathcal{C}(X) \times \Sigma_\varepsilon \times 2^X \times L$ is a finite set of *edges*, $e = \langle l, \gamma, a, R, l' \rangle \in E$ represents an edge from the location l to the location l' with the guard γ, the label a and the reset set $R \subseteq X$; $Inv \in \mathcal{C}(X)^L$ assigns an *invariant* to any location. We restrict the invariants to conjuncts of terms of the form $x \preceq r$ for $x \in X$ and $r \in \mathbb{N}$ and $\preceq \in \{<, \leq\}$.

We recall that a clock valuation (1.2) ν over a set of variables X is an element of $\mathbb{R}^X_{\geq 0}$. For $\nu \in \mathbb{R}^X_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$, $\nu + d$ denotes the valuation defined by $(\nu + d)(x) = \nu(x) + d$, and for $X' \subseteq X$, $\nu[X' \mapsto 0]$ denotes the valuation ν' with $\nu'(x) = 0$ for $x \in X'$ and $\nu'(x) = \nu(x)$ otherwise. We give now the semantics of a timed automaton (Definition 6.2).

DEFINITION 6.2 (SEMANTICS OF A TIMED AUTOMATON).– The semantics of a timed automaton $\mathcal{A} = (L, l_0, C, \Sigma_\varepsilon, E, Act, Inv)$ is a timed transition system $S_\mathcal{A} = (Q, q_0, \Sigma_\varepsilon, \rightarrow)$ with $Q = L \times (\mathbb{R}_{\leq 0})^X$, $q_0 = (l_0, \mathbf{0})$ is the initial state, and $\rightarrow$ is defined by: *i)* the discrete transitions relation $(l, v) \xrightarrow{a} (l', v')$ iff $\exists\, (l, \gamma, a, R, l') \in E$ s.t. $\gamma(v) = \mathsf{tt}$, $v' = v[R \mapsto 0]$ and $Inv(l')(v') = \mathsf{tt}$; *ii)* the continuous transition relation
$(l, v) \xrightarrow{\epsilon(t)} (l', v')$ iff $l = l'$, $v' = v + t$ and $\forall\, 0 \leq t' \leq t,\ Inv(l)(v + t') = \mathsf{tt}$.

The problem when exploring is that the semantics of timed automata results in an infinite transition system. There exists an exact finite state abstraction based on convex polyhedra in $\mathbb{R}^X$ called zones [YI 94, LAR 95] (a zone can be represented by a conjunction in $\mathcal{C}(X)$). This abstraction leads to the following symbolic semantics of timed automata (TA).

DEFINITION 6.3 (SYMBOLIC SEMANTICS OF TA).– Let $Z_0 = Inv(l_0) \wedge \bigwedge_{x,y \in X} x = y = 0$ be the initial zone. The symbolic semantics of a timed automaton $(L, l_0, X, \Sigma_\varepsilon, E, Inv)$ over X is defined as a transition system $\langle \mathcal{S}, \int_0, \Rightarrow \rangle$ called the *symbolic reachability graph*, where $\mathcal{S} \subseteq L \times \mathcal{C}(X)$ is the set of symbolic states,

$\int_0 = (l_0, Z_0)$ is the initial state, $\Rightarrow$ is the transition relation and is defined by the following rules:

– $(l, Z) \overset{\delta}{\Rightarrow} (l, \mathsf{widen}(m, (Z \wedge Inv(l))^{\uparrow} \wedge Inv(l)))$, and

– $(l, Z) \overset{e}{\Rightarrow} (l', r(\gamma \wedge Z \wedge Inv(l)) \wedge Inv(l'))$ if $e = (l, \gamma, a, R, l') \in E$,

where $Z^{\uparrow} = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ (the *future* operation), and $r(Z) = \{[x \mapsto 0]u \mid x \in R, u \in Z\}$ (the *reset* operation). The function $\mathsf{widen} : \mathbf{N} \times \mathcal{C}(X) \to \mathcal{C}(X)$ widens the clock constraints with respect to the maximum constant m of the timed automaton. This operation is also called normalization [YI 94] or extrapolation.

The relation $\overset{\delta}{\Rightarrow}$ represents the delay transitions and $\overset{e}{\Rightarrow}$ the edge transitions. The classical representation of a zone is the Difference Bound Matrix (DBM) [ROK 93, WON 94, BEN 02].

In UPPAAL, timed automata are composed into a *network of timed automata* over a common set of clocks and actions, consisting of n timed automata $\mathcal{A}_i = (L_i, l_i^0, X, \Sigma_\varepsilon, E_i, Inv_i), 1 \leq i \leq n$. A location vector is a vector $\bar{l} = (l_1, \ldots, l_n)$. We compose the invariant functions into a common function over location vectors $I(\bar{l}) = \wedge_i Inv_i(l_i)$. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l_i'. In the following we define the semantics of a network of timed automata (NTA).

DEFINITION 6.4 (SEMANTICS OF NTA).– Let $\mathcal{A}_i = (L_i, l_i^0, X, \Sigma_\varepsilon, E_i, Inv_i)$ be a network of n timed automata. Let $\bar{l}_0 = (l_1^0, \ldots, l_n^0)$ be the initial location vector. The semantics is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^X$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

– $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ if $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(\bar{l})$.

– $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i], u')$ if there exists $l_i \xrightarrow{\epsilon, \gamma, R_i} l_i'$ s.t. $u \in \gamma$,
$u' = [R_i \mapsto 0]u$ and $u' \in I(\bar{l}[l_i'/l_i])$.

– $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_j'/l_j, l_i'/l_i], u')$ if there exist $l_i \xrightarrow{\gamma_i, c?, R_i} l_i'$ and
$l_j \xrightarrow{\gamma_j, c!, R_j} l_j'$ s.t. $u \in (\gamma_i \wedge \gamma_j)$, $u' = [R_i \cup R_j \mapsto 0]u$ and $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

The symbolic semantics of timed automata is naturally extended to networks of timed automata. We omit conditions on S that define location vectors of 2^L as valid, that is, the locations may not belong to the same automaton. In addition, we omit for the sake of simplicity the integer variables that UPPAAL supports. The definitions would be extended by adding a set of integers and actions over these integers that would be part of the state. Further extensions supported by UPPAAL are:

– *Broadcast Channels* If a channel *c* is declared *broadcast* then one process taking the action *c!* synchronizes with all other processes that have enabled edges that can take the action *c?*.

– *Urgent Channels* If a channel is declared *urgent* then time cannot elapse in a given state if a transition involving an urgent channel is possible.

– *Urgent Locations* A state having an urgent location in its location vector cannot delay.

– *Committed Locations* A state having a committed location in its location vector cannot delay and must take a transition that leaves a committed location or deadlock.

Other syntactic contructions are defined on top as a convenience for the user. These include arrays of integer variables, clocks, or channels, user-defined functions and structured data types. For a more complete reference see the updated version of [BEH 04b] available on *www.uppaal.org*.

UPPAAL implements a symbolic exploration algorithm based on the symbolic semantics of timed automata. In the reachability 6.2.4 or liveness 6.2.5 algorithms UPPAAL computes successor states symbolically in the following way: When a transition is taken, the next state is delayed infinitely (if possible) and the invariant of the state is applied. This computes all the (timed) successors w.r.t. a given transition. Thus, in the algorithms, computing successors will refer to trying all possible actions followed by delay.

6.2.1.0.1. Example

We give as an example the well-known Fischer's mutual exclusion algorithm [ABA 92, KRI 96] with two processes for the sake of simplicity. The protocol scales with any number of processes, each having its own identifier (in the example only 1 and 2). Figure 6.1 shows the model of the protocol. The processes want to avoid being in

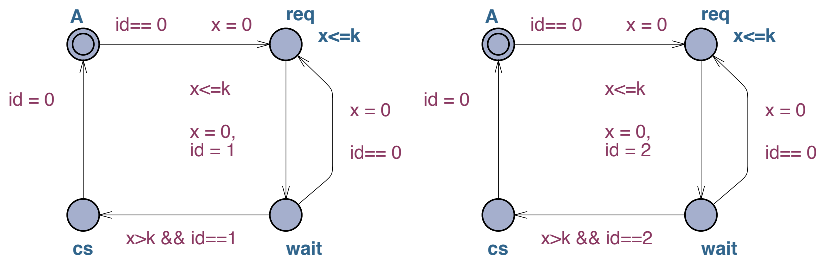


Figure 6.1. *Timed automata model of Fischer's mutual exclusion protocol.*

their critical sections (`cs`) at the same time. The protocol is using a shared idenfier (`id`) to choose which process should access the critical section and a clock to force the processes to wait at least k time units before entering `cs`. Processes may retry and go back to `req`. Figure 6.2 shows the symbolic exploration of the model. The

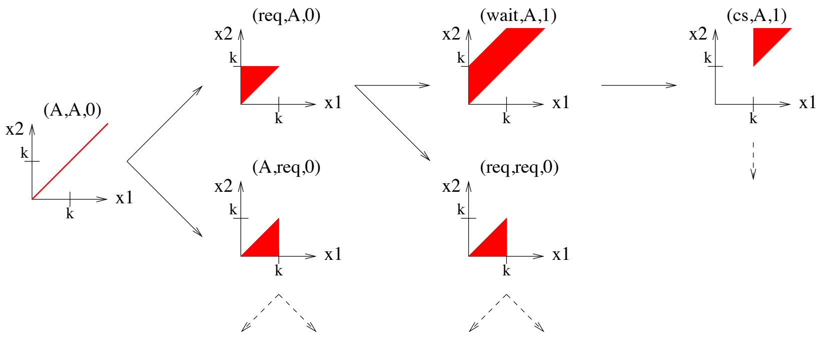


Figure 6.2. *Symbolic exploration of Fischer's mutual exclusion protocol.*

figure shows the locations and the integer variable as a tuple and the zone graphically. Resetting a clock corresponds to projecting the zone on the axis corresponding to that clock. Delaying corresponds to removing the upper bounds on the zone (but keeping the diagonal constraints). Applying a guard corresponds to intersecting a zone with the zone corresponding to the set of states described by the guard, or constraining the zone by the constraint of the guard. The (symbolic) initial state is delayed from the origin and the symbolic successors are computed from there. Either the first process goes to `req` or the second does, performing a reset of its clock, followed by a delay bounded by the invariant in `req`. From `(req,A,0)`, either the first process continues or the second tries to go to `req`. If the first process moves then again we have a reset followed by a delay but no invariant this time. From `(wait,A,1)` we apply the constraint of the guard to the zone and we obtain the states that can reach `(cs,A,1)`. The exploration continues from there.

6.2.2. *Queries*

The properties that can be checked by UPPAAL, as illustrated in Fig. 6.3 are defined in a subset of TCTL and are of the form:

– $A[]\ \phi$ "always globally ϕ",

– $E <>\ \phi$ "exists eventually ϕ",

– $A <>\ \phi$ "always eventually ϕ",

– $E[]\ \phi$ "exists globally ϕ", or

– $\phi\ {-}{-}{>}\ \psi$ "ϕ always leads to ψ", equivalent to $A[](\phi \rightarrow A <>\ \psi)$

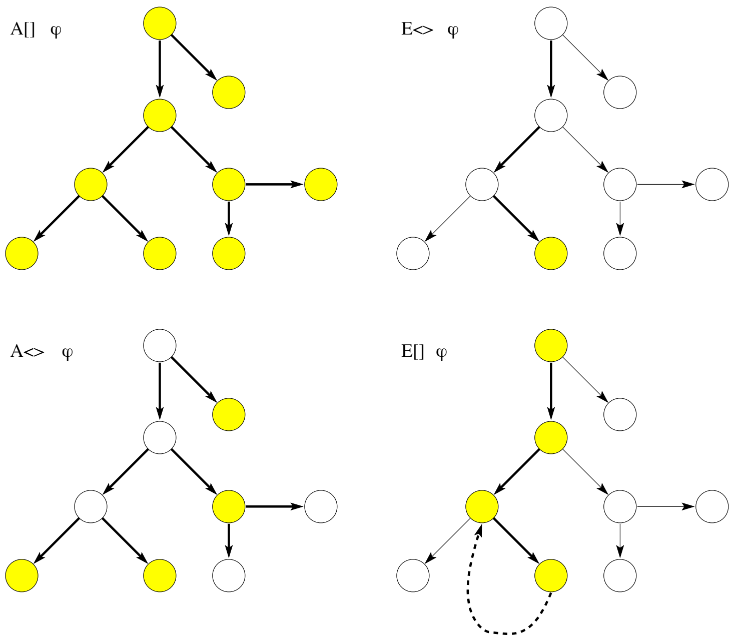


Figure 6.3. UPPAAL *basic queries.*

where ϕ and ψ are boolean expressions over locations, variables, and clocks. These queries are defined on paths: A applies for all paths and E for one existing path. "[]" queries all states along paths and $<>$ queries one state along paths. Figure 6.3 shows traces of states and paths for which CTL formulas hold. The filled states are those for which a given ϕ holds. Bold edges are used to show the paths the formulas evaluate on. The time part (**T**CTL) comes from the clock constraints used in ϕ (and ψ).

The formulas $A[]\ \phi$ and $E <>\ \ \phi$ are *reachability* properties and are symmetric: $A[]\ \phi = \neg E <>\ \ \neg\phi$. The $A[]\ \phi$ properties are also called safety properties because they check for a formula to hold for all the states. If such a property is not satisfied then $E\ <>\ \ \neg\phi$ characterizes counter-example paths. The reachability algorithm checks for $E <>\ \phi$.

The formulas $A\ <>\ \ \phi$ and $E[]\ \phi$ are *liveness* properties and are symmetric as well: $A <>\ \ \phi = \neg E[]\ \neg\phi$. These properties involve a loop detection algorithm because $E[]\ \phi$ holds for an infinite trace where every state satisfies ϕ. As the (symbolic) statespace is finite, we are looking for loops. The liveness algorithm checks for $E[]\ \phi$.

6.2.3. *Architecture of the Tool*

The tool is separated into two main components: the graphical user interface (GUI) – the client – and the model-checker engine – the server. The GUI is written in Java and is easily deployed on different platforms whereas the engine is recompiled for every platform. We refer here to the architecture of the engine [DAV 03], the performance critical part of the tool. The data structures of the engine are designed around a data flow centric architecture that forms a *pipeline*. The data going through the filters are states or states and a transition.

The different components, as shown in Fig. 6.4, are sources where states are created (typically the initial state), sinks where states disappear (typically checking an expression), buffers where states are pushed to and pulled from, and filters where states are pushed to and forwarded to the next component after processing. In addition, a pump shows where the main loop of the reachability or liveness algorithm executes.

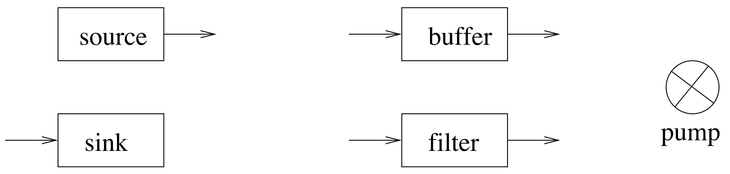


Figure 6.4. *Filter components in* UPPAAL.

The benefits of using pipeline components are *flexibility*, *code reuse*, and *efficiency*. The flexibility comes from the possibility to exchange a component for another to configure the pipeline to use, e.g., different storage structures to implement an exact exploration, an under-approximation or an over-approximation. Such dynamic configurations allows us to skip completely some stages in the pipeline if they are not necessary, e.g., storing traces. The code reuse comes from the reuse of the components accross different algorithms, e.g., the reachability and the liveness pipeline.

The common components that are used in the reachability 6.2.4 and the liveness 6.2.5 pipelines are:

– *Transition* that computes which transitions can be taken from a given states,

– *Successor* that fires a transition from a given state,

– *Delay* that computes the delay of state (if possible) bounded by its invariant,

– *Extrapolation + Active clock reduction* that applies the extrapolation according to the maximal (clock) constants of the model and at the same time active clock reduction. It appears that having locally $-\infty$ for the maximal constant of a given clock has the same effect as to free the constraints of that clock with the extrapolation algorithm.

6.2.4. *Reachability Pipeline*

The reachability algorithm is storing its states inside a so called *PWList* data structure [BEH 03b] that is unifying the traditional passed and waiting queues of model-checkers. Let us consider symbolic states to be of the form (l, Z) where l is the location vector and Z a zone. For simplicity we omit the integer variables. A traditional reachability algorithm would look like the one depicted in Fig. 6.5. Such an algorithm uses two main structures, namely the *passed* list to store previously explored states and the *waiting* list to store states to be explored. The algorithm starts with the initial state where $I(l_0)$ refers to its invariant. The expression $\forall(l', Z') : (l, Z) \rightarrow (l', Z')$ refers to computing all successors (l', Z') of (l, Z). The algorithm basically loops over the states in the waiting list, tests if the goal state has been found, computes the successors, and push the ones that are new to the waiting list. When this algorithm is implemented we have to put states in two hash tables (for efficient look-up) and make two inclusion checkings, i.e., we need to see if a states is included either in the passed or the waiting list before adding it. The second inclusion check is implicit in the algorithm when a state is added to the waiting list. Inclusion checking is crucial to improve performance (to avoid redundant exploration) but it costs $O(n^2)$ where n is the number of clocks.

```
$waiting = \{(l_0, Z_0 \wedge I(l_0))\}$
$passed = \varnothing$
**while** $waiting \neq \varnothing$ **do**
  $(l, Z) =$ select state from $waiting$
  $waiting = waiting \setminus \{(l, Z)\}$
  **if** $goal(l, Z)$ **then return true**
  **if** $\forall(l, Y) \in passed : Z \not\subseteq Y$ **then**
    $passed = passed \cup \{(l, Z)\}$
    $\forall(l', Z') : (l, Z) \rightarrow (l', Z')$ **do**
      **if** $\forall(l', Y') \in waiting : Z' \not\subseteq Y'$ **then**
        $waiting = waiting \cup \{(l', Z')\}$
      **endif**
    **done**
  **endif**
**done**
**return false**
```

Figure 6.5. *Traditional reachability algorithm.*

In contrast to having these two structures, a unified structure contains all the states but some of them are colored waiting while the others are colored passed (the implementation behind is using a colored state set). We note by (P, W) the unified structure.

P denotes all states (that are considered as passed) and W marks the subset of them that are waiting. A PW-List is described as a pair $(P, W) \in 2^S \times 2^S$, where S is the set of symbolic states, $W \subseteq P$, and the two functions $put : 2^S \times 2^S \times S \to 2^S \times 2^S$ and $get : 2^S \times 2^S \to 2^S \times 2^S \times S$, such that:

– $get(P, W) = (P, W \setminus \{(l, Z)\}, (l, Z))$ for some $(l, Z) \in W$.

– $put(P, W, (l, Z)) =$

$$\begin{cases} (P \setminus I, W \cup \{(l, Z)\}) & \text{if } \forall (l, Y) \in P : Z \not\subseteq Y \\ (P, W) & \text{otherwise,} \end{cases}$$

where $I = \{(l, Y) \in P \,\big|\, Y \subset Z\}$.

The get function removes states from W and leaves them in P. The put function removes the states of P that are included in the new state (the set I) and add this new state to W. Removing states from P implicitly removes them from W too because $W \subseteq P$. Similarly, states added to W are also added to P.

Figure 6.6 shows the simplified algorithm using the PW-List structure. Now there is no redundancy in the state-set and states are not between sets but merely re-colored. In addition, we need only one hash table and we have one inclusion check. In practice we only need a list of references to keep track of the subset W.

```
$(P, W) = \{(l_0, Z_0 \wedge I(l_0)), (l_0, Z_0 \wedge I(l_0))\}$
**while** $W \neq \varnothing$ **do**
  $(P, W, (l, Z)) = $ get(P, W)
  **if** $goal(l, Z)$ **then return true**
  $\forall (l', Z') : (l, Z) \to (l', Z')$ **do**
    $(P, W) = put(P, W, (l', Z'))$    **done**
**done**
**return false**
```

Figure 6.6. UPPAAL *reachability algorithm.*

The reachability pipeline of UPPAAL is based on the algorithm of Fig. 6.6 with the *PWList* structure at its center. The pipeline is depicted in Fig. 6.7. The pipeline computes symbolic successors, which is separated into computing transitions, firing them, delaying, and extrapolating. Expressions are evaluated at the end of the pipeline. The attentive reader notices that checking the goal in the implementation is not at the same place as in the algorithm. We do this after computing every successor. This way, we avoid going through the waiting list before finding out that the searched state is there and we can terminate earlier. The initial state (corresponding to the zero point) is inserted in the pipeline at the delay stage and then treated as a successor.

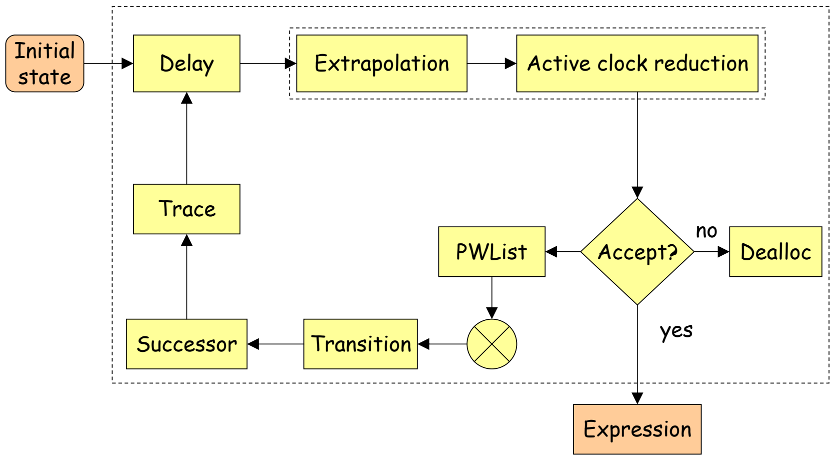


Figure 6.7. *The reachability pipeline of* UPPAAL.

6.2.5. *Liveness Pipeline*

The liveness algorithm is given in Fig. 6.8. The algorithm keeps track of a set of passed states P that verify $A <> \ \phi$ and a stack Stk of states that verify $\neg\phi$. The other states are not yet explored. The delay operation is special here and is restricted to states verifying $\neg\phi$. The goal of the algorithm is to find a loop of states that verify $\neg\phi$. Such a loop would be a counter-example to $A <> \ \phi$. If the algorithm finds a state for which there is an unbounded delay or a deadlock then this is also a counter-example. The reader notices that the recursive call is made with the proper action α in practice to keep track of the current path.

The liveness pipeline based on the algorithm of Fig. 6.8 is given in Fig. 6.9. Here the recursive call is unfolded with a waiting list that keeps track of transitions that need to be fired. As the algorithm shows, when the *Search* function returns, states are moved from the stack Stk to the set of passed states P. The main loop (the pump) explores and moves these states. The successor transitions are pushed to the waiting list. The source states are pushed to the stack if they are not explored, and they are also checked for unbounded delay or deadlock. If a state exits the pipeline, it is the entry to an infinite path, counter-example of $A <> \ \phi$.

```
proc Eventually($S_0, \phi$)
  $Stk = \varnothing$
  $P = \varnothing$
  Search($delay(S_0, \neg\phi)$)
  exit(true)
end

proc Search($S$)
  if $loop(S, Stk)$ then
    exit($false$)
  fi
  $S = S \wedge \neg\phi$
  $push(Stk, S)$
  if $unbounded(S) \vee deadlock(S)$ then
    exit($false$)
  fi
  if $\forall S' \in P : S \nsubseteq S'$ then
    foreach $S' : S \overset{\alpha}{\Rightarrow} S'$ do
      Search($delay(S', \neg\phi)$)
    od
  fi
  $P = P \cup \{pop(Stk)\}$
end
```

Figure 6.8. UPPAAL *liveness algorithm.*

6.2.6. *Leadsto Pipeline*

The leadsto pipeline is checking $A[](\phi \rightarrow A <> \ \psi)$ properties. The algorithm here is to launch a liveness check from all states that satisfy ϕ. This is implemented as a composition of the two previous pipelines as illustrated in Fig. 6.10. The reachability and liveness pipelines can be used as filter components themselves. States satisfying ϕ are pushed to the liveness pipeline that is fed by the expression ψ. A state that exits the liveness pipeline is an entry to an infinite path, counter-example of $A <>; \psi$.

6.2.7. *Active Clock Reduction*

Active clock reduction is a technique to remove irrelevant clock constraints to states. If a clock is reset before being tested later then its value is irrelevant before the reset.

DEFINITION 6.5 (INACTIVE CLOCK).– A clock x is *inactive* at a state S if on all paths from S, x is always reset before being tested. In practice, the reduction is done

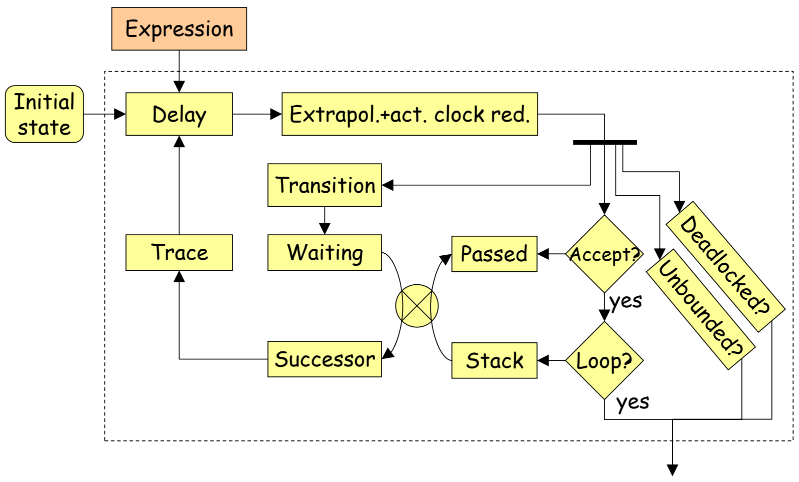


Figure 6.9. *The liveness pipeline of* UPPAAL.

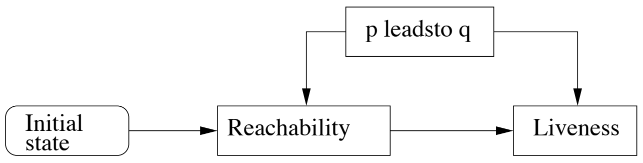


Figure 6.10. *The leadsto pipeline of* UPPAAL.

in two steps: i) There is a static analysis for every process that computes the set of active clocks for every location, and ii) during the verification the set of active clocks for a given state is obtained by computing the union of active clocks for the locations in the current location vector. Only clock constraints for the active clocks are saved in zones.

6.2.8. *Space Reduction Techniques*

6.2.8.0.2. Avoid Storing All States

One reduction technique (available with the `-S1` option of the model-checker) is to avoid storing all states. It turns out that, in order to ensure termination, we only need to store states that contain loop entries (in the *passed* list). Another more aggressive heuristic (option `-S2`)has been developed to store even fewer states while still ensuring termination [BEH 03a].

6.2.8.0.3. Sharing Data

States are tuples of the form (L, V, Z) where L is a location vector, V is a variable vector, and Z is a zone (DBM in practice). Although states are uniquely stored (if they are not included into a larger one w.r.t. their zones), it appears that the individual components L, V, and Z of the states are repeated among all the states. The reason behind it is that when changing state, a system often keeps either its location vector, variables, or zone. Therefor we can share these individual parts among all states. In practice, this gives typically 80% reduction in memory [BEH 03b].

6.2.8.0.4. Minimal Graph

We need to tighten the constraints of our DBMs to be able to check for inclusion (in $O(n^2)$, n being the number of clocks). The inclusion check is done by comparing all constraints c_{ij} and c'_{ij} by pairs. The tightening is done by running a shortest paths algorithm, typically Floyd's [FLO 62]. This tightening results in a unique representation of a given zone, the canonical form of DBMs. This is useful for storing DBMs uniquely but it consumes space in $O(n^2)$. Figure 6.11 shows what the shortest paths algorithm does (shortest paths closure from (a) to (b)) if we see a DBM as a graph with vertices being clocks and the edges from x_j to x_i being weighted by the constraints $x_i - x_j \leq c_ij$, which represents a distance. In [LAR 97b] a reduction was presented that reduces the number of necessary constraints to represent a zone. Applying this algorithm (shortest paths reduction from (b) to (c)) results in a minimal graph in the sense that the number of edges is minimal. This cost $O(n^3)$ in time. Although we still have $O(n^2)$ edges in the worst case, in practice we get $O(n)$ in average. UPPAAL stores this reduced number of edges and can restore the full DBM, i.e., the full graph, by running its shortest paths algorithm on it (and we get back to (b)).

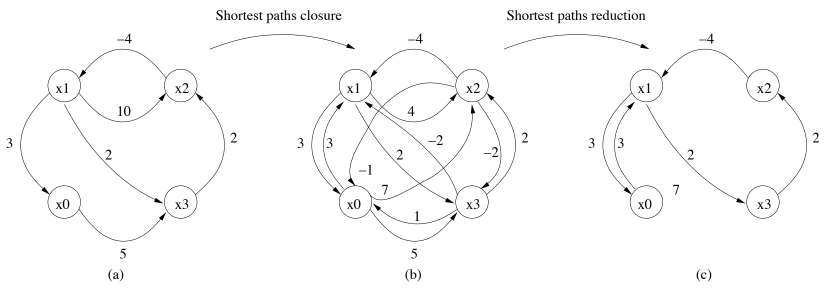


Figure 6.11. *Shortest paths closure and reduction.*

In a nutshell, the shortest paths reduction algorithm works in two main steps: i) It computes equivalence classes of the clocks, which is done by detecting zero cycles, and ii) it chooses one representant per equivalence class and removes redundant edges between them.

We note that computing the shortest paths algorithm costs $O(n^3)$ in time so it is vital for performance to avoid doing it if possible. It turns out that most operations on DBMs (delay, intersection, etc…) can be done on a canonical DBM in such a way that they preserve canonicity at no additional cost. We also note that we still do not know of any efficient way of applying these operations directly on the minimal graph. However, if the constraints of a (canonical) DBM are less or equal than the constraints (that are present) of a minimal graph then we can deduct that the DBM is included.

6.2.8.0.5. Symmetry Reduction

UPPAAL implements the algorithm presented in [HEN 03]. This algorithm finds equivalence classes of states w.r.t. symmetry (a.k.a. orbits) and chooses representant states by sorting the states (in the same equivalence class). The algorithm is implemented with the help of the *scalar* type that defines a set (of some size) of different but un-ordered scalar numbers. Figure 6.12 shows the experimental performance gains obtained on Fischer's protocol.

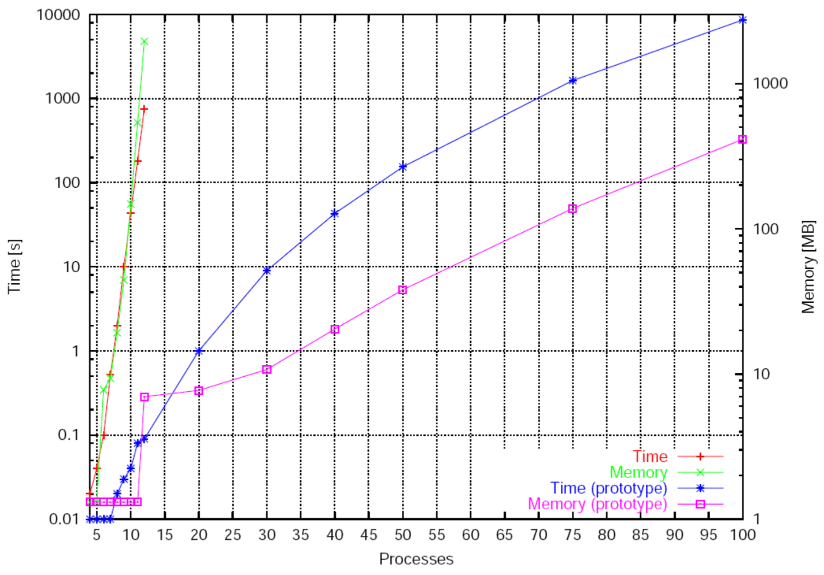


Figure 6.12. *Experimental results for the symmetry reduction on Fischer's protocol.*

6.2.9. *Approximation Techniques*

Sometimes systems are still too complex to be verified exactly. For these cases it is useful to apply approximation techniques.

6.2.9.0.6. Over-Approximation: Convex-hull

UPPAAL implements the convex-hull over-approximation technique [BAL 96] that consists in computing the convex-hull of some zones instead of keeping all of them. Figure 6.13 illustrates the following example: In some automaton there are two paths that lead to the state $S3$, thus giving two different zones that are depicted graphically. The technique stores the convex union of these two zones (in fact the smallest zone that contains both of these). Even though we are adding more states, the technique is still useful for safety properties.

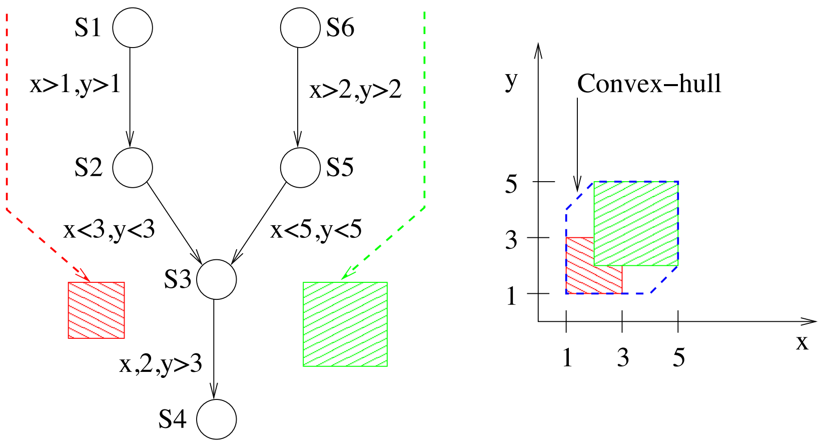


Figure 6.13. *Example of convex-hull.*

6.2.9.0.7. Under-Approximation: Bit-State Hashing

UPPAAL implements the bit-state hashing under-approximation technique [HOL 91, HOL 98]. This techniques consists in storing only one bit per state instead of its full location vector, variables, and zone. This is done by allocating a big hash table of size N (bits) initially filled with zeros and setting the $hash(state)\%N$ bit to one when states are visited (hash function applied to a state modulo N). There will be collisions that may conclude that a state was visited although it was not and avoid exploring it further. The technique is still useful for reachability properties.

6.2.10. *Extensions*

6.2.10.0.8. Robust Reachability

Traditionally, the verification is done considering all clocks perfect but in practice it is not the case and clocks are known to drift slightly over time. Specialized algorithms are needed to compute *robust* reachability analysis w.r.t. such drifts. One robust reachability algorithm is available in the recent development snapshot versions of the tool. The algorithm of [DAW 06] is implemented and is accessible via properties of the form $E <> * \phi$ and $A[] * \phi$.

6.2.10.0.9. Merging DBMs

The convex-hull technique is an over-approximation technique that reduces the number of zones dramatically. UPPAAL offers an *exact* technique [DAV 05] to merge DBMs on-the-fly. This technique replaces two or more DBMs by their convex-hull union when the union is exact in the sense that no extra states are added. States are merged in the passed list but also in the waiting list if possible. We note that although the data-structure we are using unifies waiting and passed states, we do not want to mix them when we merge to avoid duplicate exploration of states. This option is active by default in the development snapshot.

6.2.10.0.10. Stop-watches

Reachability analysis of timed automata augmented with stop-watches is undecidable but there is an efficient over-approximation technique to check such automata [CAS 00]. The technique consists in modifying the delay operator of DBMs such that clocks that are stopped keep their upper-bounds. Syntactically, the user adds to the invariant of a state expressions of the type $x' == expr$ where $expr$ evaluates to 0 or 1. This technique has proven useful in modeling schedulability problems since we want to check safety properties, e.g., deadlines are never missed. This extension is available in the development snapshot.

6.2.10.0.11. Supremum Values

When analyzing systems for worst case execution or response time, typically on models that schedule processes, it is useful to know maximal value of clocks that measure execution or response time. The development version of UPPAAL supports a special kind of property, namely, *sup: expr_list* where *expr_list* is a list of expressions that evaluate either to clocks or to integer values. The tool explores all the states and computes the maximal reached values for the integer variables or the maximal upper bound for the clocks.

6.2.10.0.12. Other Extensions

UPPAAL implements the generalized sweep line method [KRI 02]. The user needs to define progress measures to take advantage of it. Different extrapolation algorithms [BEH 04a] have been implemented. These approximations take advantage of maximal upper bounds as before but also maximal lower bounds. A distributed version of UPPAAL has been developed [BEH 00] that runs on clusters. From the model point-of-view, an acceleration technique has been developed that improves reachability analysis on models containing cycles [HEN 02].

6.3. UPPAAL-CORA

When computing traces that satisfy reachability properties, UPPAAL provides an algorithm for computing the time-wise shortest trace that satisfies the reachability property. This feature can be exploited for solving a number of general scheduling problems such as the famous travelling salesman problem.

UPPAAL-CORA is an extension of UPPAAL that performs minimum cost reachability analysis for timed automata models augmented with costs. These models, called priced timed automata, have been independently proposed and their reachability problems proven to be a decidable in [BEH 01b] and [ALU 01]. UPPAAL-CORA has been successfully applied to a number of scheduling case studies such as lacquor scheduling and aircraft landing, [BEH 05a, BEH 05b].

6.3.1. *Priced Timed Automata*

A priced timed automaton is defined similarly to a timed automaton (Definition 6.1) except that it is augmented with a cost function $\mathsf{Cost} : (L \cup E) \rightarrow \mathbb{N}$ that assigns a non-negative integral value to locations and edges. If we consider a network of priced timed automata, the semantics of the cost function is such that when delaying in a state, the cost grows by a rate given by the sum of the costs of the locations, and when taking a transition the cost grows by the sum of the costs of the edges involved. We note that costs grow monotonically.

To efficiently analyze priced timed automata using symbolic semantics, UPPAAL-CORA uses the notion of priced zones, indicated by $\mathcal{Z}$, [LAR 01]. Priced zones are convex abstractions over clock valuations similarly to regular zones together with an affine cost function over the zone. The cost function in UPPAAL-CORA is implemented as a linear combination of the intégral coefficients associated to the clocks of a zone. The use of priced zones complicates the computations of discrete and delay successors of symbolic states as these operations need zones to be split into sets of smaller disjoint zones in order to maintain the affinity of the cost function. This results in a significantly larger number of symbolic states that need to be explored. For a thorough description of the algorithms for computing delay and successors of symbolic states with priced zones, we refer to [LAR 01].

Figure 6.14 depicts the UPPAAL-CORA minimum cost reachability algorithm which outputs the minimum cost of satisfying a reachability property or ∞ if the

```
$(P, W) = \{(l_0, \mathcal{Z}_0 \wedge I(l_0)), (l_0, \mathcal{Z}_0 \wedge I(l_0))\}$
$cost = \infty$
**while** $W \neq \varnothing$ **do**
  $(P, W, (l, \mathcal{Z})) =$ get(P, W)
  **if** $goal(l, \mathcal{Z})$ **and** $mincost(\mathcal{Z}) < cost$ **then**
    $cost = mincost(\mathcal{Z})$
    **continue**
  **endif**
  **if** $mincost(\mathcal{Z}) + remain(l, \mathcal{Z}) < cost$
    $\forall (l', \mathcal{Z}') : (l, \mathcal{Z}) \rightarrow (l', \mathcal{Z}')$ **do**
      $(P, W) = put(P, W, (l', \mathcal{Z}'))$
    **done**
  **endif**
**done**
**return** $cost$
```

Figure 6.14. UPPAAL-CORA *minimum cost branch-and-bound reachability algorithm.*

property is unsatisfiable. The algorithm is a variation of the classical branch-and-bound algorithm fitted to the UPPAAL framework. The algorithm maintains a cost variable to keep track of the best solution found so far. The value of the cost variable is updated every time a better solution is found. The bounding of the algorithm is achieved by not exploring successors of states that cannot improve on the best solution. The allow for a heuristic search algorithm, UPPAAL-CORA lets the user to define remaining costs for states, i.e., a lower bound estimates on the cost required to satisfy the reachability property. If the remaining estimate is misused by not providing a valid lower bound estimate, the algorithm does not guarantee correctness.

The pipeline for UPPAAL-CORA is similar to the one of UPPAAL for the reachability algorithm depicted in 6.7. The difference is that the pipeline does not terminate upon finding a solution to the reachability problem but continues until there are no more states in the waiting list and reports the value of the cost variable as the solution to the reachability problem. Furthermore, there is no extrapolation operator defined for UPPAAL-CORA. The reader may then question why the algorithm terminates. Indeed, the guaranteed termination follows from two facts: First, it is well known that any timed automaton can be converted to a bounded timed automaton with upper bound invariants on all clocks. This means that the number of zones is finite. Second, given that the cost assignments of a priced timed automaton are integral, it is a known fact that cost functions over bounded zones are well-quasi ordered, meaning that for a given zone, there cannot exist an infinite sequence costs functions without one eventually included in a previous one of the sequence [LAR 01]. These facts combined guarantee the termination of the algorithm.

The minimum cost algorithm utilizes the PW-List data struture used in UPPAAL. In order to correctly use the PW-List data structure, the inclusion check needs to be modified to handle the cost information contained in priced zones. Obviously, it not enough for a priced zone $\mathcal{Z}$ to include another priced zone $\mathcal{Z}'$, the cost function of $\mathcal{Z}$ further needs to be consistently smaller than $\mathcal{Z}'$ in order for $\mathcal{Z}$ to include $\mathcal{Z}'$. In UPPAAL-CORA this check is implemented by solving the linear program given by minimizing the difference between the cost functions of $\mathcal{Z}'$ and $\mathcal{Z}$ over the zone of $\mathcal{Z}'$. If the solution is positive we know that $\mathcal{Z}$ includes $\mathcal{Z}'$. Moreover, the bounding of states with costs greater than the best found solution so far (potentially including a remaining estimate) is also implemented as part of the insertion into the PW-List.

One of the key aspects of the UPPAAL-CORA algorithm is solving the linear programs arising from inclusion checks on priced zones and computation of minimum costs of priced zones. Since zones have the property that they can be described solely by difference constraints, the linear programs can exploit this structure. It turns out that the minimization problem is basically the dual problem of the min-cost flow problem which is well known to have more efficient algorithms than general linear programming problems, [AHU 93, RAS 06]. Thus, UPPAAL-CORA converts every priced zone minimization problem to the related min-cost flow problem and solves this instead. This approach greatly increases the running time of the tool.

6.3.2. *Example*

Figure 6.15 depicts an example of how the cost rates of location and costs of edges are used in UPPAAL-CORA. Note that cost is a built-in variable and does not need to be declared. The cost increases continuously with a rate of 1 in A and 2 in B. In addition taking the transition to C adds 3 to the cost. The automaton could take immediately the transition to B but then it would have to wait in B where the cost rate is higher than in A. Moreover, the automaton must delay at least one unit of time in B because of y. The reader can convince herself that the minimal cost to reach C is 6, waiting one unit of time both in A and B.

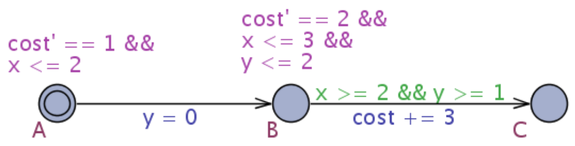


Figure 6.15. *An example of a* UPPAAL-CORA *model.*

6.4. UPPAAL-TIGA

6.4.1. *Timed Game Automata*

UPPAAL-TIGA implements the first efficient truly on-the-fly algorithm for solving timed games [CAS 05]. It is an extension of [LIU 98] with time. Our input models are specified as networks of Timed Game Automata [MAL 95] (TGA) where edges are marked either controllable or uncontrollable (see Fig. 6.16). This defines a two players game with on one side the *controller* and on the other side the *environment*. Winning conditions of the game are specified through TCTL formulas. The tool is designed to generate strategies for a controller to reach an objective or to maintain safety whatever the environment (playing as an opponent) does.

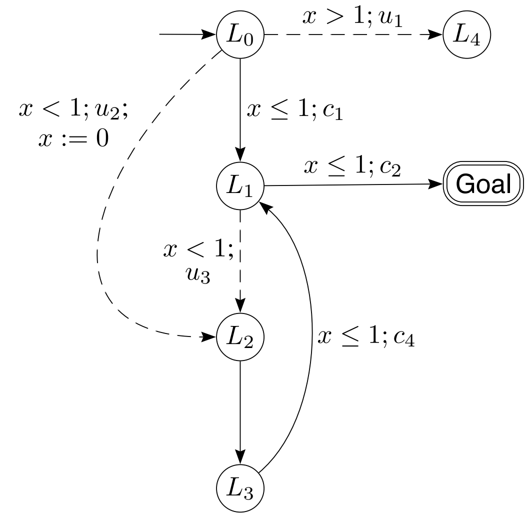


Figure 6.16. *An example of Timed Game Automaton.*

As an example, let us consider the timed game automaton of Fig. 6.16. It has one clock x and two types of edges: controllable (c_i) and uncontrollable (u_i). The reachability game consists in finding a strategy for the controller to reach the state Goal, no matter which uncontrollable transitions (u_i) the opponent takes. For all initial states of the form (l_0, x) with $x \leq 1$, there is such a strategy. This strategy consists in:

– taking c_1 immediately in all states (l_0, x) with $x \leq 1$;

– taking c_2 immediately in all states (l_1, x) with $x \leq 2$;

– taking c_3 immediately in all states (l_2, x);

– and delaying in all states (l_3, x) with $x < 1$ until the value of x is 1 at which point the edge c_4 is taken.

DEFINITION 6.6 (NETWORK OF TIMED GAME AUTOMATA (NTGA)).– A NTGA is a NTA G with the set of transitions E_i of each automaton $\mathcal{A}_\rangle$ partitioned into *controllable* (E_i^c) and *uncontrollable* (E_i^u) actions. We denote $E^c \overset{def}{=} \bigcup_{i\in\{1,\ldots,n\}} E_i^c$ and $E^u \overset{def}{=} \bigcup_{i\in\{1,\ldots,n\}} E_i^u$. In addition, invariants are restricted to $Inv_i : L_i \rightarrow \mathcal{C}'(X_i)$ where $\mathcal{C}'$ is the subset of $\mathcal{C}$ using constraints of the form $x \leq k$.

Given a NTGA G and a control property, the *reachability (resp. safety) control problem* consists in finding a *strategy* f for the controller such that all the runs of G supervised by f satisfy the formula. The different control properties handled by UPPAAL-TIGA are:

– *control:* $A[\ \phi\ \mathcal{U}\ \psi\ ]$, i.e., reach ψ while avoiding $\neg\phi$,

– *control:* $A <>\ \psi$, i.e., reach ψ, shortcut for $A[\ true\ \mathcal{U}\ \psi\ ]$,

– *control:* $A[\ \phi\ \mathcal{W}\ \psi\ ]$, i.e., *possibly* reach ψ while avoiding $\neg\phi$, and

– *control:* $A[]\ \phi$, i.e., avoid $\neg\phi$, shortcut for $A[\ \phi\ \mathcal{W}\ false\ ]$.

The formal definition of the control problems is based on the definitions of *strategies* and *outcomes*. In any given situation, the strategies suggest to do a particular action after a given delay. A strategy [MAL 95] is described by a function that during the course of the game constantly gives information as to what the players want to do, under the form of a pair $(e, \delta) \in (E \times \mathbb{R}_{\geq 0}) \cup \{(\perp, \infty)\}$. $(\perp, \infty)$ means that the strategy wants to delay forever.

The environment has priority when choosing its actions. In addition, it can decide not to take action, unless it is forced to do so. Uncontrollable actions can be forced to happen only in states q where an invariant requires to take action and no controllable transition is possible and there is a possible uncontrollable transition (from the location involving that invariant). For more details on the different cases where so called "forced" actions occur, we refer the reader to the manual of UPPAAL-TIGA available from *http://www.cs.aau.dk/~adavid/tiga/*. The implemented semantics is slightly different from [CAS 05] the game could be won only through controllable actions, which means that there was no "forced" action.

6.4.2. *Reachability Pipeline*

We adapt the reachability algorithm of [CAS 05] based on transition as depicted in Fig. 6.17 to an algorithm based on states for the reachability pipeline of UPPAAL-TIGA given in Fig. 6.18. Upon close look at the algorithm, it appears that we need

```
Initialisation:
    $Passed \leftarrow \{S_0\}$;
    $Waiting \leftarrow \{(S_0, \alpha, S') \mid S' = Post_\alpha(S_0)^{\nearrow}\}$;
    $Win[S_0] \leftarrow \emptyset$;
    $Depend[S_0] \leftarrow \emptyset$;

Main:
while $((Waiting \neq \emptyset) \wedge (s_0 \notin Win[S_0]))$ do
    $e = (S, \alpha, S') \leftarrow pop(Waiting)$;
    if $S' \notin Passed$ then
        $Passed \leftarrow Passed \cup \{S'\}$;
        $Depend[S'] \leftarrow \{(S, \alpha, S')\}$;
        $Win[S'] \leftarrow S' \cap G$;
        $Waiting \leftarrow Waiting \cup \{(S', \alpha, S'') \mid S'' = Post_\alpha(S')^{\nearrow}\}$;
        if $Win[S'] \neq \emptyset$ then $Waiting \leftarrow Waiting \cup \{e\}$;
    else (* reevaluate *)
        $Win^* \leftarrow Pred_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} Pred_c(Win[T]),$
                $\bigcup_{S \xrightarrow{u} T} Pred_u(T \setminus Win[T])) \cap S$;
        if $(Win[S] \subsetneq Win^*)$ then
            $Waiting \leftarrow Waiting \cup Depend[S]$; $Win[S] \leftarrow Win^*$;
        if $Win[S'] \subsetneq S'$ then $Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;
    endif
endwhile
```

Figure 6.17. *SOTFTR:* **S***ymbolic* **O***n*-**T***he*-**F***ly Algorithm for* **T***imed* **R***eachability Games*

only the destination state S' to explore forward and the source state S to explore backward. Following this remark, the waiting queue in the pipeline contains states and a direction for the exploration. The state-graph stores in addition the winning and losing subsets. Although the algorithm does not show it, we can keep track of the losing states as well. The upper part of the pipeline is computing successors similarly to the reachability pipeline of UPPAAL. The bottom part is back-propagating information (winning or losing states).

The implementation in Fig. 6.18 works as follows: When popping a state s that needs to be explored forward (source of a transition), the successors s' are computed and checked against the state-graph. Further successors will be computed (s', F) if s' is not included in the graph and s' is not winning (because then the game ends). In addition, we need to back-propagate some update to the source (s, B) if s' turned out to be winning. When popping a state s' to be explored backward, we need to go back to all sources that lead to s'. To do that we compute the $pred_t$ operation [CAS 05]

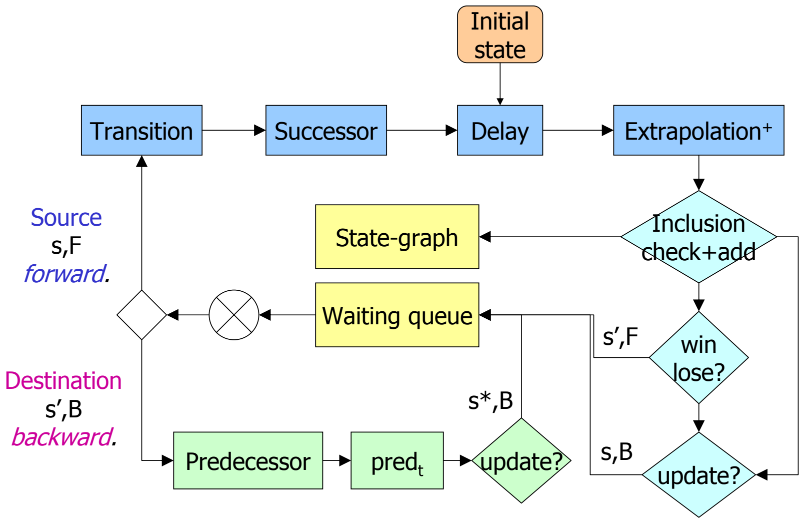


Figure 6.18. *The reachability pipeline of* UPPAAL-TIGA.

(temporal predecessors of winning states while avoiding losing states) and we back-propagate the winning subsets to the sources $(s*, B)$ for those sources that have new winning subsets.

We note that the given algorithm computes only the set of winning states but not a *strategy*. Strategies are computed on-the-fly by adding the mapping state-to-action when a given state is winning and taking this action leads to another winning state. In addition, if the winning part derives from a delay then the mapping adds the delay action instead. The only important point is to keep previously assigned states and not change their mappings when more winning states are discovered on-the-fly. This is to guarantee progress on the already known paths and avoid being caught in a loop.

6.4.3. *Time Optimality*

Time optimality for reachability games consists in computing the best (optimal) time the controller can guarantee to reach a winning state: If t^* is the optimal-time, the controller has a strategy that guarantees to reach a winning state within t^* time units whatever the opponent is doing, and moreover, the controller has no strategy to guarantee this for any $t < t^*$. This problem is solved [CAS 05] by adding a new clock z to the original TGA and the invariant $Inv(\ell) \equiv z \leq T$ for all locations ℓ where T is an upper-bound to reach the winning state. Furthermore, z is unconstrained in the initial state. The algorithm is then to compute the fix-point of all winning states and use z to deduct the optimal time. In practice, we compute iteratively the upper-bound on-the-fly to prune the state-space when a solution is found and then we continue the search to refine the current optimal. The optimal time is given in the initial state by the interval between the max of z on the zero axis and T. When pruning and updating T, the algorithm effectively converges to T being the upper-bound and the max of z being zero.

Timed optimality queries are defined by *control_t*(u,g):* $A[\ \phi\ \mathcal{U}\ \psi\ ]$, which is, only for reachability. The additional expression u defines an upper-bound to prune the search, corresponding to T in the algorithm. This upper-bound is updated on-the-fly. The expression g gives a lower bound from the current state in the search to the (goal) winning state. States that are at time $t + g > u$ are pruned. Here t is the elapsed time from the initial state. In case of doubt, it is always possible to assign u to a large value and g to zero, but meaningful values will help the search greatly.

6.4.4. *Cooperative Strategies*

In games where there is no winning strategy it may be useful to know what is the maximal set of states for which there is a winning strategy and how the environment can "help" the controller to reach such states. We call such strategies cooperative [DAV 08]. By "helping" we mean either the environment takes friendly uncontrollable actions or it lets the controller take action instead of preventing it. The result of the search is then a partition between i) states that have a winning strategy, ii) states that need cooperation of the environment, and iii) states for which there is no hope of winning.

The algorithm is using the previous reachability analysis algorithm as a component. The algorithm is shown in Fig. 6.19. The main idea is to compute the fix-point of the original model (and construct the strategy on-the-fly) and then recompute a fix-point of the modified model where we consider all transitions controllable, i.e., the environment is helping, *but very importantly*, we complete the previously obtained strategy. By completing we mean adding more states to the previous mapping but we keep the previously assigned states. These additional actions are part of the cooperative strategy, the original actions define the winning sub-strategy, and for all other states there is no hope of winning. The algorithm computes these two fix-points, although the first one may terminate early if we know that there is a winning strategy from the initial state, which we test. If the set of winning states is not reachable at all then there is no hope. In fact, there is a cooperative strategy *iff* a winning state is reachable (in the worst case the environment always cooperate).

Cooperative strategies are queried with $E <> \Phi$ where Φ is an ordinary UPPAAL-TIGA formula (including *control:*). The algorithm is generalized to safety as well where the environment needs to avoid losing states.

```
fixpoint(G,$l_0, Z_0$)
if win($l_0, Z_0$) then return true
if $\neg$reached(Win) then return false
fixpoint(G[c/u],$l_0, Z_0$)
return true
```

Figure 6.19. UPPAAL-TIGA *cooperative reachability algorithm.*

6.4.5. *Timed Games with Büchi Objectives*

In games with Büchi objectives, at least one of the goal states must be visited infinitely often. The obtained strategy guarantees this while maintaining some other (optional) safety property. The algorithm is based on the symbolic on-the-fly timed reachability algorithm of [CAS 05] (SOTFTR) with a few notable changes as follows:

– The set Win (or winning states) contains only the states that can reach $Goal$ (the set of goal states) but not $Goal$ itself (unless these states can themselves reach some goal states) contrary to the original algorithm where $Goal$ was always included in Win.

– The algorithm does not stop exploring states even if they are goal states.

– The algorithm is not on-the-fly any more in the sense that it needs to complete the search (forward and backward).

```
Initialization:
    $G = Goal$
    $Win = SOTFTR(G)$

Main:
    while $G \neq G \cap Win$ do
      $G = G \cap Win$
      $Win = SOTFTR(G)$
    done
```

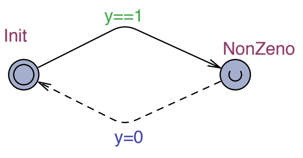
Figure 6.20. *Simplified algorithm for solving timed games with Büchi objectives.*

Figure 6.20 shows the simplified algorithm using parts of the original algorithm with the aforementioned changes. The first step is to run the original algorithm that returns the set of winning states. The input $Goal$ is the set of all goal states. The second step is to compute the fixed point of the set of goal states that are also winning, i.e., that can reach other goal states. Upon update of the set, we back-propagate again the set of goal states to compute which states are winning. We note that the call to

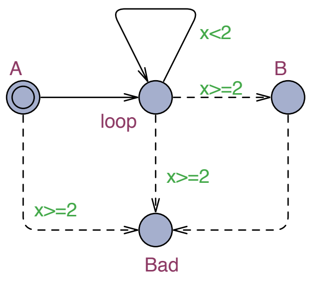
SOTFTR in the loop executes only the back-propagation part. In addition, in order to be declared winning, a goal state must reach another goal state by a discrete transition. Delaying is not enough, e.g., if $x \geq 0$ is asked with x being a clock we still need discrete transitions to be taken (this is the current semantics but it may change in the future).

The complexity of this algorithm is quadratic in the size S of the underlying untimed game (based on the region graph) of the game, which is in line with results on untimed games [CHA 06b]. It is straight-forward to extend this algorithm to be on-the-fly. In essence, we add to the main while loop condition $\neg(\forall S \in Passed, Goal[S] = Win[S] \wedge q_0 \in Win[S_0])$. Also, we stop the SOTFTR procedure with this condition. This basically means that if all the goal states we have explored so far and the initial state are winning, we can stop because they can all enforce some of these goal states we have already explored.

The syntax for these properties is of the form (i) `control: A[] ( p and A<> q )` and (ii) `control: A[] A<> q`, where `p` is the safety predicate and `q` the Büchi control objective.



(a)



(b)

Figure 6.21. *(a) Monitor automaton to avoid zeno behaviour. (b) Example exhibiting zeno behaviour.*

A major application is to generate non-zeno strategies. If we add the automaton of figure 6.4.5.(a) to some system (the guard $y == 1$ is unimportant and the constant can be tuned to the particular model), then we can ask UPPAAL-TIGA to reach the NonZeno state as a Büchi objective in addition to some other safety property we want the original system to satisfy. In the example of figure 6.4.5.(b), the control objective is to avoid the Bad state. There is a winning strategy that consists in looping without delaying in the $loop$ state. However, this strategy is zeno. If we add the automaton of figure 6.4.5.(a) and we update the query to make $NonZeno$ a repeated location, there will be no such strategy any more. In addition, if we fix the model and we add the reset `x=0` to the loop transition, the zeno strategy is still possible with a classical safety objective. With the Büchi objective however, we get a non-zeno strategy.

6.4.6. *Timed Games with Partial Observability*

UPPAAL-TIGA supports timed games with partial observability. Theoretical results on decidability are known on control of systems for event-based partial observation [LAM 00, BOU 03] and state-based partial observation [ARN 03, CHA 06a]. In this section we summarize the algorithm proposed in [CAS 07] where we consider the problem of controller synthesis for timed games under state-based partial observation. Given a timed game automaton and a finite collection of *observations* (state predicates), we compute if there exists a strategy such that a controller seeing only these observations can guarantee a safety or reachability control objective. In addition, these strategies are *stuttering invariant* in the sense that repeated identical observations will not change the strategy. The game is played as follows: Initially and whenever the observation of the system state changes, player 1 (the controller) proposes to take an action or to delay. The proposed action may be taken whenever and as long as it is enabled in the system until the observation changes. Delay means that the player is waiting for a change of observation. Then player 2 (the environment) decides the evolution of the system according the rules:

1) if player 1 chose a discrete action then player 2 can choose to play this action or another of its (enabled and uncontrollable) actions or let time pass while the action of player 1 is not enabled – as long as the observation does not change,

2) if player 1 chose to delay then player 2 can choose to play its own (enabled and uncontrollable) actions or let time pass – as long as the observation does not change,

3) player 1 can choose again what to do as soon as the observation changes.

The first rule entails that actions of the controller are urgent and that the environment has priority. Also, the controller does not know *a priori* whether his proposed action has effectively been taken or not.

The property supported are the same reachability and safety property as with perfect information but extended with observation. In practice, the query language is extended by prefixing a list of observations to the ordinary control queries. The supported queries are:

– `{ o1, o2,...} control: A[ p U q ]`: must reach `q` while maintaining `p`,

– `{ o1, o2,...} control: A[ p W q ]`: may reach `q` while maintaining `p`,

– `{ o1, o2,...} control: A<> q`: must reach `q`,

– `{ o1, o2,...} control: A[] p`: maintain `p`.

The expressions `p, q, o1, o2`...are state predicates. This extension has been used in the context of testing [DAV 09].

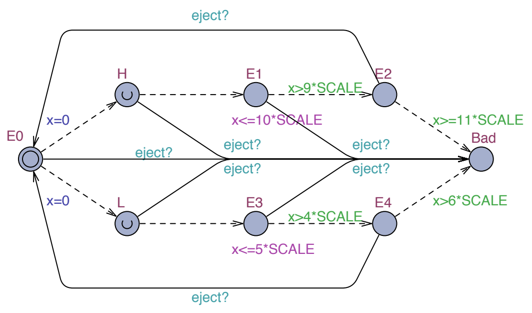


Figure 6.22. *Timed game for sorting heavy and light bricks.*

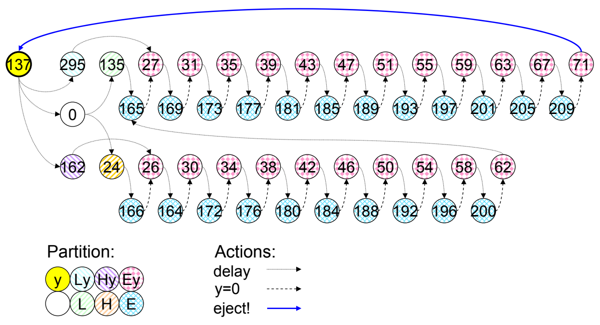


Figure 6.23. *Generated strategy to sort light and heavy boxes.*

Figure 6.22 illustrates an example of a game played with imperfect information. The goal is to sort heavy and light boxes and avoid the bad state Bad. To do so, the controller must eject the box at the right moment. The controller can reset its own clock y or synchronize on the eject channel. However, the controller can only observe the states H, L, one of $E1$, $E2$, $E3$, or $E4$ (E), Bad, and if $y \in [0, 1/2)$. In addition, if the controller ejects the box at the wrong moment then the Bad state is reached. Figure 6.4.6 shows the obtained strategy (here determinized and reduced). The generated controller is discretizing time with the only clock it can control by doing a series of reset/delay to measure time. The length of the chain depends on observing a light or heavy box. The states are partitioned in function of the observations. In 6.22 we note that the model is parameterized the variable SCALE to encode $[0, 1)$ or $[0, 1/2)$

while keeping the same query. The language limits the use of constraints to integers so we scale the timing constraints of the model instead. Interestingly, the property is not satisfied if the accuracy on y is too low (i.e. $y \in [0, 1)$) but it is if we have a finer observation (i.e. $y \in [0, 1/2)$). The query for this example is:

```
{ Box.E1 or Box.E2 or Box.E3 or Box.E4, Box.H, Box.L,
Controller.y >= 0 and Controller.y < 1 } control:  A[] not
Box.Bad
```

6.4.6.0.13. Algorithm

To solve such games with partial observation, we extend the timed game structure of UPPAAL-TIGA with two disjoint alphabets of actions σ_1 and σ_2 for the set of actions of the players 1 and 2. The algorithm given in Fig. 6.24 is similar in its main structure to the previous algorithm for solving timed games with perfect information of Fig. 6.17. It has two phases of exploring the states forward and then back-propagating winning or losing information. The main differences are that we are exploring *sets* of symbolic states (W). Furthermore, the successor states depend on the observations because the algorithm needs to compute a local fix-point at every step to find out which states (and thus construct a set of them) leave the current observation. This is what $W' = \mathsf{Next}_\alpha(W)$ is doing. The observation sets (of states) are given by $\gamma(o)$. In addition, the back-propagation is simpler than before because we do not need to use the $pred_t$ operator. Another remark is that it is possible that when computing the successors for a given action, there is a way to either deadlock or loop in the same observation. This is a *sink* for the current set of states (and action) and it is considered to be losing. We refer to [CAS 07] for a more complete description of the theory and the algorithm.

6.4.6.0.14. Implementation

The pipeline architecture is depicted in Fig. 6.25. The pipeline has different levels, the top level working on sets of (symbolic) states. Sets are explored forward or backward. The forward exploration is the most complex part since it needs to constrain the successor computation w.r.t. observations. As mentioned in the algorithm description, we have a local reachability filter (a compound filter) that has a second level where the exploration is done at the state level. The action filter on the figure has another internal level that decomposes into computing the successor (basically the chain transition - successor - delay - extrapolation) and takes care of sorting the successors in function of their actions. The observation identifier computes and check the observation to see if the states belong to the same observation or not, in particular it is detecting the winning and losing observations. We note that states are split in function of the observations and that the state-space is partitioned in function the observations.

```
Initialization:
    $Passed \leftarrow \{\{s_0\}\}$;
    $Waiting \leftarrow \{(\{s_0\}, \alpha, W') \mid \alpha \in \Sigma_1,\ o \in \mathcal{O},\ W' = \mathsf{Next}_\alpha(\{s_0\}) \cap o \wedge W' \neq \emptyset\}$;
    $Win[\{s_0\}] \leftarrow (\{s_0\} \subseteq \gamma(\mathsf{Goal})\ ?\ 1 : 0)$;
    $Losing[\{s_0\}] \leftarrow (\{s_0\} \not\subseteq \gamma(\mathsf{Goal}) \wedge (Waiting = \emptyset \vee \forall \alpha \in \Sigma_1, \mathsf{Sink}_\alpha(s_0) \neq \emptyset)\ ?\ 1 : 0)$;
    $Depend[\{s_0\}] \leftarrow \emptyset$;

Main:
while $((Waiting \neq \emptyset) \wedge Win[\{s_0\}] \neq 1 \wedge Losing[\{s_0\}] \neq 1)$ do
    $e = (W, \alpha, W') \leftarrow pop(Waiting)$;
    if $s' \notin Passed$ then
        $Passed \leftarrow Passed \cup \{W'\}$;
        $Depend[W'] \leftarrow \{(W, \alpha, W')\}$;
        $Win[W'] \leftarrow (W' \subseteq \gamma(\mathsf{Goal})\ ?\ 1 : 0)$;
        $Losing[W'] \leftarrow (W' \not\subseteq \gamma(\mathsf{Goal}) \wedge \mathsf{Sink}_\alpha(W') \neq \emptyset\ ?\ 1 : 0)$;
        if $(Losing[W'] \neq 1)$ then (* if losing it is a deadlock state *)
            $NewTrans \leftarrow \{(W', \alpha, W'') \mid \alpha \in \Sigma,\ o \in \mathcal{O},\ W' = \mathsf{Next}_\alpha(W) \cap o \wedge W' \neq \emptyset\}$;
            if $NewTrans = \emptyset \wedge Win[W'] = 0$ then $Losing[W'] \leftarrow 1$;
        if $(Win[W'] \vee Losing[W'])$ then $Waiting \leftarrow Waiting \cup \{e\}$;
        $Waiting \leftarrow Waiting \cup NewTrans$;
    else (* reevaluate *)
        $Win^* \leftarrow \bigvee_{c \in \mathsf{Enabled}(W)} \bigwedge_{W \xrightarrow{c} W''} Win[W'']$ ;
        if $Win^*$ then
            $Waiting \leftarrow Waiting \cup Depend[W]$; $Win[W] \leftarrow 1$;
        $Losing^* \leftarrow \bigwedge_{c \in \mathsf{Enabled}(W)} \bigvee_{W \xrightarrow{c} W''} Losing[W'']$ ;
        if $Losing^*$ then
            $Waiting \leftarrow Waiting \cup Depend[W]$; $Losing[W] \leftarrow 1$;
        if $(Win[W'] = 0 \wedge Losing[W'] = 0)$ then $Depend[W'] \leftarrow Depend[W'] \cup \{e\}$;
    endif
endwhile
```

Figure 6.24. *OTFPOR: **O**n-**T**he-**F**ly Algorithm for **P**artially **O**bservable **R**eachability Timed Game Structures*

6.4.7. *Simulation Checking*

UPPAAL-TIGA can be used to check weak alternating simulation relation, i.e., the simulation relation that is defined over the set of pairs of timed *game* automata (TGA) and that treats *silent* actions separately from other actions.

In this section we are checking whether there exists a simulation relation between two TGA $A = (L_A, l_{0A}, X_A, \Sigma_c \cup \{\varepsilon_c\}, \Sigma_u \cup \{\varepsilon_u\}, E_A, Inv_A)$ and $B =$

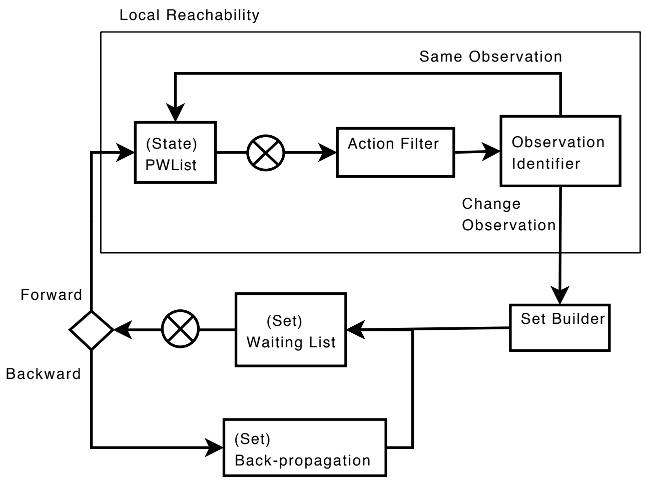


Figure 6.25. *Pipeline architecture for reachability analysis of timed games with partial observability.*

$(L_B, l_{0B}, X_B, \Sigma_c, \Sigma_u, E_B, Inv_B)$ be two TGA. Remark that we have imposed that B has any silent actions, whereas A may have controllable (ε_c) or uncontrollable ε_u silent transitions. It is a natural limitation, because abstract models usually do not have any invisible behavior. Secondly, A is not allowed to have uncontrollable silent loops, i.e., sequences of states $q_1, \ldots, q_n$ such, that $q_1 \xrightarrow{\varepsilon_u}_u q_2 \xrightarrow{\varepsilon_u}_u \ldots \xrightarrow{\varepsilon_u}_u q_n \xrightarrow{\varepsilon_u}_u q_1$. The presence of such silent loops would complicate the simulation checking algorithm.

Let us define $q \xrightarrow{\varepsilon_u *}_u \xrightarrow{a}_u q'$ iff there exists a sequence of states $q_1, \ldots, q_n$ such, that $q = q_1, q_i \xrightarrow{\varepsilon_u}_u q_{i+1}$ (for $i = 1 \ldots n-1$) and $q_n \xrightarrow{a}_u q'$. We use subscripts c and u to distinguish between controllable and uncontrollable transitions, i.e. $q \xrightarrow{a}_c q'$ is controllable and $q \xrightarrow{a}_u q'$ in uncontrollable.

DEFINITION 6.7 (TIMED WEAK ALTERNATING SIMULATION).– A *weak alternating simulation relation* between two TGA A and B is a relation $R \subseteq Q_A \times Q_B$ such that $(q_{0A}, q_{0B}) \in R$ and for every $(q_A, q_B) \in R$ and for every action $a \in \Sigma_c \cup \Sigma_u$:

– $(q_A \xrightarrow{\varepsilon_c}_c q'_A) \implies ((q'_A, q_B) \in R)$ (silent transitions)

– $(q_A \xrightarrow{a}_c q'_A) \implies \exists q'_B \quad (q_B \xrightarrow{a}_c q'_B \wedge (q'_A, q'_B) \in R)$ (controllable)

– $(q_B \xrightarrow{a}_u q'_B) \implies \exists q'_A \quad (q_A \xrightarrow{\varepsilon_u *}_u \xrightarrow{a}_u q'_A \wedge (q'_A, q'_B) \in R)$ (uncontrollable)

– $(q_A \xrightarrow{\delta} q'_A) \implies \exists q'_B \quad (q_B \xrightarrow{\delta} q'_B \wedge (q'_A, q'_B) \in R)$ (delay)

We write $A \leq B$ if there exists a weak alternating simulation relation between A and B. The intuition behind this definition is that every controllable transition that can be taken from q_A must be matched by an equally labeled controllable transition from q_B. And on the other hand, every uncontrollable transition in B tends to make B harder to control than A; then we require that it is matched by an equally labeled uncontrollable transition in A. It is also necessary to check that if the controller of A is able to avoid playing any action during a given delay, then the controller of B is able to do the same.

It can be shown that timed weak alternating simulation preserves the satisfiability of formulas of the universal fragment of TCTL (i.e. formulas of the form $A[]\ \phi$, $A <>\ \phi, \phi -- > \psi$). This means that if $A \leq B$ and A satisfies some TCTL formula that doesn't contain E path quantifier, then B also satisfies this formula. Timed weak alternating simulation can be checked between *networks* of timed automata as well.

In order to check for timed weak alternating simulation between networks of timed automata in UPPAAL-TIGA, one should use a query of the form $\{A_1, \ldots, A_m\} <= \{B_1, \ldots B_n\}$. We assume for simplicity that we are checking simulation between single automata. Consider that we are checking $\{A\} \leq \{B\}$. There are several restrictions on the automata A and B in UPPAAL-TIGA. Firstly, B is not allowed to have any silent actions. It is a natural limitation, because abstract models usually do not have any invisible behavior. Secondly, A is not allowed to have uncontrollable silent loops, i.e., sequences of states $q_1, \ldots, q_n$ such, that $q_1 \xrightarrow{\varepsilon}_u q_2 \xrightarrow{\varepsilon}_u \ldots \xrightarrow{\varepsilon}_u q_n \xrightarrow{\varepsilon}_u q_1$. The presence of such silent loops would complicate the simulation checking algorithm.

6.4.7.0.15. Algorithm

UPPAAL-TIGA uses a well-known game-theoretic approach to the simulation checking problem that reduces the simulation checking problem to solving a two-players game [ETE 01]. In this game one player, $Spoiler$, tries to put the models in inconsistent state by taking controllable transitions in A and uncontrollable in B, and the other player, $Duplicator$, tries to prevent $Spoiler$ of doing that by repeating $Spoiler$'s transition in the opposite model.

Consider the task of checking weak alternating simulation between TGAs $A = (L_A, l_{0A}, X_A, \Sigma_c \cup \{\varepsilon_c\}, \Sigma_u \cup \{\varepsilon_u\}, E_A, Inv_A)$ and $B = (L_B, l_{0B}, X_B, \Sigma_c, \Sigma_u, E_B, Inv_B) A = (L_A, l_{0A}, X_A, \Sigma_c \cup \{\varepsilon_c\}, \Sigma_u \cup \{\varepsilon_u\}, E_A, Inv_A)$ and $B = (L_B, l_{0B}, X_B, \Sigma_c, \Sigma_u, E_B, Inv_B)$. The game states of a simulation checking game are represented by tuples (l_A, l_B, Z, a), where $l_A \in L_A, l_B \in L_B, Z \subseteq R_{\geq 0}^{X_A \cup X_B}$ and $a \in \{\perp\} \cup \Sigma$. We will use functions $Z(S)$ and $Type(S)$ that return the third component of a game state S and its owner correspondingly. All the game states (l_A, l_B, Z, a) such that $a = \perp$ are the states of the player $Spoiler$ and we will use the shortcut $(l_A, l_B, Z)_S$ for identifying them. All other states belong to player $Duplicator$ and we'll use shortcut $(l_A, l_B, Z, a)_D$ for them. The initial game state is $S_0 = (l_{A0}, l_{B0}, \{\vec{0}\}^{\nearrow} \cap [\![Inv_A(l_{A0})]\!])_S$.

For two game states S_1 and S_2 we will write $S_1 \rightarrow S_2$ if there is a game transition from S_1 to S_2. The transition relation of the simulation checking game is constructed as follows:

– $(l_A, l_B, Z)_S \rightarrow (l'_A, l_B, Z')_S$ iff $e = (l_A, g, \varepsilon_c, Y, l'_A) \in E^c_A$ and $Z' = Post_e(Z)^{\nearrow} \cap [\![Inv_A(l'_A)]\!]$

– $(l_A, l_B, Z)_S \rightarrow (l'_A, l_B, Z', a)_D$ iff $e = (l_A, g, a, Y, l'_A) \in E^c_A$, $a \in \Sigma_c$ and $Z' = Post_e(Z)$

– $(l_A, l_B, Z)_S \rightarrow (l_A, l'_B, Z', a)_D$ iff $e = (l_B, g, a, Y, l'_B) \in E^u_B$ and $Z' = Post_e(Z)$

– $(l_A, l_B, Z, a)_D \rightarrow (l_A, l'_B, Z')_S$ iff $e = (l_B, g, a, Y, l'_B) \in E^c_B$, $Z' = Post_e(Z)^{\nearrow} \cap [\![Inv_B(l'_B)]\!]$

– $(l_A, l_B, Z, a)_D \rightarrow (l'_A, l_B, Z')_S$ iff $e = (l_A, g, a, Y, l'_A) \in E^u_A$, $Z' = Post_e(Z)^{\nearrow} \cap [\![Inv_A(l'_A)]\!]$

– $(l_A, l_B, Z, a)_D \rightarrow (l'_A, l_B, Z', a)_D$ iff $e = (l_A, g, \varepsilon_u, Y, l'_A) \in E^u_A$, $Z' = Post_e(Z)$

Each game state S includes a possibly infinite set of clock valuations $Z(S)$, some of them are winning for $Spoiler$. The function $Win(S) \subseteq Z(S)$ will be used to define them.

DEFINITION 6.8 .– Let us say that function Win defines the set of winning states of the player $Spoiler$, if the following requirements are fulfilled:

– if $Type(S) = Spoiler$ and $S = (l_A, l_B, Z)_S$, then
$Win(S) = Z(S) \cap \left(Z(S) \cap \left(\neg\, [\![Inv_B(l_B)]\!] \cup \bigcup_{\alpha = S \rightarrow S'} Pred_\alpha(Win(S')) \right) \right)^{\searrow}$,

– if $Type(S) = Duplicator$, then
$Win(S) = Z(S) \setminus \bigcup_{\alpha = S \rightarrow S'} Pred_\alpha(Z(S') \setminus Win(S'))$,

– Win is the least such function according to the preorder $f \leq g \equiv \forall S(f(S) \subseteq g(S))$.

The first point of this definition stands for the fact that $Spoiler$ wins in some state if the invariant of A is violated or if he can delay and move to some other winning state. The second point means that $Duplicator$ loses in some state if he can't move to some other game state, which is winning for him.

It can be proved that if Win satisfies definition 6.8, then $A \leq B$ iff $\vec{0} \notin Win(S_0)$. The winning subsets defined by Win function can be computed incrementally until the fixpoint is reached. However if we see at some point that $\vec{0} \in Win(S_0)$, then we can already build a counterexample showing that simulation is violated and thus avoid building and solving the whole game graph.

Given two models A and B and query $\{A\} <= \{B\}$ UPPAAL-TIGA explores a simulation game graph using on-the fly algorithm which is implemented using the pipeline architecture (see Fig. 6.18). Compared to the original algorithm for solving arbitrary timed games we modified the forward and backward filters (where winning conditions are defined) as well as the transition and delay filters (where the game transition relation is defined).

We exploited the fact that simulation checking game is turn-based, i.e., in each game state only one player is permitted to take a move. This allows us to simplify the algorithm to avoid using the expensive function $pred_t$ that is necessary to have in the case when both players can take a move from the same state.

6.5. TAPAAL

6.5.1. *Introduction*

TAPAAL is a platform independent tool for modelling, simulation and verification of timed-arc Petri nets. TAPAAL provides a stand-alone editor and simulator, while the verification module translates timed-arc Petri net models into networks of timed automata and uses the UPPAAL engine for the automatic analysis. The tool is available at `www.tapaal.net`.

Since the introduction of Petri nets by Carl Adam Petri [PET 62] in 1962 numerous extensions of the basic place/transition model were studied and supported by a number of academic as well as industrial tools [HEI 09]. Many recent studies on Petri net models are concerned with adding timed features that can be associated to places, transitions, arcs or tokens in the net. A recent overview aiming at a comparison of the different time dependent models (including timed automata) is given in [SRB 08].

The model considered in TAPAAL is a *Timed-Arc Petri Net* (TAPN) [BOL 90, HAN 93]. It associates an age (real number) to each token in the net and time intervals to arcs that restrict the ages of tokens that can be used for firing a transition. The advantages of this model are an intuitive semantics and a number of positive decidability results of problems like coverability and boundedness (for detailed references see [SRB 08]). On the other hand, the impossibility to describe urgent behaviours limits its modelling power and wider applicability.

TAPAAL extends the TAPN model with new features such as *invariants* for modelling of urgency and *transport arcs* for modelling systems like production lines and workflow processes. It provides an intuitive modelling environment for editing and simulating of TAPN models. The verification module of TAPAAL allows for automatic checking of bounded TAPN models against safety and liveness requirements via a translation to networks of timed automata. The UPPAAL [UPP 09] engine is then used as a back-end for the actual verification.

The connection between bounded TAPN and timed automata was studied in [SIF 96, SRB 05, BOU 08] and while theoretically satisfactory, the translations described in these papers are not suitable for a tool implementation as they either cause an exponential blow-up in the size or create a new parallel component with a fresh local clock for *each* place in the net. As UPPAAL performance becomes significantly slower with the growing number of parallel processes and clocks, the verification of larger nets with little or no concurrent behaviour (few tokens in the net) becomes intractable.

TAPAAL implements a novel translation technique where a new parallel component (with a local clock) is created for every token in the net. One of the main advantages of this approach is the possibility to use *active clock reduction* and *symmetry reduction* techniques recently implemented in UPPAAL. As a result the size of verifiable models increases by orders of magnitude as demonstrated on several examples.

6.5.2. *Definition of Timed-Arc Petri Nets Used in TAPAAL*

The set $\mathcal{I}$ of *time intervals* is defined by the following abstract syntax where a and b range over $\mathbb{N}$ and $a < b$:

$$I \ ::= \ [a,b] \ \mid \ [a,a] \ \mid \ (a,b] \ \mid \ [a,b) \ \mid (a,b) \ \mid \ [a,\infty) \ \mid \ (a,\infty) \ .$$

The set $\mathcal{I}_{Inv}$ of *invariants* is defined as the following subset of time intervals ($\mathcal{I}_{Inv} \subseteq \mathcal{I}$) where b and b' range over $\mathbb{N}$ and $b' > 0$:

$$I \ ::= \ [0,b] \ \mid \ [0,b') \ \mid \ [0,\infty) \ .$$

Let $I \in \mathcal{I}$. Given a time point $d \in \mathbb{R}^{\geq 0}$, the validity of the expression $d \in I$ is defined in the usual way, e.g., $d \in [a,b)$ iff $a \leq d < b$ and $d \in (a,\infty)$ iff $a < d$.

A *Timed-Arc Petri Net with transport arcs and place invariants* (TAPN) is a tuple $N = (P, T, F, c, F_{tarc}, c_{tarc}, \iota)$, where

– P is a finite set of *places*,

– T is a finite set of *transitions* such that $T \cap P = \emptyset$,

– $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*,

– $c : F|_{P\times T} \rightarrow \mathcal{I}$ is a function assigning a time interval to every arc from a place to a transition,

– $F_{tarc} \subseteq (P \times T \times P)$ is the set of *transport arcs* which satisfies

$$\begin{aligned} \forall (p,t,p') \in F_{tarc}, \forall r \in P.\ [&(p,t,r) \in F_{tarc} \Rightarrow p' = r \ \wedge \\ &(r,t,p') \in F_{tarc} \Rightarrow p = r \ \wedge \\ &(p,t) \notin F \ \wedge \ (t,p') \notin F], \end{aligned}$$

– $c_{tarc} : F_{tarc} \rightarrow \mathcal{I}$ is a function assigning a time interval to every transport arc, and

– $\iota : P \rightarrow \mathcal{I}_{Inv}$ is an *invariant assignment* of invariants to places.

REMARK 6.1 .– The conditions imposed on the transport arcs guarantee that for any $p \in P$ and any $t \in T$ there is at most one $p' \in P$ such that $(p, t, p') \in F_{tarc}$ and at most one $p'' \in P$ such that $(p'', t, p) \in F_{tarc}$. In other words, for any given p and t, if there is a transport arc of the form (p, t, p') or (p'', t, p) then the places p' and p'' are uniquely defined. Whenever the places p' and p'' are not relevant for the context, we shall simply denote the transport arcs as $(p, t, \_)$ or $(\_, t, p)$.

The *preset* of a transition t in the net is defined as ${}^\bullet t = \{p \in P \mid (p, t) \in F \vee (p, t, \_) \in F_{tarc}\}$, and the *postset* of a transition t is defined as $t^\bullet = \{p \in P \mid (t, p) \in F \vee (\_, t, p) \in F_{tarc}\}$.

By $\mathcal{B}(\mathbb{R}^{\geq 0})$ we denote the set of finite multisets on $\mathbb{R}^{\geq 0}$. Let $B \in \mathcal{B}(\mathbb{R}^{\geq 0})$ and $d \in \mathbb{R}^{\geq 0}$. We define $B + d$ in such a way that we add the value d to every element of B, i.e., $B + d \overset{def}{=} \{b + d \mid b \in B\}$.

Let $N = (P, T, F, c, F_{tarc}, c_{tarc}, \iota)$ be a TAPN. A *marking* M on the net N is a function $M \colon P \rightarrow \mathcal{B}(\mathbb{R}^{\geq 0})$ such that every $p \in P$ and every $x \in M(p)$ satisfies $x \in \iota(p)$. Each place is thus assigned a certain number of tokens, and each token is annotated with a real number (*age*). We moreover consider only markings such that all their tokens satisfy the place invariants imposed by the invariant assignment ι. By $|M|$ we denote the total number of tokens in the marking M, formally $|M| = \sum_{p \in P} |M(p)|$ where $|M(p)|$ is the cardinality of the multiset $M(p)$. The set of all markings on N is denoted by $\mathcal{M}(N)$.

A *marked TAPN* is a pair (N, M_0) where N is a timed-arc Petri net and M_0 is an initial marking. As *initial markings* we allow only markings with all tokens of age 0.

Let us now outline the dynamics of TAPNs. We introduce two types of transition rules: *firing* of a transition and *time delay*.

For a TAPN N we say that a transition $t \in T$ is *enabled* in a marking M if

– in all places $p \in {}^\bullet t$ there is a token x such that its age belongs to the time interval on the arc from p to t, and

– if there is a transport arc of the form (p, t, p') then moreover the age of the token in p satisfies also the invariant imposed by p'.

If a transition t is enabled then it can *fire*. This means that it consumes one token (of an appropriate age) from each place in ${}^\bullet t$, and then produces one new token to every place in $t^\bullet$. The age of the newly produced token is either 0 for the standard arcs, or it preserves the age of the consumed token in case of a transport arc.

Another behaviour of the net is a so-called *time delay* step where all tokens in the net grow simultaneously older by a given time factor (a real number in general). A time delay step is allowed only as long as all invariants in places are satisfied.

Formal definitions of transition firing and time delay steps follow.

Transition Firing

In a marking M, we can fire a transition t if it is enabled, i.e.

$$\forall p \in {}^\bullet t.\ \exists x \in M(p).\ [x \in c(p,t) \vee (x \in c_{tarc}(p,t,p') \wedge x \in \iota(p'))]\ .$$

Before firing t, we fix the sets $C_t^-(p)$ and $C_t^+(p)$ for all places $p \in P$ so that they satisfy the following equations (note that all operations are on multisets, and there may be several options for fixing these sets):

– for every $p \in P$ such that $(p,t) \in F$
$C_t^-(p) = \{x\}$ where $x \in M(p)$ and $x \in c(p,t)$,

– for every $p \in P$ such that $(t,p) \in F$
$C_t^+(p) = \{0\}$, and

– for every $p, p' \in P$ such that $(p,t,p') \in F_{tarc}$
$C_t^-(p) = \{x\} = C_t^+(p')$ where $x \in M(p)$, $x \in c_{tarc}(p,t,p')$ and $x \in \iota(p')$;

– in all other cases (when the place in the argument is unrelated to the firing of the transition t) we set the above sets to $\emptyset$.

Firing a transition t in the marking M yields a new marking M' defined as

$$\forall p \in P.\ M'(p) = \Big(M(p) \setminus C_t^-(p)\Big) \cup C_t^+(p)\ .$$

Time Delays

In a marking M we can let time pass by $d \in \mathbb{R}^{\geq 0}$ time units if

$$\forall p \in P.\ \forall x \in M(p).\ (x+d) \in \iota(p)$$

and this time delay step then yields a marking M' defined as

$$\forall p \in P.\ M'(p) = M(p) + d\ .$$

A TAPN $N = (P, T, F, c, F_{tarc}, c_{tarc}, \iota)$ generates a timed labelled transition system where states are markings of N, the set of actions is T, and the transition relation is defined by $M \xrightarrow{t} M'$ whenever the firing of a transition t in a marking M yields a marking M', and $M \xrightarrow{d} M'$ whenever a time delay of d time units in a marking M yields a marking M'.

In a marked TAPN (N, M_0) we say that a marking M is reachable iff $M_0 \longrightarrow^* M$. A marked net N is *k-bounded* for a natural number k if the total number of tokens in any of its reachable markings is less than or equal to k. A marked net is called *bounded* if it is k-bounded for some k.

6.5.3. *TAPAAL Logic*

In order to introduce TAPAAL logic formulae we have to define the set of atomic proposition $\mathcal{AP}$. Let

$$\mathcal{AP} \overset{def}{=} \{p \bowtie n \mid p \in P, n \in \mathbb{N} \text{ and } \bowtie \in \{<, \leq, =, \geq, >\}\}.$$

The interpretation is that a proposition $p \bowtie n$ is true in a marking M iff the number of tokens in the place p respects the given proposition with respect to n.

We shall now define a subset of Computation Tree Logic (CTL) used in TAPAAL (essentially mimicking the logic used in UPPAAL, except for the *leads-to* operator). The logical formulae are given by the following abstract syntax

$$\begin{aligned} \psi ::= &\ \mathsf{EF}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{AF}\,\varphi \mid \mathsf{AG}\,\varphi \\ \varphi ::= &\ p \bowtie n \mid \neg\varphi \mid \varphi \wedge \varphi \end{aligned}$$

where $p \bowtie n \in \mathcal{AP}$ and EF, EG, AF and AG are the standard CTL temporal operators.

The satisfaction relation $M \models \psi$ for a marking M and a formula ψ is defined inductively as follows:

– $M \models p \bowtie n$ iff $|M(p)| \bowtie n$,

– $M \models \neg\varphi$ iff $M \not\models \varphi$,

– $M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$,

– $M \models \mathsf{EF}\,\varphi$ iff $M \longrightarrow^* M'$ and $M' \models \varphi$

– $M \models \mathsf{EG}\,\varphi$ iff there is a (finite or infinite) alternating run ρ of the form

$$M = M_1 \xrightarrow{d_1} M_1' \xrightarrow{a_1} M_2 \xrightarrow{d_2} M_2' \xrightarrow{a_2} M_3 \xrightarrow{d_3} M_3' \xrightarrow{a_3} M_4 \xrightarrow{d_4} M_4' \xrightarrow{a_4} \ldots$$

such that for all i and for all d, $0 \leq d \leq d_i$ we have $M_i[d] \models \varphi$ (where $M_i[d]$ is the unique marking reachable from M_i by time delay of d time units) and

(i) ρ is infinite, or

(ii) ρ is finite and ends in M_k where for every $d \in \mathbb{R}^{\geq 0}$ we have $M_k \xrightarrow{d}$ and $M_k[d] \models \varphi$, or

(iii) ρ is finite and ends in a state M' (where M' is either of the form M_k or M'_k) such that whenever $M' \xrightarrow{d} M'[d]$ is possible for a $d \in \mathbb{R}^{\geq 0}$ then $M'[d] \models \varphi$ and there is no marking M'' such that $M'[d] \xrightarrow{t} M''$ for any $t \in T$,

– $M \models \mathsf{AF}\,\varphi$ iff $M \not\models \mathsf{EG}\,\neg\varphi$, and

– $M \models \mathsf{AG}\,\varphi$ iff $M \not\models \mathsf{EF}\,\neg\varphi$.

REMARK 6.2 .– The meaning of the $\mathsf{EG}\,\varphi$ formula is that there should exist a *maximal* run of the system such that at any point the formula φ is satisfied. The conditions (i), (ii) and (iii) list the three possibilities when a run is considered as maximal. It is either if (i) it consists of an infinite alternating sequence of actions and time delays (note that Zeno behaviours are not excluded), or (ii) it ends in a state where the invariants allow time to diverge, or (iii) it ends in a state from which no discrete transitions are possible after any time delay (this includes time-locks).

6.5.4. *Tool Details*

TAPAAL offers an editor, simulator and verifier for TAPN. It is written in Java 6.0 using Java Swing for the GUI components and is so available for the majority of existing platforms.

TAPAAL's graphical *editor* features all necessary elements for the creation of TAPN models, including invariants on places and transport arcs. The user interface supports, among others, a select/move feature for moving a selected sub-net of the model as well as an undo/redo buttons allowing the user to move backward and forward in the history during a creation of larger models. Constructed nets are saved in an interchangeable XML format. An important aspect of the graphical editor is that it disallows to draw syntactically incorrect nets and hence no syntax checks are necessary before calling further TAPAAL modules.

The *simulator* part of TAPAAL allows to inspect the behaviour of a TAPN by graphically simulating the effects of time delays and transition firings. When firing a transition the user can either manually select the concrete tokens that should be used for the firing or simply allow the simulator to automatically select the tokens based on some predefined strategy (the youngest, the oldest or a random token). The simulator also allows the user to step back and force in the simulated trace, which makes it easier to investigate alternative net behaviours.

TAPAAL's *verification* module allows for checking of safety and liveness queries in the constructed net. Queries are created using a novel graphical query dialog, completely eliminating the possibility of introducing syntactical errors and offering an intuitive and easy to use query formulation mechanism. The TAPAAL query language is a subset of the CTL logic comprising EF, AG, EG and AF temporal operators [1], however, several TCTL properties can be verified by encoding them into the net. The actual verification is done by the UPPAAL verification engine via translating TAPN models into networks of timed automata. The verification calls to UPPAAL are seamlessly integrated inside the TAPAAL environment and the returned error traces (if any) are displayed in the TAPAAL's simulator. For the safety questions concrete traces are displayed whenever the command-line UPPAAL engine can output them, otherwise the user is offered an untimed trace and can in the simulation mode experiment with suitable time delays in order to realize the displayed trace in the net. A number of verification/trace options found in UPPAAL are also available in TAPAAL, including a symmetry reduction option which often provides improvements of the verification times in orders of magnitude, though at the expense of disallowing trace options (a current limitation of UPPAAL). Finally, it is possible to check if the constructed net is k-bounded for any given k. If the net is not bounded, the tool provides a suitable under-approximation of the behaviour of the net (by asking for a maximum number of tokens that can be used during any transition firing sequence).

6.6. ROMÉO : A Tool for the Analysis of Timed Extensions of Petri Nets

In this paper, we present the features of ROMÉO, a tool that allows to analyse and simulate timed extensions of Petri nets that are Time Petri Nets (TPNs). The tool ROMÉO allows state space computation of TPN and on-the-fly model-checking of reachability properties. ROMÉO is a free software available for Linux, MacOSX and Windows platforms.

It can be downloaded at URL `http://romeo.rts-software.org/`.

ROMÉO analyses T-Time Petri nets, *i.e.* nets such that each transition t is associated to a time interval $[a(t), b(t)]$ (in the following, we call such nets time Petri nets). It does not only allow to compute the state space of the models but also to perform on-the-fly model-checking of quantitative temporal properties. It is able to translate time Petri nets into timed automata preserving the behavioral semantics (w.r.t. time bisimulation) of the net. The software allows to model and process an extension of time Petri nets that encompass preemption features: time Petri nets with inhibitor arcs (ITPNs). In order to model specifications that are not yet completely defined, parameters can be added to both TPN and ITPN models. ROMÉO is able to deal with such extensions and gives the possibility to check quantitative temporal properties.

1. At the moment the EG and AF queries are supported only for nets with transitions that do not contain more than two input and two output places.

The current or past contributors to the software are the following: Olivier (H.) Roux, Didier Lime, Guillaume Gardey, Morgan Magnin, Charlotte Seidner, Louis-Marie Traonouez and Gilles Bénattar.

6.6.1. *Models*

6.6.1.1. *Time Petri Nets*

Time Petri nets have been defined by Merlin [MER 74]. Time is integrated to the Petri net model by adding a timing interval associated to each transition. For a given transition, this interval specifies when it can be fired regarding the instant when the transition has been newly enabled the most recently

DEFINITION 6.9 (TIME PETRI NET).– A Time Petri net is a 7-uple $\mathcal{N} = \langle P, T, ^\bullet(.), (.)^\bullet, a, b, M_0\rangle$ where:

– $P = \{P_1, \ldots, P_m\}$ is a finite and non-empty set of *places* ;

– $T = \{t_1, \ldots, t_n\}$ is a finite and non-empty set of *transitions* ;

– $^\bullet(.) : T \to \mathbb{N}^P$ is the *backward incidence function* ;

– $(.)^\bullet : T \to \mathbb{N}^P$ is the *forward incidence function* ;

– $a : T \to \mathbb{N}$ and $b : T \to \mathbb{N} \cup \{\infty\}$ are functions giving, for each transition, its *earliest* and *latest* firing times $(a \| eqb)$;

– $M_0 \in \mathbb{N}^P$ is the *initial marking* of the net.

A *marking* M of the net is an element of $\mathbb{N}^P$ such that $\forall p \in P$, $M(p)$ is the number of *tokens* in the place p.

A transition t is said to be *enabled* by the marking M if the number of tokens in M in each input place of t is greater or equal to the value on the arc between this place and the transition (*i.e.* $M \geq^\bullet t$). We denote it by $t \in enabled(M)$.

A transition t is said to be *disabled* by the firing of t' from marking M if it is enabled by M but not by $M -^\bullet t'$. We then denote it by $t \in disabled(M, t')$.

A transition t is said to be *newly enabled* by the firing of the transition t' from the marking M if it is enabled by the new marking $M -^\bullet t' + t'^\bullet$ but was not by $M -^\bullet t'$. We denote it by $t \in \uparrow enabled\,(M, t')$ where $\uparrow enabled\,(\cdot,\cdot)$ is defined as follows:

$$\uparrow enabled\,(M, t') = \{t \in T \,|\, M -^\bullet t' + t'^\bullet \geq^\bullet t \wedge (t = t' \vee \neg(M -^\bullet t' \geq^\bullet t)\}.$$

We define the semantics of a dense-time TPN as a time transition system. In this model, two kinds of transitions may occur: *time* transitions when time elapses and *discrete* transitions when a transition of the net is fired.

DEFINITION 6.10 (SEMANTICS OF A DENSE-TIME TPN).– The semantics of a *dense-time* TPN $\mathcal{N}$ is defined as a Timed Transition system $\mathcal{S}_{\mathcal{N}}^{dense} = (Q, q_0, T, \rightarrow)$ such that:

– $Q = \mathbb{N}^P \times (\mathbb{R}^+)^T$;

– $q_0 = (M_0, \overline{0})$;

– $\rightarrow \in Q \times (\mathbb{R}^+ \cup T) \times Q$ is the transition relation including a time transition relation and a discrete transition relation :

- let $q = (M, \nu) \in Q$ and $q' = (M, \nu') \in Q$ be two states of the net, the continuous time transition relation is defined $\forall d \in \mathbb{R}^+$ by:

$$(M,\nu) \xrightarrow{d} (M,\nu') \text{ if } \forall t_i \in T, \begin{cases} \nu'(t_i) = \nu(t_i) + d \\ M \geq^{\bullet} t_i \Rightarrow \nu'(t_i) \leq b(t_i) \text{ ;} \end{cases}$$

- let $q = (M, \nu) \in Q$ and $q' = (M', \nu') \in Q$ be two states of the net, the discrete transition relation is defined $\forall t_i \in T$ by:

$$(M,\nu) \xrightarrow{t_i} (M',\nu') \text{ ssi } \begin{cases} t_i \in enabled(M) \\ M' = M -^{\bullet} t_i + t_i^{\bullet} \\ a(t_i) \leq \nu(t_i) \leq b(t_i) \\ \forall t_k \in T, \nu'(t_k) = \begin{cases} 0 \text{ if } t_k \in \uparrow enabled(M, t_i) \\ \nu(t_k) \; otherwise. \end{cases} \end{cases}$$

In the *dense-time* approach, time is seen as *"jumping" from one integer to the other*, with no care of what may happen between. The behaviors of a discrete-time model are obviously included in the behaviors of the corresponding model with a dense-time semantics. We define the *discrete-time* semantics of a time Petri net $\mathcal{N}$ by a transition system with two kinds of discrete transitions: first, a transition relation modifying the marking of the net and, second, a transition corresponding to a discrete elapsing of time (which is characterized by an increment of one time unit for all the clocks associated to transitions). We choose to write this transition system under the form of a timed transition system $\mathcal{S}_{\mathcal{N}}^{discrete} = (Q, q_0, T, \rightarrow)$: starting from the definition we previously gave for dense-time semantics, we replace the continuous time transition relation by a discrete-time transition relation:

$$(M,\nu) \xrightarrow{1} (M,\nu') \text{ ssi } \forall t_i \in T, \begin{cases} \nu'(t_i) = \nu(t_i) + 1 \\ M \geq^{\bullet} t_i \Rightarrow \nu'(t_i) \leq b(t_i). \end{cases}$$

6.6.1.2. *Petri Nets with Stopwatches*

In order to take into account the global complexity of systems, models now encompass the notion of actions that can be suspended and resumed. This implies extending traditional clock variables by "stopwatches". Several extensions of TPNs that address the modeling of stopwatches have been proposed: Scheduling-TPNs [ROU 02] , Preemptive-TPNs [BUC 04] (these two models add resources and priorities attributes to the TPN formalism) and Inhibitor Hyperarc TPNs (ITPNs) [ROU 04]. ITPNs introduce special inhibitor arcs that control the progress of transitions. These three models belong to the class of PNs extended with stopwatches (SwPNs) [BER 07].

Roméo implements ITPNs. Inhibitor hyperarcs make it easier to model systems with priority relations between transitions, but they do not increase the theoretical expressivity of the model compared to inhibitor arcs. That is why we can equivalently work on time Petri nets with inhibitor arcs or inhibitor hyperarcs. For the sake of simplicity, we focus on nets with inhibitor arcs (ITPNs) in this chapter.

DEFINITION 6.11 (TIME PETRI NETS WITH INHIBITOR ARCS).– A time Petri net with inhibitor arcs (ITPN) is a n-tuple $\mathcal{N} = (P, T, {}^\bullet(.), (.)^\bullet, {}^\circ(.), a, b, M_0)$, where

– $P = \{p_1, p_2, \dots, p_m\}$ is a non-empty finite set of *places*,

– $T = \{t_1, t_2, \dots, t_n\}$ is a non-empty finite set of *transitions*,

– ${}^\bullet(.) \in (\mathbb{N}^P)^T$ is the *backward incidence function*,

– $(.)^\bullet \in (\mathbb{N}^P)^T$ is the *forward incidence function*,

– ${}^\circ(.) \in (\mathbb{N}^P)^T$ is the *inhibition function*,

– $a : T \to \mathbb{N}$ and $b : T \to \mathbb{N} \cup \{\infty\}$ are functions giving, for each transition, its *earliest* and *latest* firing times ($a \| eqb$) ;

– $M_0 \in \mathbb{N}^P$ is the *initial marking* of the net,

A transition t is said to be *inhibited* by the marking M if the place connected to one of its inhibitor arc is marked with at least as many tokens than the weight of the considered inhibitor arc between this place and t: $0 < {}^\circ t \leq M$. We denote it by $t \in inhibited(M)$. Practically, inhibitor arcs are used to stop the elapsing of time for some transitions: an inhibitor arc between a place p and a transition t means that the stopwatch associated to t is stopped as long as place p is marked with enough tokens.

Transitions that are enabled but inhibited are said to be *suspended*.

A transition t is said to be *active* in the marking M if it is enabled and not inhibited by M.

A transition t is said to be *firable* when it has been enabled and not inhibited for at least $a(t)$ time units.

DEFINITION 6.12 (SEMANTICS OF A DENSE-TIME ITPN).– Given a time domain $\mathbb{T}$, the semantics of a dense-time ITPN $\mathcal{N}$ is defined as a Timed Transition System $\mathcal{S}_{\mathcal{N}}^{dense} = (Q, q_0, T, \to)$ such that:

– $Q = \mathbb{N}^P \times (\mathbb{R}^+)^T$;

– $q_0 = (M_0, \overline{0})$;

– $\rightarrow \in Q \times (T \cup \mathbb{R}) \times Q$ is the transition relation including a continuous time transition relation and a discrete transition relation.

- The time transition relation is defined $\forall d \in \mathbb{R}^+$ by:

$$(M,\nu) \xrightarrow{d} (M,\nu') \text{ if } \forall t_i \in T,$$
$$\left\{\begin{array}{l} \nu'(t_i) = \left\{\begin{array}{l} \nu(t_i) + d \text{ if } t_i \in enabled(M) \text{ et } t_i \in active(M) \\ \nu(t_i) \text{ otherwise,} \end{array}\right. \\ M \geq^\bullet t_i \Rightarrow \nu'(t_i) \leq b(t_i); \end{array}\right.$$

- The discrete transition relation is defined $\forall t_i \in T$ by:

$$(M,\nu) \xrightarrow{t_i} (M',\nu') \text{ if ,}$$
$$\left\{\begin{array}{l} t_i \in enabled(M) \text{ and } t_i \in active(M), \\ M' = M -^\bullet t_i + t_i^\bullet, \\ a(t_i) \leq \nu(t_i) \leq b(t_i), \\ \forall t_k \in T, \nu'(t_k) = \left\{\begin{array}{l} 0 \text{ if } t_k \in\uparrow enabled(M,t_i) \\ \nu(t_k)\ otherwise. \end{array}\right. \end{array}\right.$$

The discrete-time semantics of ITPNs results from the replacement of the continuous time transition by a discrete time transition in the the definition we previously gave for dense-time semantics:

$$(M,\nu) \xrightarrow{1} (M,\nu') \text{ if } \forall t_i \in T,$$
$$\left\{\begin{array}{l} \nu'(t_i) = \left\{\begin{array}{l} \nu(t_i) + 1 \text{ if } t_i \in enabled(M) \text{ and } t_i \in active(M) \\ \nu(t_i) \text{ otherwise,} \end{array}\right. \\ M \geq^\bullet t_i \Rightarrow \nu'(t_i) \leq b(t_i). \end{array}\right.$$

ROMÉO also implements *reset* arcs for both TPNs and ITPNs. Reset arcs allow to remove all the tokens that a place contains, making it easier to model systems with reset functions included.

6.6.1.3. *Parametric Petri Nets with Stopwatches*

These two classes of Petri nets previously presented can be extended with the use of parameters, for instance to model specifications that are not yet completely defined. In this purpose, parametric time Petri nets and parametric Petri nets with stopwatches [TRA 08] are parametric extensions of respectively time Petri nets and Petri nets with stopwatches, in which the firing intervals of the transitions can be replaced by parametric firing intervals that involve time parameters.

We present below the definition and the semantics of the parametric time Petri net with inhibitor arcs model (PITPN) that extends the ITPN model with parameters.

DEFINITION 6.13 (DEFINITION AND SEMANTICS OF PITPN).– A PITPN is a n-tuple $\mathcal{N} = \langle P, T, Par, {}^\bullet(.), (.)^\bullet, {}^\circ(.), a, b, M_0, D_p\rangle$, where:

– $Par = \{\lambda_1, \lambda_2, \ldots, \lambda_l\}$ is a finite set of *parameters*; let $\Gamma(Par)$ be the set of linear expressions over Par;

– $a : T \to \Gamma(Par)$ is the function that gives the *earliest firing time* of a transition, expressed as a linear expression over the set of parameters;

– $b : T \to \Gamma(Par) \cup \{\infty\}$ is the function that gives the *latest firing time* of a transition, that is either a linear expression over the set of parameters or equal to ∞;

– $D_p \subseteq \mathbb{N}^{Par}$ is the *domain of the parameters*;

and such that for a valuation $\nu \in D_p$, the semantics $[\![\mathcal{N}]\!]_\nu = \langle P, T, {}^\bullet(.), (.)^\bullet, {}^\circ(.), a_\nu, b_\nu, M_0\rangle$ of $\mathcal{N}$ is a non parametric ITPN such that a_ν and b_ν define the firing intervals of the transitions by replacing in a and b the parameters by their valuations ν.

6.6.2. *Global Architecture*

ROMÉO consists of a graphical user interface (GUI) (written in Tcl/Tk), a dedicated library for networks simulation and a computation module MERCUTION, written in C++. It is dedicated to the design, simulation, state space computation, model-checking and control of dense-time TPNs and their extension to stopwatches thanks to inhibitor arcs. The tool implements automatic translations of discrete-time TPNs and ITPNs into untimed Petri nets and counter automata. It also allows to symbolically compute the state space of discrete-time ITPNs.

ROMÉO offers a parametric extension for both TPNs and ITPNs. In these latter extensions, ROMÉO supports the use of parametric linear expressions in the time bounds of the transitions, and allows to add linear constraints on the parameters to restrict their domain.

We will give further details on these features in the following paragraphs.

6.6.3. *Systems Modelling*

In a system modelling activity, the ROMÉO GUI allows to model reactive systems or preemptive reactive systems using TPNs or ITPNs. Both benefit from an easy graphical representation and from an easy representation of common real-time features (parallelism, synchronization, resources management, watch-dogs, ...).

As a design helper, ROMÉO implements on-line simulation and reachability model-checking on TPNs and ITPNs . It allows the early detection of some modeling issues during the conception stage.

6.6.4. *Verification of Properties*

Once the system have been described thanks to a Petri net model (TPNs or ITPNs), a crucial step is to formalize specifications corresponding to safe behaviors. Properties then should be written thanks to observers or a dedicated timed logic.

6.6.4.1. *On-Line Model Checking*

ROMÉO provides an on-line model-checker for reachability. Properties over markings can be expressed and tested. It is then possible to test the reachability of a marking such that it verifies $M(P_1) = 1 \vee M(P_3) \geq 3$ where $M(P_i)$ is the number of tokens in the place P_i of the net. The tool returns a trace leading to such a marking if reachable.

In [GAR 05a, BOU 06] the authors went further in the model-checking of time Petri nets, by defining a specific TCTL logic for time Petri nets in dense time, called *TPN-TCTL*. The decidability of the model-checking of *TPN-TCTL* on time Petri nets is proved, and they have shown that its complexity is PSPACE.

They also have introduced a restricted subset of *TPN-TCTL* with no recursion in the formulae for which they can propose on-the-fly model-checking. Moreover, this subset appears to be sufficient to verify many interesting properties on time models. Reachability properties can be checked with formulae such as $\exists\Diamond_{[a,b]}(p)$ (where $[a, b]$ is a time interval, with b possibly infinite, and p a property on the markings of the net) and safety properties with $\forall\Box_{[a,b]}(p)$. Liveness properties can be checked with $\forall\Diamond_{[a,b]}(p)$ or by using a bounded response property such as $p \leadsto_{[0,b]} q$. It is equivalent to $\forall\Box(p \Rightarrow \forall\Diamond_{[0,b]}(q))$, and thus allows one level of recursion.

The method is extended to the state-class graph in [HAD 06] and leads to efficient model-checking algorithms for TPNs that are implemented in ROMÉO.

6.6.4.1.1. Model-Checking of a Subset of TCTL on Petri Nets with Stopwatches

In dense time, it has been shown that state and marking accessibility are not decidable on Petri nets with stopwatches, even if bounded [BER 07]. Those two problems become however decidable once a discrete time semantics is considered [MAG 06]. Then, an efficient method to compute the symbolic state-space of a Petri net with stopwatches has been proposed in [MAG 08].

The method consists in extending the classical symbolic representations of dense time (handled by convex polyhedra) to discrete time. For this purpose, a solution could be to compute the state-space of discrete time nets as the discretization of the state-space of the associated dense time models. However, altough this solution is correct for time Petri nets, it is not for Petri nets with stopwatches: indeed, in these latter this method can add wrong discrete behaviors, that is to say behaviors that are not permissible with the discrete time semantics. A solution has been proposed to overcome this problem: it consists in decomposing the polyhedra that represent the timing information of the net into an union of simpler polyhedra, that assures the validity of the computation of the symbolic successor.

It is thus possible to check real-time properties expressed by TCTL formulae on bounded Petri nets with stopwatches in discrete time, through a simple adaptation of the tool ROMÉO [GAR 05b].

Let us give an intuitive overview of the process. In [GAR 05a, BOU 06], the authors propose a method to check properties expressed in the TCTL logic (or in a subclass of TCTL logic) on time Petri nets by using the zone-based graph. This method is naturally extended to the state-class graph in [HAD 06]. Actually, its principles are general and can be applied to all time extensions of Petri nets such that the firing domains of the state-classes can be represented by DBM. Besides, the authors of [MAG 08] propose an algorithm to compute the state-space of Petri nets with stopwatches in discrete time by using only DBM. By combining the two previous procedures, an elegant method to check TCTL formulae on Petri nets with stopwatches in discrete time is obtained.

Thanks to the implementation of these algorithms in ROMÉO, the tool is able to check quantitative temporal properties on Petri nets with stopwatches with a discrete time semantics.

6.6.4.1.2. Parametric Model-checking of Petri Nets with Stopwatches

Parametric model-checking can be used to synthesize constraints on the parameters of a parametric model to assure that a property is verified. However, the parametric reachability problem is known to be undecidable in general [ALU 93].

For parametric time Petri nets and parametric Petri nets with stopwatches, semi-algorithms are proposed in [TRA 08] to verify parametric TCTL formulae (in which the bounds of the temporal constraints can be replaced by parameters). The goal is to determine the valuations of the parameters, such that for these valuations the model verifies the formula. The method consist in extending the model-checking approach for time Petri nets propose in [HAD 06], by defining the parametric state-class graph of a parametric model and on-the-fly parametric model-checking semi-algorithms of a subset of parametric TCTL.

These semi-algorithms are implemented in the tool ROMÉO [LIM 09]. As a result, it can synthesize a set of constraints (a disjunction of polyhedra encoded with the *Parma Polyhedra Library* [BAG 02]) that represents the set of solutions.

6.6.4.2. *Off-Line Model Checking*

6.6.4.2.1. Verification based on observers

Observers are a method to model check TPNs and ITPNs. It consists in adding to the Petri net - in a non-intrusive manner - places and transitions to model the property to check. The property is transformed in testing for the reachability of a given marking [TOU 97]. Then, as for the construction of the state class graph, it is possible to check properties on TPNs/ITPNs with observers.

A main advantage of this approach is the transformation of a property into a reachability or a trace execution problem. Nevertheless, observers are still a not easy way to model check TPNs and ITPNs. On the one hand, there is no automatic procedure to build observers: it is sometimes quite difficult to turn a property to check into a reachability problem with observers. On the other hand, for each property to be checked, a new state class graph has to be built and the observer can dramatically increase the size of the state space.

6.6.4.2.2. Verification based on translations into other models

An interesting alternative to check temporal quantitative propertives on time Petri nets consists in building a translation of the nets into Timed Automata (TA). There exists efficient tools working on this model and which are capable to check such properties. There are two main families of translations: on the one hand, *structural translations* (like the translation introduced in [CAS 06] that takes as input a TPN with finite or infinite latest firing times) and, on the other hand, *translations with state space computation* (see, for example, the translation of [LIM 04a, LIM 06] which consists in building the state space of a TPN as a timed automaton).

ROMÉO implements various theoretical methods allowing to translate the analyzed models into automata, timed automata or stopwatch automata (*SWA*). This method benefits from the existence of efficient *model checking* tools available on these models (MEC, ALDEBARAN, UPPAAL, KRONOS, HYTECH). These translations extend the class of properties that can be checked by the use of the observers to temporal (LTL, CTL) and quantitative temporal (TCTL) logics.

A first translation consists in the computation of the state class graphs (SCG) that provide finite representations for the behavior of bounded nets preserving their LTL properties [BER 91a]. For bounded TPNs the algorithm is based on DBM (Difference

Bounds Matrix) data structure whereas, for ITPNs, the semi-algorithm is based on polyhedra (using the *Parma Polyhedra Library* [BAG 02]).

Two different methods are implemented for TPNs to generate a TA that preserves its semantics (in the sense of *timed bisimilarity*): the first one is derived from TA framework [GAR 06], the other one from the classical state class graph approach [LIM 03]. In the latter method, we reduce the number of clocks needed during the translation, so that the subsequent verification on the resulting TA is more efficient. In both methods, the automata are generated in UPPAAL or KRONOS input format.

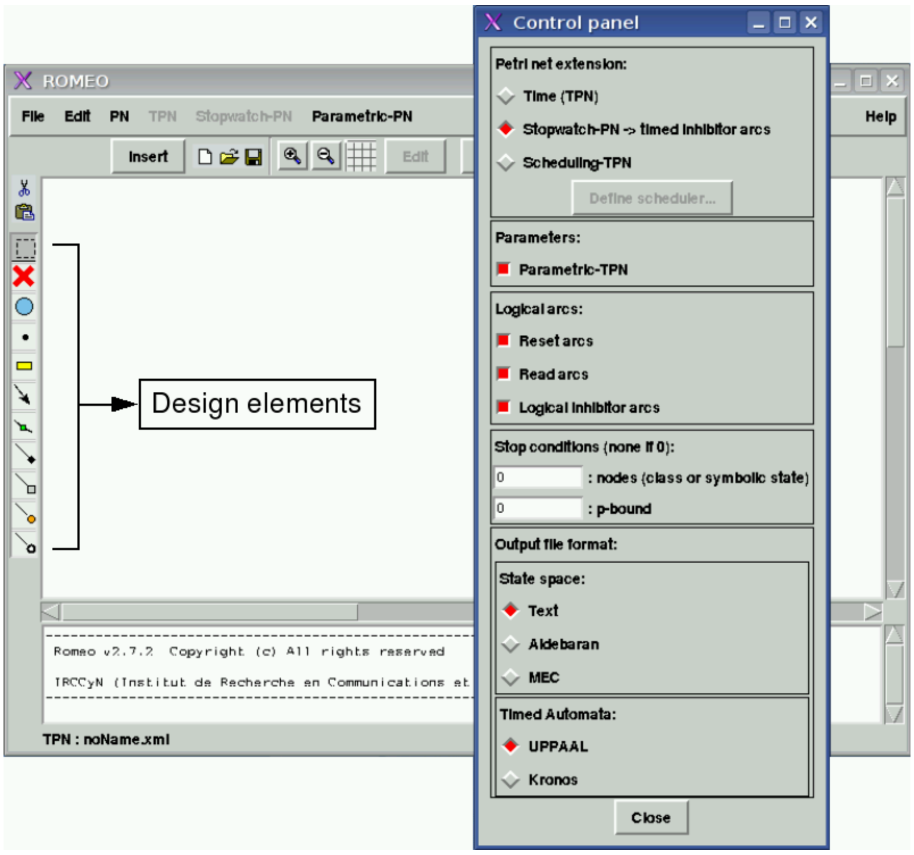
Concerning ITPNs, the approximated and exact methods introduced in [LIM 04b, MAG 05] are implemented. The first one allows a fast translation into a stopwatch automaton using an overapproximating semi-algorithm (DBM-based). Despite the overapproximation, it has been proven that the SWA is timed-bisimilar to the original ITPN. The SWA is produced in the HYTECH input format and is computed with a low number of stopwatches. Since the number of stopwatches is critical for the complexity of the verification, the method increases the efficiency of the timed analysis of the system; moreover, in some cases, it may just make the analysis possible while it would be a dead-end to model the system directly with HYTECH. The second method computes the exact state space of ITPNs. The algorithms may not terminate as the reachability problem is undecidable on dense-time ITPNs. But it may act as a (slower but still efficient) *replacement for the DBM over-approximation* in the cases when the over-approximation introduces an infinite number of markings while the net is actually bounded and prevents this method to yield results.

6.6.5. *Using* ROMÉO *in an example*

The features of ROMÉO that have been previously presented are illustrated in this section in a scheduling problem taken from [BUC 04]. We consider a system of three tasks: $task_1$ and $task_3$ are periodic, $task_2$ is sporadic. The periods are expressed in function of a time parameter a and are respectively $a, 2.a$ and $3.a$ for the tasks $1, 2$ and 3. The system has fixed priorities between the tasks: $task_1$ has the greatest priority, then $task_2$ and then $task_3$.

We design a PITPN model of this system in ROMÉO. The graphical user interface of the tool is presented in Figure 6.26. We choose in the control panel the type of net we want to edit, and we add the elements of the net (places, transitions, arcs).

We obtain the model presented in Figure 6.27, in which the inhibitors arcs, drawn with a circle end, are used to model the priorities between the tasks. Besides, we can restrict the domain of the parameter, so that $D_p = \{30 \leq a \leq 70\}$.



```
Romeo v2.7.2  Copyright (c) All rights reserved
IRCCyN (Institut de Recherche en Communications et
```

TPN : noName.xml

Petri net extension:
- Time (TPN)
- [✓ Stopwatch-PN -> timed inhibitor arcs]
- Scheduling-TPN

Define scheduler...

Parameters:
- 🟥 Parametric-TPN

Logical arcs:
- 🟥 Reset arcs
- 🟥 Read arcs
- 🟥 Logical inhibitor arcs

Stop conditions (none if 0):
- 0 : nodes (class or symbolic state)
- 0 : p-bound

Output file format:

State space:
- [✓ Text]
- Aldebaran
- MEC

Timed Automata:
- [✓ UPPAAL]
- Kronos

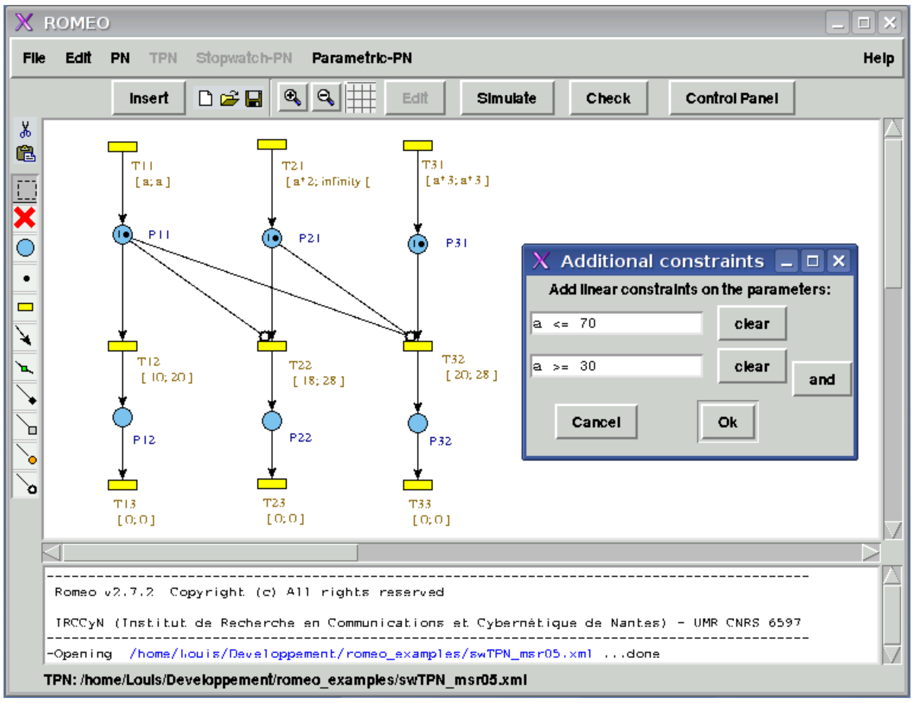
Close

Figure 6.26. *GUI of* ROMÉO *with the control panel*

The simulator of ROMÉO can be used to test scenarios for an early verification of some properties. Then, we can perform model-checking on the model. The interesting problems on this system first concern the schedulability of the three tasks, which is expressed by the property that the PITPN model is safe (i.e. $1-$bounded). We can verify this property in ROMÉO with a TCTL formula:

$$\forall P_i,\ \forall\Box_{[0,\infty[}(M(P_i) \leq 1)$$

The result of the parametric model-checking is $a > 48$, as shown in Figure 6.28.



Add linear constraints on the parameters:

a <= 70

a >= 30

```
--------------------------------------------------------------------------------
 Romeo v2.7.2  Copyright (c) All rights reserved

 IRCCyN (Institut de Recherche en Communications et Cybernétique de Nantes) - UMR CNRS 6597
--------------------------------------------------------------------------------
-Opening  /home/louis/Developpement/romeo_examples/swTPN_msr05.xml  ...done
TPN: /home/Louis/Developpement/romeo_examples/swTPN_msr05.xml
```

Figure 6.27. *PITPN model of the system with constraints on the parameters*

We can add this new constraint in ROMÉO, which assures that the system is now schedulable, and consequently we can verify new properties on the model. For example, we can compute the worst case response time (WCRT) of $task_3$ with the parametric TCTL formulae:

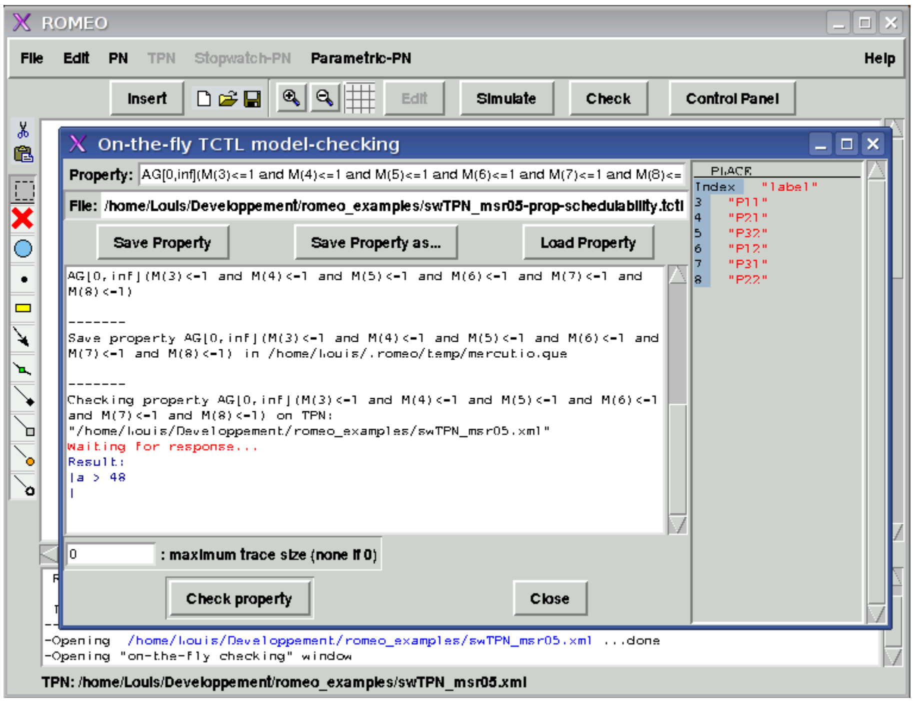
$$M(P_{31}) > 0 \rightsquigarrow_{[0,b]} M(P_{32}) > 0$$

This formula uses a new parameter b that is a maximum bound for the WCRT. The result of the parametric model-checking with ROMÉO is $b \geq 96$ and thus 96 is the WCRT of $task_3$, which is in accordance with [BUC 04] in which $a = 50$..

6.7. Bibliography

[ABA 92] ABADI M., LAMPORT L., "An Old-Fashioned Recipe for Real Time", *Proc. of REX Workshop "Real-Time: Theory in Practice"*, num. 600LNCS, p. 1–27, Springer, 1992.

[ABD 01a] ABDULLA P.A., "Using (Timed) Petri Nets for Verification of Parameterized (Timed) systems", *VEPAS'2001, Verification of Parameterized Systems, ICALP'2001 satellite workshop*, 2001.



Property: AG[0,inf](M(3)<=1 and M(4)<=1 and M(5)<=1 and M(6)<=1 and M(7)<=1 and M(8)<=

File: /home/Louis/Developpement/romeo_examples/swTPN_msr05-prop-schedulability.tctl

```
AG[0,inf](M(3)<=1 and M(4)<=1 and M(5)<=1 and M(6)<=1 and M(7)<=1 and
M(8)<=1)

-------
Save property AG[0,inf](M(3)<=1 and M(4)<=1 and M(5)<=1 and M(6)<=1 and
M(7)<=1 and M(8)<=1) in /home/louis/.romeo/temp/mercutio.que

-------
Checking property AG[0,inf](M(3)<=1 and M(4)<=1 and M(5)<=1 and M(6)<=1
and M(7)<=1 and M(8)<=1) on TPN:
"/home/louis/Developpement/romeo_examples/swTPN_msr05.xml"
Waiting for response...
Result:
|a > 48
|
```

PLACE	
Index	"label"
3	"P11"
4	"P21"
5	"P32"
6	"P12"
7	"P31"
8	"P22"

0 : maximum trace size (none if 0)

```
-Opening /home/louis/Developpement/romeo_examples/swTPN_msr05.xml ...done
-Opening "on-the-fly checking" window
```

TPN: /home/Louis/Developpement/romeo_examples/swTPN_msr05.xml

Figure 6.28. *Parametric model-checking*

[ABD 01b] ABDULLA P.A., NYLÉN A., "Timed Petri Nets and BQQs", *Proceedings of ICATPN'2001, 22nd International Conference on application and theory of Petri nets*, 2001.

[AHU 93] AHUJA R.K., MAGNANTI T.L., ORLIN J.B., *Network Flows - Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[ALU 90a] ALUR R., COURCOUBETIS C., DILL D.L., "Model-checking for Real-time Systems", *5th Symposium on Logic in Computer Science (LICS'90)*, p. 414–425, 1990.

[ALU 90b] ALUR R., DILL D.L., "Automata for Modeling Real-Time Systems", *Proc. of Int. Colloquium on Algorithms, Languages, and Programming*, vol. 443 of *LNCS*, p. 322–335, 1990.

[ALU 93] ALUR R., HENZINGER T.A., VARDI M.Y., "Parametric Real-time Reasoning", *ACM Symposium on Theory of Computing*, p. 592-601, 1993.

[ALU 94] ALUR R., DILL D.L., "A theory of timed automata", *Theoretical Computer Science*, vol. 126, num. 2, p. 183–235, 1994.

[ALU 01] ALUR R., LA TORRE S., PAPPAS G.J., "Optimal Paths in Weighted Timed Automata", *Fourth International Workshop on Hybrid Systems: Computation and Control*, vol. 2034 of *Lecture Notes in Computer Science*, p. 49–62, Springer, 2001.

[AMN 01] AMNELL T., BEHRMANN G., BENGTSSON J., D'ARGENIO P.R., DAVID A., FEHNKER A., HUNE T., JEANNET B., LARSEN K.G., MÖLLER M.O., PETTERSSON P., WEISE C., YI W., "UPPAAL - Now, Next, and Future", CASSEZ F., JARD C., ROZOY B., RYAN M. (dir.), *Modelling and Verification of Parallel Processes*, num. 2067Lecture Notes in Computer Science Tutorial, p. 100–125, Springer–Verlag, 2001.

[ARN 03] ARNOLD A., VINCENT A., WALUKIEWICZ I., "Games for Synthesis of Controllers with Partial Observation", *Theoretical Computer Science*, vol. 1, num. 303, p. 7-34, 2003.

[ASA 98] ASARIN E., MALER O., PNUELI A., SIFAKIS J., "Controller Synthesis for Timed Automata", *Proc. IFAC Symp. on System Structure & Control*, p. 469-474, Elsevier Science, 1998.

[BAG 02] BAGNARA R., RICCI E., ZAFFANELLA E., HILL P.M., "Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library", *Proceedings of the 9th International Symposium on Static Analysis*, p. 213–229, Springer-Verlag, 2002.

[BAI 08] BAIER C., KATOEN J.P., *Principles of Model Checking*, MIT Press, 2008.

[BAL 96] BALARIN F., "Approximate reachability analysis of timed automata", *17th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 1996.

[BEH 00] BEHRMANN G., HUNE T., VAANDRAGER F., "Distributed Timed Model Checking - How the Search Order Matters", *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Springer, Juli 2000.

[BEH 01a] BEHRMANN G., DAVID A., LARSEN K.G., MÖLLER M.O., PETTERSSON P., YI W., "UPPAAL - Present and Future", *Proc. of 40*th *IEEE Conference on Decision and Control*, IEEE Computer Society Press, 2001.

[BEH 01b] BEHRMANN G., FEHNKER A., HUNE T., LARSEN K.G., PETTERSSON P., ROMIJN J., "Efficient Guiding Towards Cost-Optimality in UPPAAL", MARGARIA T., YI W. (dir.), *Proceedings of the 7*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, num. 2031Lecture Notes in Computer Science, p. 174–188, Springer, 2001.

[BEH 01c] BEHRMANN G., FEHNKER A., HUNE T., LARSEN K.G., PETTERSSON P., ROMIJN J., VAANDRAGER F., "Minimum-Cost Reachability for Priced Timed Automata", BENEDETTO M.D.D., SANGIOVANNI-VINCENTELLI A. (dir.), *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, num. 2034Lecture Notes in Computer Sciences, p. 147–161, Springer, 2001.

[BEH 02] BEHRMANN G., BENGTSSON J., DAVID A., LARSEN K.G., PETTERSSON P., YI W., "UPPAAL Implementation Secrets", *Proc. of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[BEH 03a] BEHRMANN G., LARSEN K.G., PELANEK R., "To store or not to store", *Proceedings of the 15th International Conference on Computer Aided Verification*, vol. 2725 of *LNCS*, p. 433–445, Springer Verlag, 2003.

[BEH 03b] BEHRMANN G., DAVID A., LARSEN K.G., YI W., "Unification & Sharing in Timed Automata Verification", *SPIN Workshop 03*, vol. 2648 of *LNCS*, p. 225–229, 2003.

[BEH 04a] BEHRMANN G., BOUYER P., LARSEN K., PELNEK R., "Lower and upper bounds in zone based abstractions of timed automata", *TACAS 2004*, vol. 2988 of *LNCS*, p. 312–326, Springer–Verlag, 2004.

[BEH 04b] BEHRMANN G., DAVID A., LARSEN K.G., "A Tutorial on UPPAAL", BERNARDO M., CORRADINI F. (dir.), *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, num. 3185LNCS, p. 200–236, Springer–Verlag, September 2004.

[BEH 05a] BEHRMANN G., BRINKSMA E., HENDRIKS M., MADER A., "Scheduling Lacquer Production by Reachability Analysis – A Case Study", *Workshop on Parallel and Distributed Real-Time Systems 2005*, p. 140-, IEEE Computer Society, 2005.

[BEH 05b] BEHRMANN G., LARSEN K.G., RASMUSSEN J.I., "Optimal scheduling using priced timed automata", *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, num. 4, p. 34–40, ACM Press, 2005.

[BEH 07] BEHRMANN G., COUGNARD A., DAVID A., FLEURY E., LARSEN K.G., LIME D., "UPPAAL-TIGA: Time for Playing Games!", *Proceedings of the 19th International Conference on Computer Aided Verification*, num. 4590LNCS, p. 121–125, Springer, 2007.

[BEN 98] BENGTSSON J., LARSEN K.G., LARSSON F., PETTERSSON P., WANG Y., WEISE C., "New Generation of UPPAAL", *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[BEN 02] BENGTSSON J., Clocks, DBMs and States in Timed Systems, PhD thesis, Uppsala University, 2002.

[BER 91a] BERTHOMIEU B., DIAZ M., "Modeling and verification of time dependent systems using time Petri nets", *IEEE transactions on software engineering*, vol. 17, num. 3, p. 259–273, 1991.

[BER 91b] BERTHOMIEU B., DIAZ M., "Modeling and verification of time dependent systems using time Petri nets", *IEEE Trans. on Software Engineering*, vol. 17, num. 3, p. 259–273, 1991.

[BER 07] BERTHOMIEU B., LIME D., ROUX O.H., VERNADAT F., "Reachability Problems and Abstract State Spaces for Time Petri Nets with Stopwatches", *Journal of Discrete Event Dynamic Systems (DEDS)*, vol. 17, num. 2, Springer, 2007, To appear.

[BOL 90] BOLOGNESI T., LUCIDI F., TRIGILA S., "From Timed Petri Nets to Timed LOTOS", *Proceedings of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing and Verification (Ottawa 1990)*, p. 1–14, North-Holland, Amsterdam, 1990.

[BOU 03] BOUYER P., D'SOUZA D., MADHUSUDAN P., PETIT A., "Timed Control with Partial Observability", *Proc. 15^{th} Conf. on Computer Aided Verification (CAV'2003)*, vol. 2725 of *LNCS*, p. 180-192, Springer, 2003.

[BOU 06] BOUCHENEB H., GARDEY G., ROUX O.H., TCTL model checking of Time Petri Nets, Report num. number RI2006-14, IRCCyN, 2006.

[BOU 08] BOUYER P., HADDAD S., REYNIER P.A., "Timed Petri nets and timed automata: On the discriminating power of zeno sequences", *Information and Computation*, vol. 206, num. 1, p. 73-107, 2008.

[BOW 98] BOWMAN H., FACONTI G.P., KATOEN J.P., LATELLA D., MASSINK M., "Automatic Verification of a Lip Synchronisation Algorithm using UPPAAL", JAN FRISO GROOTE B.L., VAN WAMEL J. (dir.), *In Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems. Amsterdam , The Netherlands*, 1998.

[BUC 04] BUCCI G., FEDELI A., SASSOLI L., VICARIO E., "Time state space analysis of real-time preemptive systems", *IEEE transactions on software engineering*, vol. 30, num. 2, p. 97–111, February 2004.

[BYG 09] BYG J., JOERGENSEN K., SRBA J., "TAPAAL: Editor, Simulator and Verifier of Timed-Arc Petri Nets", *Submitted to ATVA'09 tool track*, 2009.

[CAS 00] CASSEZ F., LARSEN K.G., "The Impressive Power of Stopwatches", *CONCUR 2000*, vol. 1877 of *LNCS*, p. 138–152, Springer-Verlag, 2000.

[CAS 05] CASSEZ F., DAVID A., FLEURY E., LARSEN K.G., LIME D., "Efficient On-the-fly Algorithms for the Analysis of Timed Games", *CONCUR'05*, vol. 3653 of *LNCS*, p. 66–80, Springer–Verlag, August 2005.

[CAS 06] CASSEZ F., ROUX O.H., "Structural Translation from Time Petri Nets to Timed Automata – Model-Checking Time Petri Nets via Timed Automata", *The journal of Systems and Software*, vol. 79, num. 10, p. 1456-1468, Elsevier, 2006.

[CAS 07] CASSEZ F., DAVID A., LARSEN K.G., LIME D., RASKIN J.F., "Timed Control with Observation Based and Stuttering Invariant Strategies", *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, vol. 4762 of *LNCS*, p. 192–206, Springer, 2007.

[CHA 06a] CHATTERJEE K., DOYEN L., HENZINGER T., RASKIN J.F., "Algorithms for Omega-Regular games with Incomplete Information", *Computer Science Logic*, vol. 4207 of *LNCS*, p. 287–302, Springer, 2006.

[CHA 06b] CHATTERJEE K., HENZINGER T., PITERMAN N., "Algorithms for Buchi Games", *GDV 06*, August 2006.

[DAR 97] D'ARGENIO P.R., KATOEN J.P., RUYS T.C., TRETMANS J., "The bounded retransmission protocol must be on time!", *In Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1217 of *LNCS*, p. 416–431, Springer–Verlag, April 1997.

[DAV 00] DAVID A., YI W., "Modelling and Analysis of a Commercial Field Bus Protocol", *Proceedings of the 12th Euromicro Conference on Real Time Systems*, p. 165–172, IEEE Computer Society, 2000.

[DAV 02] DAVID A., BEHRMANN G., LARSEN K.G., YI W., "New UPPAAL Architecture", PETTERSSON P., YI W. (dir.), *Workshop on Real-Time Tools*, Uppsala University Technical Report Series, 2002.

[DAV 03] DAVID A., BEHRMANN G., LARSEN K.G., YI W., "A Tool Architecture for the Next Generation of UPPAAL", *10th Anniversary Colloquium. Formal Methods at the Cross Roads: From Panacea to Foundational Support*, LNCS, 2003.

[DAV 05] DAVID A., "Merging DBMs Efficiently", *17th Nordic Workshop on Programming Theory*, p. 54–56, DIKU, University of Copenhagen, October 2005.

[DAV 06] DAVID A., HÅKANSSON J., LARSEN K.G., PETTERSSON P., "Model Checking Timed Automata with Priorities using DBM Subtraction", *Proceedings of the 4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, vol. 4202 of *LNCS*, p. 128–142, 2006.

[DAV 08] DAVID A., LARSEN K.G., LI S., NIELSEN B., "Cooperative Testing of Uncontrollable Timed Systems", Fourth Workshop on Model-Based Testing MBT'08, March 2008.

[DAV 09] DAVID A., LARSEN K.G., LI S., NIELSEN B., "Timed Testing under Partial Observability", *Proceedings of the 2nd International Conference on Sofware Testing, Verification, and Validation*, IEEE Computer Society, 2009, To appear.

[DAW 06] DAWS C., KORDY P., "Symbolic Robustness Analysis of Timed Automata.", *FORMATS*, vol. 4202 of *Lecture Notes in Computer Science*, p. 143-155, Springer, 2006.

[DEA 01] DE ALFARO L., HENZINGER T.A., MAJUMDAR R., "Symbolic Algorithms for Infinite-State Games", *Proc. 12th Conf. on Concurrency Theory (CONCUR'01)*, vol. 2154 of *LNCS*, p. 536-550, Springer, 2001.

[ETE 01] ETESSAMI K., WILKE T., SCHULLER R.A., "Fair Simulation Relations, Parity Games, and State Space Reduction for BÃ¼chi Automata", *Automata, Languages and Programming*, vol. 2076 of *LNCS*, p. 694–707, Springer, 2001.

[FLO 62] FLOYD R.W., "Acm algorithm 97: Shortest Path", *Communications of the ACM*, vol. 5, num. 6, Page345, 1962.

[GAR 05a] GARDEY G., Contribution à la vérification et au contrôle des systèmes temps réel – Application aux réseaux de Petri temporels et aux automates temporisés, PhD thesis, Université de Nantes et École Centrale de Nantes, décembre 2005.

[GAR 05b] GARDEY G., LIME D., MAGNIN M., ROUX O.H., "Roméo: A tool for analyzing time Petri nets", *Proceedings of the 17th International Conference on Computer Aided Verification*, vol. 3576 of *LNCS*, p. 418-423, Springer Berlin, 2005.

[GAR 06] GARDEY G., ROUX O.H., ROUX O.F., "State Space Computation and Analysis of Time Petri Nets", *Theory and Practice of Logic Programming (TPLP). Special Issue on Specification Analysis and Verification of Reactive Systems*, vol. 6, num. 3, p. 301–320, Cambridge University Press, 2006.

[HAD 06] HADJIDJ R., BOUCHENEB H., "On-the-fly TCTL model checking for Time Petri Nets using state class graphs", *ACSD*, p. 111-122, IEEE Computer Society, 2006.

[HAN 93] HANISCH H., "Analysis of Place/Transition Nets with Timed-Arcs and its Application to Batch Process Control", *Proceedings of the 14th International Conference on Application and Theory of Petri Nets (ICATPN'93)*, vol. 691 of *LNCS*, p. 282–299, 1993.

[HAV 97] HAVELUND K., SKOU A., LARSEN K.G., LUND K., "Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL", *Proceedings of the 18th IEEE Real-Time Systems Symposium*, p. 2–13, December 1997.

[HEI 09] HEITMANN F., MOLDT D., MORTENSEN K., RÖLKE H., "Petri Nets Tools Database Quick Overview", `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/ quick.html`, Accessed: 28.4.2009.

[HEN 92] HENZINGER T.A., NICOLLIN X., SIFAKIS J., YOVINE S., "Symbolic Model Checking for Real-Time Systems", *Proc. of IEEE Symposium on Logic in Computer Science*, 1992.

[HEN 94] HENZINGER T.A., "Symbolic Model Checking for Real-time Systems", *Information and Computation*, vol. 111, p. 193–244, 1994.

[HEN 02] HENDRIKS M., LARSEN K.G., "Exact Acceleration of Real-Time Model Checking", ASARIN E., MALER O., YOVINE S. (dir.), *Electronic Notes in Theoretical Computer Science*, vol. 65, Elsevier Science Publishers, April 2002.

[HEN 03] HENDRIKS M., BEHRMANN G., LARSEN K., NIEBERT P., VAANDRAGER F., "Adding Symmetry Reduction to Uppaal", LARSEN K., NIEBERT P. (dir.), *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, vol. 2791 of *LNCS*, p. 46-49, Springer Verlag, 2003.

[HOL 91] HOLZMANN G.J., *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.

[HOL 98] HOLZMANN G.J., "An Analysis of Bitstate Hashing", *Formal Methods in System Design*, vol. 13, p. 289–307, 1998.

[HUN 00] HUNE T., LARSEN K.G., PETTERSSON P., "Guided Synthesis of Control Programs Using UPPAAL", LAI T.H. (dir.), *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, p. E15–E22, IEEE Computer Society Press, April 2000.

[IVE 00] IVERSEN T.K., KRISTOFFERSEN K.J., LARSEN K.G., LAURSEN M., MADSEN R.G., MORTENSEN S.K., PETTERSSON P., THOMASEN C.B., "Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using UPPAAL", *Proc. of 12*th *Euromicro Conference on Real-Time Systems*, p. 147–155, IEEE Computer Society Press, June 2000.

[JES 07] JESSEN J.J., RASMUSSEN J.I., LARSEN K.G., DAVID A., "Guided Controller Synthesis for Climate Controller Using UPPAAL-TIGA", *Proceedings of the 19th International Conference on Formal Modeling and Analysis of Timed Systems*, num. 4763LNCS, p. 227–240, Springer, 2007.

[KRI 96] KRISTOFFERSON K.J., LAROUSSINIE F., LARSEN K.G., PETTERSSON P., YI W., A Compositional Proof of a Real-Time Mutual Exclusion Protocol, Report num. RS-96-55, BRICS, December 1996.

[KRI 02] KRISTENSEN L., MAILUND T., "A Generalised Sweep-Line Method for Safety Properties", *Proc. of FME'02*, vol. 2391 of *LNCS*, p. 549–567, Springer-Verlag, 2002.

[LAM 00] LAMOUCHI H., THISTLE J., "Effective control synthesis for DES under partial observations", *Proceedings of the 39th IEEE Conference on Decision and Control*, p. 22–28, 2000.

[LAR 95] LARSEN K.G., PETTERSSON P., YI W., "Model-Checking for Real-Time Systems", *Proc. of Fundamentals of Computation Theory*, num. 965Lecture Notes in Computer

Science, p. 62–88, August 1995.

[LAR 97a] LARSEN K.G., PETTERSSON P., YI W., “UPPAAL in a Nutshell”, *Int. Journal on Software Tools for Technology Transfer*, vol. 1, num. 1–2, p. 134-152, Springer–Verlag, October 1997.

[LAR 97b] LARSSON F., LARSEN K.G., PETTERSSON P., YI W., “Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction”, *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, p. 14–24, IEEE Computer Society Press, December 1997.

[LAR 01] LARSEN K.G., BEHRMANN G., BRINKSMA E., FEHNKER A., HUNE T., PETTERSSON P., ROMIJN J., “As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata”, BERRY G., COMON H., FINKEL A. (dir.), *Proceedings of CAV 2001*, num. 2102Lecture Notes in Computer Science, p. 493–505, Springer, 2001.

[LIM 03] LIME D., ROUX O.H., “State class Timed Automaton of a Time Petri Net”, *The 10th International Workshop on Petri Nets and Performance Models, (PNPM’03)*, IEEE Computer Society, Sept. 2003.

[LIM 04a] LIME D., Vérification d’applications temps réel à l’aide de réseaux de Petri temporels étendus, PhD thesis, Université de Nantes et École Centrale de Nantes, décembre 2004.

[LIM 04b] LIME D., ROUX O.H., “A translation based method for the timed analysis of scheduling extended time Petri nets”, *The 25th IEEE International Real-Time Systems Symposium, (RTSS’04)*, p. 187–196, Lisbon, Portugal, IEEE Computer Society Press, December 2004.

[LIM 06] LIME D., ROUX O.H., “Model checking of time Petri nets using the state class timed automaton”, *Journal of Discrete Events Dynamic Systems - Theory and Applications (DEDS)*, vol. 16, num. 2, p. 179–205, Kluwer Academic Publishers, 2006.

[LIM 09] LIME D., ROUX O.H., SEIDNER C., TRAONOUEZ L.M., “Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches”, KOWALEWSKI S., PHILIPPOU A. (dir.), *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, vol. 5505 of *Lecture Notes in Computer Science*, p. 54-57, York, United Kingdom, Springer, March 2009.

[LIN 01] LINDAHL M., PETTERSSON P., YI W., “Formal Design and Analysis of a Gearbox Controller”, *Springer International Journal of Software Tools for Technology Transfer (STTT)*, vol. 3, num. 3, p. 353–368, 2001.

[LIU 98] LIU X., SMOLKA S., “Simple Linear-Time Algorithm for Minimal Fixed Points”, *Proc. 26th Conf. on Automata, Languages and Programming (ICALP’98)*, vol. 1443 of *LNCS*, p. 53-66, Springer, 1998.

[LON 97] LÖNN H., PETTERSSON P., “Formal Verification of a TDMA Protocol Startup Mechanism”, *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, p. 235–242, December 1997.

[MAG 05] MAGNIN M., LIME D., ROUX O., “An efficient method for computing exact state space of Petri nets with stopwatches”, *third International Workshop on Software*

Model-Checking (SoftMC'05), Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, UK, Elsevier, July 2005.

[MAG 06] MAGNIN M., MOLINARO P., ROUX O.H., "Decidability, expressivity and state-space computation of Stopwatch Petri nets with discrete-time semantics.", *8th International Workshop on Discrete Event Systems (WODES'06)*, Ann Arbor, USA, July 2006.

[MAG 08] MAGNIN M., LIME D., ROUX O., "Symbolic state space of Stopwatch Petri nets with discrete-time semantics", CORTADELLA J., REISIG W. (dir.), *The 29th International Conference on Application and Theory of Petri Nets and other models of concurrency (ICATPN 2008)*, Lecture Notes in Computer Science, Xi'an, China, Springer, June 2008.

[MAL 95] MALER O., PNUELI A., SIFAKIS J., "On the Synthesis of Discrete Controllers for Timed Systems", *Proc. 12^{th} Symp. on Theoretical Aspects of Computer Science (STACS'95)*, vol. 900, p. 229-242, Springer, 1995.

[MER 74] MERLIN P., A study of the recoverability of computing systems, PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1974.

[PET 62] PETRI C., Kommunikation mit Automaten, PhD thesis, Darmstadt, 1962.

[RAS 06] RASMUSSEN J.I., LARSEN K.G., SUBRAMANI K., "On using priced timed automata to achieve optimal scheduling", *Form. Methods Syst. Des.*, vol. 29, num. 1, p. 97–114, Kluwer Academic Publishers, 2006.

[RAZ 85] RAZOUK R.R., PHELPS C.V., "Performance analysis using timed Petri nets", *Protocol Testing, Specification, and Verification*, p. 561–576, 1985.

[ROK 93] ROKICKI T.G., Representing and Modeling Digital Circuits, PhD thesis, Stanford University, 1993.

[ROU 02] ROUX O.H., DÉPLANCHE A.M., "A T-time Petri net extension for real time-task scheduling modeling", *European Journal of Automation (JESA)*, vol. 36, num. 7, p. 973–987, 2002.

[ROU 04] ROUX O.H., LIME D., "Time Petri Nets with Inhibitor Hyperarcs. Formal Semantics and State Space Computation", CORTADELLA J., REISIG W. (dir.), *The 25th International Conference on Application and Theory of Petri Nets (ICATPN 2004)*, vol. 3099 of *Lecture Notes in Computer Science*, p. 371–390, Bologna, Italy, Springer-Verlag, June 2004.

[SIF 96] SIFAKIS J., YOVINE S., "Compositional specification of timed systems", *Proceedings of the 13th Annual Symposim on Theoretical Aspects of Computer Science (STACS'96)*, vol. 1046 of *LNCS*, p. 347–359, Springer-Verlag, 1996.

[SRB 05] SRBA J., "Timed-Arc Petri Nets vs. Networks of Timed Automata", *Proceedings of the 26th International Conference on Application and Theory of Petri Nets (ICATPN 2005)*, vol. 3536 of *LNCS*, p. 385–402, Springer-Verlag, 2005.

[SRB 08] SRBA J., "Comparing the Expressiveness of Timed Automata and Timed Extensions of Petri Nets", *Proceedings of the 6th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'08)*, vol. 5215 of *LNCS*, p. 15–32, Springer-Verlag, 2008.

[TOU 97] TOUSSAINT J., SIMONOT-LION F., THOMESSE J.P., "Time constraint verifications methods based time Petri nets", *6th Workshop on Future Trends in Distributed Computing Systems (FTDCS'97)*, p. 262–267, Tunis, Tunisia, 1997.

[TRA 08] TRAONOUEZ L.M., LIME D., ROUX O.H., "Parametric Model-Checking of Time Petri Nets with Stopwatches Using the State-Class Graph", CASSEZ F., JARD C. (dir.), *6th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2008)*, vol. 5215 of *Lecture Notes in Computer Science*, p. 280-294, Saint-Malo, France, Springer, September 2008.

[TRI 99] TRIPAKIS S., ALTISEN K., "Controller Synthesis for Discrete and Dense-Time Systems", *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, vol. 1708 of *LNCS*, p. 233-252, Springer, 1999.

[UPP 09] UPPAAL, `www.uppaal.com`, Accessed: 28.4.2009.

[WON 94] WONG-TOI H., Symbolic Approximations for Verifying Real-time Systems, PhD thesis, Stanford University, 1994.

[YI 94] YI W., PETTERSSON P., DANIELS M., "Automatic Verification of Real-Time Communicating Systems By Constraint-Solving", HOGREFE D., LEUE S. (dir.), *Proc. of the* 7th *Int. Conf. on Formal Description Techniques*, p. 223–238, North–Holland, 1994.

[ZUB 80] ZUBEREK W.M., "Timed Petri nets and preliminary performance evaluation", *Proceedings of the 7th anual symposium on Computer Architecture*, p. 88–96, ACM Press, 1980.

[ZUB 85] ZUBEREK W.M., "Extended D-timed Petri nets, timeouts, and analysis of communication protocols", *Proceedings of the 1985 ACM annual conference on the range of computing : mid-80's perspective*, p. 10–15, ACM Press, 1985.