Real Time Controller Synthesis

with

Gerd Behrmann,
Franck Cassez, Agnes Counard, Alexandre David
Emmanuel Fleury, Didier Lime

BRICS
Basic Research in Computer Science

CISS
Center for Indlejrede Software Systemer
Welcome!

UPPAAL TIGA (Fig. 1) is an extension of UPPAAL [BDL04] and it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties. Though timed games for long have been known to be decidable there has until now been a lack of efficient and truly on-the-fly algorithms for their analysis.

The algorithm we propose [CDD+05] is a symbolic extension of the on-the-fly algorithm suggested by Liu & Smolka [LS96] for linear-time model-checking of finite-state systems. Being on-the-fly, the symbolic algorithm may terminate long before having explored the entire state-space. Also the individual steps of the algorithm are carried out efficiently by the use of so-called zones as the underlying data structure. Our tool implements various optimizations of the basic symbolic algorithm, as well as methods for obtaining time-optimal winning strategies (for

Figure 1: UPPAAL TIGA on screen.

Latest News

Versions 0.10 and 0.11 released.
7 July 2007

Versions 0.10 and 0.11 are released today. Version 0.11 contains a new concrete simulator that allows the user to play strategies from the GUI. Both versions fix the following bugs: maximal constants in the formula are now taken into account, the command line simulator is new and works better, delay when no clock was used, better user feedback, end-of-game detection fixed, other bugs involving delays in the strategy, precision problems in the simulator, and leak in the DEM library. These new versions have also the following new features: options to control the type of strategy output, better control on the search ordering (forward and backward), cooperative strategies, and time optimal strategies. The manual has been updated to reflect these new features.
Real Time Model Checking

Plant
Continuous

Controller Program
Discrete

Model of environment
(user-supplied / non-determinism)

Model of tasks
(automatic?)

UPPAAL Model

sensors

actuators

SAT ??
Real Time Scheduling &
Control Synthesis

Plant
Continuous

Controller Program
Discrete

sensors

actuators

Model of
environment
(user-supplied)

Partial UPPAAL Model

Model of
evironment (user-supplied)

Synthesis of
tasks/scheduler
(automatic)

SAT $\phi$ !!
**Controller Synthesis and Timed Games**

**Production Cell**

---

**GIVEN** System moves $S$, Controller moves $C$, and property $\phi$

**FIND** strategy $s_C$ such that $s_C \parallel S \models \phi$

→ **A Two-Player Game**
Dynamic Scheduling = Controller Synthesis

Section

Reading time is uncontrollable
Untimed and Timed Games

Reachability / Safety Games

Uncontrollable

Controllable

\[ x > 1 \]
\[ x \leq 1 \]
\[ x = 0 \]

\[ x > 2 \]
\[ x < 1 \]

\[ x < 1 \]
Untimed Games

Reachability / Safety Games

Strategy:
\[ F : \text{Run}(A) \rightarrow E_c \]

Memoryless strategy:
\[ F : Q \rightarrow E_c \]

Winning Run:
\[ \text{States}(\rho) \cap G \neq \emptyset \]

Winning Strategy:
\[ \text{Runs}(F) \subseteq \text{WinRuns} \]
Untimed Games

Reachability / Safety Games

Strategy:
\[ F : \text{Run}(A) \rightarrow E_c \]

Memoryless strategy:
\[ F : Q \rightarrow E_c \]

Winning Run:
\[ \text{States}(\rho) \cap G \neq \emptyset \]

Winning Strategy:
\[ \text{Runs}(F) \subseteq \text{WinRuns} \]
Untimed Games

Reachability / Safety Games

Strategy:
\[ F : \text{Run}(A) \rightarrow E_c \]

Memoryless strategy:
\[ F : Q \rightarrow E_c \]

Winning Run:
\[ \text{States}(\rho) \cap G \neq \emptyset \]
\[ \text{States}(\rho) \cap B = \emptyset \]

Winning Strategy:
\[ \text{Runs}(F) \subseteq \text{WinRuns} \]

Loosing (memoryless) strategy

-red- Uncontrollable

-green- Controllable
Untimed Games

Reachability / Safety Games

Strategy:

\[ F : \text{Run}(A) \rightarrow E_c \]

Memoryless strategy:

\[ F : Q \rightarrow E_c \]

Winning Run:

\[ \text{States}(\rho) \cap G \neq \emptyset \]
\[ \text{States}(\rho) \cap B = \emptyset \]

Winning Strategy:

\[ \text{Runs}(F) \subseteq \text{WinRuns} \]

Winning \text{(memoryless) strategy)
Untimed Games

Backwards Fixed-Point Computation

\begin{align*}
\text{cPred}(X) &= \{ q \in Q \mid \exists q' \in X. q \rightarrow_c q' \} \\
\text{uPred}(X) &= \{ q \in Q \mid \exists q' \in X. q \rightarrow_u q' \} \\
\pi(X) &= \text{cPred}(X) \setminus \text{uPred}(X^c)
\end{align*}

Theorem:
The set of winning states is obtained as the least fixpoint of the function:

\[ X \mapsto \pi(X) \cup \text{Goal} \]
Untimed Games

Backwards Fixed-Point Computation

\[ c_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{c} q' \} \]
\[ u_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{u} q' \} \]

\[ \pi(X) = c_{\text{Pred}}(X) \setminus u_{\text{Pred}}(X^c) \]

**Theorem:**
The set of winning states is obtained as the least fixpoint of the function:

\[ X \mapsto \pi(X) \cup \text{Goal} \]
Untimed Games

Backwards Fixed-Point Computation

\[
cPred(X) = \{ q \in Q \mid \exists q' \in X. q \rightarrow_c q' \}
\]

\[
uPred(X) = \{ q \in Q \mid \exists q' \in X. q \rightarrow_u q' \}
\]

\[
\pi(X) = cPred(X) \setminus uPred(X^c)
\]

Theorem:
The set of winning states is obtained as the least fixpoint of the function:

\[
X \mapsto \pi(X) \cup \text{Goal}
\]
Untimed Games

Backwards Fixed-Point Computation

\[ c_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{c} q' \} \]
\[ u_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{u} q' \} \]
\[ \pi(X) = c_{\text{Pred}}(X) \setminus u_{\text{Pred}}(X^c) \]

**Theorem:**
The set of winning states is obtained as the least fixpoint of the function:
\[ X \mapsto \pi(X) \cup \text{Goal} \]
Untimed Games

Backwards Fixed-Point Computation

\[ c_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{c} q' \} \]

\[ u_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{u} q' \} \]

\[ \pi(X) = c_{\text{Pred}}(X) \setminus u_{\text{Pred}}(X^C) \]

**Theorem:**
The set of winning states is obtained as the least fixpoint of the function:

\[ X \mapsto \pi(X) \cup \text{Goal} \]
Uncontrollable

Controllable

Backwards Fixed-Point Computation

Untimed Games

\[ c_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{c} q' \} \]
\[ u_{\text{Pred}}(X) = \{ q \in Q \mid \exists q' \in X. q \xrightarrow{u} q' \} \]
\[ \pi(X) = c_{\text{Pred}}(X) \setminus u_{\text{Pred}}(X^c) \]

**Theorem:**
The set of winning states is obtained as the least fixpoint of the function:

\[ X \mapsto \pi(X) \cup \text{Goal} \]
**Strategy:**

- $F : \text{Run}(A) \Rightarrow E_c \cup \lambda$

**Memoryless strategy:**

- $F : Q \Rightarrow E_c \cup \lambda$

**Winning Run:**

- $\text{States}(\rho) \cap G \neq \emptyset$
- $\text{States}(\rho) \cap G = \emptyset$

**Winning Strategy:**

- $\text{Runs}(F) \subseteq \text{WinRuns}$
Strategy:
F : Run(A) → Ec ∪ λ

Memoryless strategy:
F : Q → Ec ∪ λ

Winning Run:
States(ρ) ∩ G ≠ Ø
States(ρ) ∩ G = Ø

Winning Strategy:
Runs(F) ⊆ WinRuns

Winning (memoryless) strategy)
Timed Games – State-of-the-Art

- Timed Automata + Reachability [AD94]
- Time Game Automata: Control [MPS95, AMPS98]
- Time Optimal Control (reachability) [AM99]
- “False” On-the-fly Algorithm [AT01]

- Priced Timed Automata (reachability) [LBB+01, ALTP01, LRS04, RL05]
- Price Timed Automata (safety) [BBL04]
- Price Optimal Control (reachability):
  - Acyclic PTA [LTMM02]
  - Bounded length [ABM04]
  - Strong non-zeno cost-behaviour [BCFL04]

More to come!!
Theorem: The set of winning states is obtained as the least fixpoint of the function: \( X \mapsto \pi(X) \cup \text{Goal} \)
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation

Diagram:

- States: 1, 2, 3, 4
- Edges:
  - From 1 to 2 with label $x > 1$
  - From 2 to 1 with label $x \leq 1$
  - From 2 to 3 with label $x > 2$
  - From 3 to 2 with label $x < 1$
  - From 3 to 4 with label $x \leq 1$
  - From 4 to 3 with label $x := 0$

- Initial state:
  - State 1

- Winning regions:
  - State 1
  - State 3

- Time intervals:
  - 0 to 1
  - 1 to 2
  - 2 to 3
  - 3 to 4

- Equations:
  - $x := 0$
  - $x > 1$
  - $x \leq 1$
  - $x > 2$
  - $x < 1$
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation

Diagram showing states and transitions with conditions: x > 1, x ≤ 1, x ≥ 2, x < 1, x := 0.
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation

Diagram showing states and transitions with conditions like $x > 1$, $x \leq 1$, etc.
Timed Games – State-of-the-Art

Backwards Fixed-Point Computation

We want Forward and On-The-Fly Algorithm in order to avoid constructing all (backwards) reachable state-space and to allow for discrete variables (e.g. in UPPAAL)
On-the-fly Algorithms for Timed Games

- \( S \) and \( S' \) are symbolic states, i.e., sets of concrete states.
- \( G \) is the set of (concrete) goal states.
- \( E = \{ S \xrightarrow{\alpha} S', S \xrightarrow{\mu} S' \} \)
  the (finite) set of symbolic transitions (concrete).
- \( \text{Waiting} \subseteq E \)
  is the list of symbolic transitions waiting for \( \text{Passed} \).
- \( \text{Passed} \) is the list of the passed symbolic states.
- \( \text{Win} \subseteq S \)
  is the subset of \( S \) currently known to be reachable.
- \( \text{Depend} \subseteq E \)
  indicates the edges (predecessors) of \( S \) whose information about \( S \) is obtained.

### Initialization:
- \( \text{Passed} \leftarrow \{ S_0 \} \) where \( S_0 = \{(\ell_0, 0)\} \)
- \( \text{Waiting} \leftarrow \{(S_0, \alpha, S') \mid S' = \text{Post}_\alpha(S_0) \} \)
- \( \text{Win}[S_0] \leftarrow S_0 \cap (\{ \text{Goal} \} \times \mathbb{R}_{\geq 0}^X) \)
- \( \text{Depend}[S_0] \leftarrow \emptyset \)

### Main:
```plaintext
while (\( \text{Waiting} \neq \emptyset \) \( \land \) \( s_0 \notin \text{Win}[S_0] \)) do
  \( e = (S, \alpha, S') \leftarrow \text{pop}(\text{Waiting}) \);
  if \( S' \notin \text{Passed} \) then
    \( \text{Passed} \leftarrow \text{Passed} \cup \{ S' \} \);
    \( \text{Depend}[S'] \leftarrow \{ (S, \alpha, S') \} \);
    \( \text{Win}[S'] \leftarrow S' \cap (\{ \text{Goal} \} \times \mathbb{R}_{\geq 0}^X) \);
    \( \text{Waiting} \leftarrow \text{Waiting} \cup \{ (S', \alpha, S'') \mid S'' = \text{Post}_\alpha(S') \} \);
  else (* reevaluate *)
    \( \text{Win}^* \leftarrow \text{Pred}_t(\text{Win}[S] \cup \bigcup_{S \xrightarrow{\mu} T} \text{Pred}_c(\text{Win}[T]), \)
    \( \bigcup_{S \xrightarrow{\mu} T} \text{Pred}_u(T \setminus \text{Win}[T]) \) \( \cap S \);
    if \( \text{Win}[S] \not\subseteq \text{Win}^* \) then
      \( \text{Waiting} \leftarrow \text{Waiting} \cup \text{Depend}[S] ; \text{Win}[S] \leftarrow \text{Win}^* ; \)
      \( \text{Depend}[S'] \leftarrow \text{Depend}[S'] \cup \{ e \} ; \)
  endif
endwhile
```
On-the-fly Algorithms for Timed Games

- $S, S'$ are symbolic states, i.e. sets of concrete states.
- $G$ is the set of (concrete) goal states.
- $E = \{ S \xrightarrow{a} S', S \xrightarrow{a} S' \}$ the (finite) set of symbolic transitions (considering guards).
- $\text{Waiting} \subseteq E$ is the list of symbolic transitions waiting to be considered.
- $\text{Passed}$ is the list of the passed symbolic states.
- $\text{Win}[S] \subseteq S$ is the subset of $S$ currently in the winning set.
- $\text{Depend}[S]$ indicates which states are currently in the winning set.

**Initialization:**

- $\text{Passed} \leftarrow \{ S_0 \}$ where $S_0 = \{(\ell_0, 0)\}$
- $\text{Waiting} \leftarrow \{(S_0, \alpha, S') \mid S' = \text{Post}_\alpha(S_0)\}$
- $\text{Win}[S_0] \leftarrow S_0 \cap (\{\text{Goal}\} \times \mathbb{R}_{\geq 0})$
- $\text{Depend}[S_0] \leftarrow \emptyset$

**Main:**

\begin{algorithmic}
\Function{On-the-fly algorithm for timed games [CONCUR'05]}{\text{UPPAAL Tiga}}
\State $\text{Passed} \leftarrow \emptyset$
\State $\text{Waiting} \leftarrow \emptyset$
\While{$\text{Waiting} \neq \emptyset$}
\If{$\exists \alpha$ such that $(S, \alpha, S') \in \text{Waiting}$}
\State $\text{Passed} \leftarrow \text{Passed} \cup \{(S, \alpha, S') \mid S'' = \text{Post}_\alpha(S')\}$
\Else (* reevaluate *)
\State $\text{Win}^* \leftarrow \text{Pred}_T(\text{Win}[S] \cup \bigcup_{S \xrightarrow{a} T} \text{Pred}_T(\text{Win}[T])$
\EndIf
\State $\text{Win}[S] \leftarrow \text{Win}[S] \cup \text{Win}^*$
\State $\text{Depend}[S] \leftarrow \text{Depend}[S] \cup \{e\}$
\EndWhile
\EndFunction
\end{algorithmic}
UPPAAL Tiga: New Concrete Time Simulator
UPPAAL Tiga : CTL Control Objectives

- **Reachability properties:**
  - control: \( A[p U q] \) until
  - control: \( A<> q \Leftrightarrow \) control: \( A[true U q] \)

- **Safety properties:**
  - control: \( A[p W q] \) weak until
  - control: \( A[]p \Leftrightarrow \) control: \( A[p W false] \)

- **Time-optimality :**
  - control_t*(u,g): \( A[p U q] \)
    - \( u \) is an upper-bound to prune the search, act like an invariant but on the path = expression on the current state.
    - \( g \) is the time to the goal from the current state (a lower-bound in fact), also used to prune the search. States with are pruned
A Buggy Brick Sorting Program

Exercise:

Design Controller so that only yellow boxes are being pushed out.
Brick Sorting

Controller

Generic Plate

Piston
Brick Sorting

Strategy for EJECT

Controller

Generic Plate

Piston

- pos >= 9
- pos <= 8
- on1
- pos <= 9
- sensor
- pos <= 1
- ok?
- turn == ID
- pos = 0, turn++

- start

- eject!
- red?
- blck?

- s1
- eject?
- y = 0

- s2
- y <= 1
- y >= 1

- remove!
Balancing Plates / Timed Automata

E\[\neg(\text{Plate1.Bang or Plate2.Bang or ...})\]
Balancing Plates / Time Uncertainty

Strategy

BDD/CDD
Production Cell
## Experimental Results

<table>
<thead>
<tr>
<th>Plates</th>
<th>Basic</th>
<th>Basic + inc</th>
<th>Basic + inc + pruning</th>
<th>Basic+lose + inc + pruning</th>
<th>Basic+lose + inc + topt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>mem</td>
<td>time</td>
<td>mem</td>
<td>time</td>
</tr>
<tr>
<td>2</td>
<td>win</td>
<td>0.0s</td>
<td>1M</td>
<td>0.0s</td>
<td>1M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>0.0s</td>
<td>1M</td>
<td>0.0s</td>
<td>1M</td>
</tr>
<tr>
<td>3</td>
<td>win</td>
<td>0.5s</td>
<td>19M</td>
<td>0.0s</td>
<td>1M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>1.1s</td>
<td>45M</td>
<td>0.1s</td>
<td>1M</td>
</tr>
<tr>
<td>4</td>
<td>win</td>
<td>33.9s</td>
<td>1395M</td>
<td>0.2s</td>
<td>8M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>-</td>
<td>-</td>
<td>0.5s</td>
<td>11M</td>
</tr>
<tr>
<td>5</td>
<td>win</td>
<td>-</td>
<td>-</td>
<td>3.0s</td>
<td>31M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>-</td>
<td>-</td>
<td>11.1s</td>
<td>61M</td>
</tr>
<tr>
<td>6</td>
<td>win</td>
<td>-</td>
<td>-</td>
<td>89.1s</td>
<td>179M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>-</td>
<td>-</td>
<td>699s</td>
<td>480M</td>
</tr>
<tr>
<td>7</td>
<td>win</td>
<td>-</td>
<td>-</td>
<td>3256s</td>
<td>1183M</td>
</tr>
<tr>
<td></td>
<td>lose</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**New Experimental Results**
**Using UPPAAL 4.0 architecture**

<table>
<thead>
<tr>
<th>Model</th>
<th>3</th>
<th>6</th>
<th>12</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>c</td>
<td>0.1s</td>
<td>1M</td>
<td>12s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>u</td>
<td>0.2s</td>
<td>3M</td>
<td>235s</td>
<td>-</td>
</tr>
<tr>
<td>New</td>
<td>c</td>
<td>0.05s</td>
<td>3.5M</td>
<td>0.05s</td>
<td>0.14s</td>
</tr>
<tr>
<td></td>
<td>u</td>
<td>0.02s</td>
<td>3.5M</td>
<td>0.04s</td>
<td>0.12s</td>
</tr>
</tbody>
</table>

**Tricks (Alexandre):**
- UPPAAL pipeline architecture, which implies
  * active clock reduction
  * PW-list
  * UPPAAL optimizations (successor computation, postponed evaluation, reduced copies..)
  * improved DBM library
  * improved copy-on-write implementations
  * improved subtraction (vital)
  * enormously improved merge (between DBMs) (vital)
Climate Control

Syvsten, Northern Jutland, DK

With Jan J. Jessen
Jacob I. Rasmussen
Climate Control

Zone i-1

get

have_{i-1}

give

Zone i

inlet

Heater

Zone i+1

have_{i+1}

give

want_{i+1}

get
Climate Control / Neighbor

Neighboring zone

Temperature in neighbor zone (lower/higher)

temp[id]? temp[id] = false : temp[id] = true, check_hotness_integrity()

Neighbor wants to receive flow?

c : choice_t
state_changed!

x <= 0

n[id] = c
Climate Control / Controller

bool flow_balance(const choice_t n0, const choice_t n1, bool in, bool out)
{
    bool o = out || (n[0] == WANT && n0 == HAVE) || (n[1] == WANT && n1 == HAVE);
    bool i = in || (n[0] == HAVE && n0 == WANT) || (n[1] == HAVE && n1 == WANT);
    return o == i;
}

state_changed?

Init

c0 : choice_t,
c1 : choice_t,
heat : intbool_t,
in : intbool_t,
out : intbool_t

flow_balance(c0,c1,in,out)
c[0] = c0,
c[1] = c1,
heater = heat,
inlet = in,
outlet = out,
temp_derivative = compute_temperature(c0,c1,in,out,heat)
```c
int compute_temperature(const choice_t c0, const choice_t c1, const intbool_t in, const intbool_t out,
{
    int o, i, amp;
    //active out-flow
    o = out + (c0 == WANT && n[0] == WANT) + (c1 == WANT && n[1] == WANT);
    //active in-flow
    i = in + (c0 == WANT && n[0] == HAVE) + (c1 == WANT && n[1] == HAVE);
    i = i >> 1;
    //Multiplier per incoming flow
    amp = (o * PER_OUT_CONTRIBUTION) / i;
    if (objective) //heating
    {
        return heat
        + amp*(c0 == WANT && n[0] == HAVE) ? (temp[0] ? (!hottest ? 3 : 1) : -1) : 0)
        + amp*(in ? -3 : 0)
        + -(c0 == HAVE) //Motivation for participation, even when not neighbor doesn’t want, og has air
        + -(c1 == HAVE); //Motivation for participation, even when not neighbor doesn’t want, og has air
    }
    else //cooling
    {
        return (heat ? -3 : 0)
        + amp*(in ? 5 : 0)
        + amp*(c0 == WANT && n[0] == HAVE) ? (!temp[0] ? (hottest ? 3 : 1) : -1) : 0)
        + (c0 == HAVE) //Motivation for participation, even when not neighbor doesn’t want, og has air
        + (c1 == HAVE); //Motivation for participation, even when not neighbor doesn’t want, og has air
    }
}
```
Obtaining executable code

---

Strategy for state:
Zone i-1: (Temp. lower/equal, wants flow)
Zone i+1: (Temp. lower/equal, no interaction)
Hottest neighbor: i-1
Objective: heat
is:
Wants flow from i-1
Wants flow from i+1
inlet closed
outlet off
heater on
---

Strategy for state:
Zone i-1: (Temp. greater, offers flow)
Zone i+1: (Temp. greater, offers flow)
Hottest neighbor: i+1
Objective: cool
is:
Has flow for i-1
Has flow for i+1
inlet open
outlet on
heater off
---

Strategy for state:
Zone i-1: (Temp. lower/equal, no interaction)
Zone i+1: (Temp. greater, no interaction)
Hottest neighbor: i+1
Objective: cool

control : A[]
((ZC.Init && objective) imply temp_derivative > 0) &&
((ZC.Init && !objective) imply temp_derivative < 0)

---

BDD 289 nodes

1296 cases
Obtaining executable code
More Applications - we need you!
Efficient Algorithms for Optimal Infinite Scheduling
Multipriced Timed Automata
Priced Timed Games
- Optimal strategies undecidable in general [Raskin ao]
- Decidability in setting of 1 clock or strong non-zenoness.
Timed Games with Imperfect information.
Distributed and parallel implementations (PC clusters, GRID, Shared Memory Machines)
Reading material

- **UPPAAL Cora (Priced Timed Automata):**

- **UPPAAL Tiga (Timed Games & Controller Synthesis):**