Controller Synthesis with UPPAAL-TIGA

Alexandre David
Kim G. Larsen, Didier Lime,
Franck Cassez, Jean-François Raskin...
Overview

- Timed Games.
  - Algorithm (CONCUR’05).
  - Strategies.
  - Code generation.
  - Architecture of UPPAAL-TIGA.
- Timed Games with Partial Observability.
  - Algorithm (ATVA’07).
Controller Synthesis/TGA

Given
- System moves $S$,
- Controller moves $C$,
- and a property $\varphi$,

find
- a strategy $S_c$ s.t. $S_c \parallel S \models \varphi$,
- or prove there is no such strategy.
Timed Game Automata

- Introduced by Maler, Pnueli, Sifakis [Maler & al. ’95].
- The controller continuously observes the system (all delays & moves are observable).
- The controller can
  - wait (delay action),
  - take a controllable move, or
  - prevent delay by taking a controllable move.
Timed Game Automata

- Timed automata with controllable and uncontrollable transitions.
- Reachability & safety games.
  - control: $A<> TGA\.goal$
  - control: $A[\] not TGA\.L4$
- Memoryless strategy:
  - state $\rightarrow$ action.
TGA – Let’s Play!

- control: $A<> \text{TGA.goal}$

  - $x<1$ : $\lambda$
  - $x=1$ : $c$

  - $x<2$ : $\lambda$
  - $x\geq 2$ : $c$

Strategy

- $x\leq 1$ : $c$

- $x<1$ : $\lambda$
- $x=1$ : $c$

Note: This is one strategy. There are other solutions.
Results

- [Maler & al. ’95, De Alfaro & al. ’01] There is a **symbolic iterative algorithm** to compute the set $W^*$ of winning states for timed games.

- [Henziger & Kopke ’99] **Safety** and **reachability** control are **EXPTIME-complete**.

Algorithm

- On-the-fly forward algorithm with a backward fix-point computation of the winning/losing sets.
  - Use all the features of UPPAAL in forward.
  - Possible to mix forward & backward exploration.
- Solved by Liu & Smolka 1998 for untimed games.
- Extended symbolic version at CONCUR’05.
Initialization:
\[
\text{Passed} \leftarrow \{S_0\} \quad \text{where} \quad S_0 = \{(\ell_0, \tilde{0})\};
\]
\[
\text{Waiting} \leftarrow \{(S_0, \alpha, S') \mid S' = \text{Post}_\alpha(S_0)\}
\]
\[
\text{Win}[S_0] \leftarrow S_0 \cap (\{\text{Goal}\} \times \mathbb{R}^X_{\geq 0});
\]
\[
\text{Depend}[S_0] \leftarrow \emptyset;
\]

Main:
\\
while \((\text{Waiting} \neq \emptyset) \land (s_0 \not\in \text{Win}[S_0])\) do
\\
e = (S, \alpha, S') \leftarrow \text{pop}(\text{Waiting});
\\
if \(S' \not\in \text{Passed}\) then
\\
\text{Passed} \leftarrow \text{Passed} \cup \{S'\};
\]
\[
\text{Depend}[S'] \leftarrow \{(S, \alpha, S')\};
\]
\[
\text{Win}[S'] \leftarrow S' \cap (\{\text{Goal}\} \times \mathbb{R}^X_{\geq 0});
\]
\[
\text{Waiting} \leftarrow \text{Waiting} \cup \{(S', \alpha, S'') \mid S'' = \text{Post}_\alpha(S')\};
\]
\\
if \(\text{Win}[S'] \neq \emptyset\) then \text{Waiting} \leftarrow \text{Waiting} \cup \{e\};
\\
else (** reevaluate **)\(^a\)
\\
\text{Win}^* \leftarrow \text{Pred}_t(\text{Win}[S] \cup \bigcup_{S \prec T} \text{Pred}_c(\text{Win}[T]),
\]
\[
\bigcup_{S \to T} \text{Pred}_u(\text{T} \setminus \text{Win}[T])) \cap S;
\]
\\
if \((\text{Win}[S] \subset \text{Win}^*)\) 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
\\
18-11-2010
Backward Propagation

L0
L1
L2
L3

Note: This is not a strategy, it’s only the set of winning states.

Back-propagate when goal is reached.

L0
L1
L2
L3
L4

Note: This is not a strategy, it’s only the set of winning states.
Backward Propagation

Predecessors of G avoiding B?

pred_t
\[ \text{pred}_t \left( \bigcup_{i} G_i, \bigcup_{j} B_j \right) = \bigcup_{i} \bigcap_{j} \text{pred}_t (G_i, B_j) \]

\[ \text{pred}_t (G, B) = (G^\downarrow \setminus B^\downarrow) \bigcup ((G \bigcap B^\downarrow)^{\downarrow} \setminus B)^{\downarrow} \]
Query Language (1)

- Reachability properties:
  - control: $A[p \mathcal{U} q]$
  - control: $A<> q \iff control: A[true \mathcal{U} q]$

- Safety properties:
  - control: $A[p \mathcal{W} q]$
  - control: $A[]p \iff control: A[p \mathcal{W} false]$

- Tuning:
  - change search ordering,
  - add back-propagation of winning+losing states.
Query Language (2)

- **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 t+g > u are pruned.

- **Cooperative strategies.**
  - E<> control: φ
    - Property satisfied iff φ is reachable but the obtained strategy is maximal.
Cooperative Strategies

- State-space is partitioned between states from which there is a strategy and those from which there is no strategy.
- Cooperative strategy suggests moves from the opponent that would “help” the controller.
- Being used in testing.
Strategies?

- The algorithm computes sets of winning and losing states, *not* strategies.
- Strategies are computed on top:
  - Take actions that lead to winning states (reachability).
  - Take actions to avoid losing states (safety).
  - **Partition** states with actions to guarantee progress.
  - This is done on-the-fly and the obtained strategy depends on the exploration order.
Winning States → Strategy

Also possible

Winning states

Strategy
Strategies as Partitions

- Built on-the-fly.
- Guarantee progress in the strategy.
  - No loop.
- Deterministic strategy.
- Different problem than computing the set of winning states.
- Different ordering searches can give different strategies ... with possibly the same set of winning states.
Code Generation

- Mapping state $\rightarrow$ action.
  - $\#$ entries = $\#$ states.
- Decision graph state $\rightarrow$ action.
  - $\#$ tests = $\#$ variables.
  - More compact.
  - Based on a hybrid BDD/CDD with multi-terminals.
Decision Graph

- BDD: boolean variables.
- CDD: constraints on clocks.
- Multi-terminals: actions.
  - It works because we have a partition.
Graph Reduction

- Testing consecutive bits:
  - Replace by one testing with a mask.
  - Can span on several variables.
Decision Graph
Pipeline Architecture

Pipeline Components

- Source
- Sink
- Buffer
- Filter

Data
- State
- Successor
Pipeline Architecture

Transition → Successor → Delay → Extrapolation +

Source $s,F$ forward.

State-graph

Destination $s',B$ backward.

Predecessor → $\text{pred}_t$ → update?

Waiting queue

Inclusion check + add

$s',F$

$s,B$

$\text{win}, \text{lose}?$

update?
Query Language (4)

- Partial observable systems:
  - \{ obs1, obs2 ... \} control: reachability | safety
  - Generate a controller based on partial observations of the system (obs1, obs2...).

- Games with Buchi accepting states.
  - control: A[] ( p && A<> q )
  - control: A[] A<> q
Query Language(5)

- Simulation checking of TA & TGA.
  - \{ A, B \ldots \} \leq \{ C, D \ldots \}
  - Built-in new algorithm that bypasses a manual encoding we had [LSV tech. report KGL, AD, TC].
  - Strategies & simulation in the GUI available.
  - FORMATS’09
TGA with Buchi Accepting States

- Use TIGA’s algorithm as an intermediate steps in a fixpoint, but modified:
  - winning states are states that can reach goals
  - goal states defined by queries, not necessarily winning
- Fixpoint:
  \[ \text{Win} = \text{SOTF}(\text{goal}) \]
  \[ \text{while } (\text{Win} \cap \text{goal}) \neq \text{goal} \]
  \[ \text{goal} = \text{Win} \cap \text{goal} \]
  \[ \text{Win} = \text{SOTF}(\text{goal}) \]
  \[ \text{done} \]
  \[ \text{return } S_0 \in \text{Win} \]
Application: Non-zeno Strategies

- Add a monitor with the rest of the system.
- Ask control: $A[] (p \land \land A<>\text{Monitor.Check})$

![Diagram](image-url)
Timed Games

*with Partial Observability*

- Previous: Perfect information.
  - Not always suitable for controllers.

- Partial observation.
  - States or events, here states.
  - Distinguish states w.r.t. observations.
  - Strategy keeps track of states w.r.t. observations.
  - Observations = predicates over states.
Results

- **Discrete event systems**
  - [Kupferman & Vardi ‘99, Reif ‘84, Arnold & al. ‘03]. Game given as modal logic formula: Full-observation as hard as partial observation.
  - [Chatterjee & al. ’06, De Wulf & al. ’06]. Game given as explicit graph: Full-observation \textsc{PTIME}, partial observation \textsc{EXPTIME}.

- **Timed systems, game given as a TA**
  - [Cassez & al. ’07] Efficient on-the-fly algorithm, \textsc{EXPTIME}.
State Based Full Observation

- 2-player reachability game, controllable + uncontrollable actions.
- Full observation: in $l_2$ do $c_1$, in $l_3$ do $c_2$. 
Partition the state-space \( \ell_2 = \ell_3 \).

Can’t win here.
State Based Partial Observation

Winning Strategy:
after: play \( c_1 \)
after: play \( c_1 \)
after: play \( c_2 \)

The controller can observe each state’s change
Observation For Timed Systems

In Continuous Timed Systems, “next state” is reached by:

- either a discrete step
- or a continuous time-step

Possible Observations:
- each 1/2 t.u.: \( \ldots \)
- each 1/4 t.u.: \( \ldots \)
- as it wishes: \( \ldots \)

\( x = 1; c_1 \)

2000 times within 1 t.u.

- the controller cannot observe each state’s change

Issue: When does the controller observe the system?
Stuttering-Free Invariant Observations

Assumption: the controller can only see changes of observations

Stuttering-free observation:

Must play based on stuttering-free observations
TGA with PO: Rules

- If player 1 wants to take an action $c$, then
  - player 2 can choose to play any of his actions or $c$ as long as the observation stays the same, or
  - player 2 can delay as long as $c$ is not enabled and the observation stays the same.
  - $\Rightarrow$ $c$ is urgent.

- If player 1 wants to delay, then
  - player 2 can delay or take any of his actions as long as the observation stays the same.

- The turn is back to player 1 as soon as the observation changes.
On-the-Fly Algorithm

Initialization:
\[ \text{Passed} \leftarrow \{s_0\}; \]
\[ \text{Waiting} \leftarrow \{(s_0, \alpha, W') \mid \alpha \in \Sigma_1, \ o \in \mathcal{O}, \ W' = \text{Next}_\alpha(s_0) \cap o \wedge W' \neq \emptyset\}; \]
\[ \text{Win}[[s_0]] \leftarrow \{s_0\} \subseteq \gamma(\text{Goal}) \land 1 : 0; \]
\[ \text{Losing}[[s_0]] \leftarrow \{s_0\} \subseteq \gamma(\text{Goal}) \land (\text{Waiting} = 0 \lor \forall \alpha \in \Sigma_1, \text{Sink}_\alpha(s_0) \neq \emptyset) \land 1 : 0; \]
\[ \text{Depend}[[s_0]] \leftarrow \emptyset; \]

Main:
while \((\text{Waiting} \neq \emptyset) \land \text{Win}[[s_0]] \neq 1 \land \text{Losing}[[s_0]] \neq 1\) do
\[ e = (W, \alpha, W') \leftarrow \text{pop} \text{(Waiting)}; \]
if \(s' \notin \text{Passed}\) then
\[ \text{Passed} \leftarrow \text{Passed} \cup \{W'\}; \]
\[ \text{Depend}[W'] \leftarrow \{(W, \alpha, W')\}; \]
\[ \text{Win}[W'] \leftarrow \{W' \subseteq \gamma(\text{Goal}) \land 1 : 0; \}
\[ \text{Losing}[W'] \leftarrow \{W' \subseteq \gamma(\text{Goal}) \land \text{Sink}_\alpha(W') \neq \emptyset \land 1 : 0; \}
if \((\text{Losing}[W'] = 1)\) then \((\ast \text{ if losing it is a deadlock state } \ast)\)
\[ \text{NewTrans} \leftarrow \{(W', \alpha, W'') \mid \alpha \in \Sigma, \ o \in \mathcal{O}, \ W' = \text{Next}_\alpha(W) \land o \wedge W' \neq \emptyset\}; \]
if \(\text{NewTrans} = \emptyset \land \text{Win}[W'] = 0\) then \(\text{Losing}[W'] = 1; \)
if \((\text{Win}[W'] \lor \text{Losing}[W'])\) then \(\text{Waiting} \leftarrow \text{Waiting} \cup \{e\}; \)
\[ \text{Waiting} \leftarrow \text{Waiting} \cup \text{NewTrans}; \]
else \((\ast \text{ reevaluate } \ast)\)
\[ \text{Win}^{*} \leftarrow \bigvee_{c \in \text{Enabled}(W)} \bigwedge_{W'' \subseteq W'''} \text{Win}[W''']; \]
if \(\text{Win}^{*}\) then
\[ \text{Waiting} \leftarrow \text{Waiting} \cup \text{Depend}[W]; \text{Win}[W] \leftarrow 1; \]
\[ \text{Losing}^{*} \leftarrow \bigwedge_{c \in \text{Enabled}(W)} \bigvee_{W'' \subseteq W''' \cap \text{Win}[W'']} \text{Losing}[W'']; \]
if \(\text{Losing}^{*}\) then
\[ \text{Waiting} \leftarrow \text{Waiting} \cup \text{Depend}[W]; \text{Losing}[W] \leftarrow 1; \]
if \(\text{Win}[W'] = 0 \land \text{Losing}[W'] = 0\) then \(\text{Depend}[W'] \leftarrow \text{Depend}[W'] \cup \{e\}; \)
endwhile
Algorithm

Partition the state-space w.r.t. observations. Observations O1 O2 O3. Winning/losing is observable.

\[ \lnot O_1 \land \lnot O_2 \land O_3 \]
\[ \lnot O_1 \land O_2 \land \lnot O_3 \]
\[ O_1 \land O_2 \land \lnot O_3 \]
\[ O_1 \land \lnot O_2 \land O_3 \]
\[ \lnot O_1 \land O_2 \land O_3 \]

change of observation
Algorithm with Reachability Objective

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

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

Passed ← \{\{s_0\}\};
Waiting ← \\{((s_0), \alpha, W') \mid \alpha \in \Sigma_1, \sigma \in \mathcal{O}, \ W' = \text{Next}_\alpha(\{s_0\}) \cap \sigma \land W' \neq \emptyset\};
Win[\{s_0\}] ← (\{s_0\} \subseteq \gamma(\text{Goal}) ? 1 : 0);
Losing[\{s_0\}] ← (\{s_0\} \not\subseteq \gamma(\text{Goal}) \land (\text{Waiting} = \emptyset \lor \forall \alpha \in \Sigma_1, \text{Sink}_\alpha(s_0) \neq \emptyset) ? 1 : 0);
Depend[\{s_0\}] ← \emptyset;

- Passed list \{ sets of W \}.
- Waiting list \{ tuples (W,\alpha,W') \}.
- Win[W] maps to 1 (winning) or 0 (unknown).
- Similar Losing[W].
- Depend[W] records the graph.
- Sink_\alpha: sink states by doing \alpha.
Sets of Symbolic States & Successors

Next_\alpha(W): sets of symbolic successors for some action \alpha.

Set W of symbolic states within one observation.

Next_\alpha(W) \cap O_2

Next_\alpha(W) \cap O_3

Next_\alpha: do \{\alpha asap and wait for \alpha\} until change of observation.
Forward Phase

\[
\begin{align*}
\text{Passed} &\leftarrow \text{Passed} \cup \{W'\}; \\
\text{Depend}[W'] &\leftarrow \{(W, \alpha, W')\}; \\
\text{Win}[W'] &\leftarrow (W' \subseteq \gamma(\text{Goal}) ? 1 : 0); \\
\text{Losing}[W'] &\leftarrow (W' \not\subseteq \gamma(\text{Goal}) \land \text{Sink}_\alpha(W') \neq \emptyset ? 1 : 0); \\
\text{if } (\text{Losing}[W'] \neq 1) \text{ then } (* \text{ if losing it is a deadlock state } *) \\
\text{NewTrans} &\leftarrow \{(W', \alpha, W'') \mid \alpha \in \Sigma, o \in \mathcal{O}, W' = \text{Next}_\alpha(W) \cap o \land W'' \neq \emptyset\}; \\
\text{if } \text{NewTrans} = \emptyset \land \text{Win}[W'] = 0 \text{ then } \text{Losing}[W'] \leftarrow 1; \\
\text{if } (\text{Win}[W'] \lor \text{Losing}[W']) \text{ then } \text{Waiting} &\leftarrow \text{Waiting} \cup \{e\}; \\
\text{Waiting} &\leftarrow \text{Waiting} \cup \text{NewTrans}; \\
\end{align*}
\]

Update Win, Losing, graph.

Continue forward.

Back-propagate if necessary.

Detect deadlocks.

Partition successors.
Backward Phase

If there is a $c$ whose successors are all winning

$$Win^* \leftarrow \bigvee_{c \in \text{Enabled}(W)} \bigwedge_{W \xrightarrow{c} W''} Win[W''];$$

if $Win^*$ then

$$Waiting \leftarrow Waiting \cup \text{Depend}[W]; Win[W] \leftarrow 1;$$

then $W$ is winning and back-propagate.

If every $c$ has a losing successor

$$Losing^* \leftarrow \bigwedge_{c \in \text{Enabled}(W)} \bigvee_{W \xrightarrow{c} W''} Losing[W''];$$

if $Losing^*$ then

$$Waiting \leftarrow Waiting \cup \text{Depend}[W]; Losing[W] \leftarrow 1;$$

then $W$ is losing and back-propagate.

Update graph in case of new paths to passed states.

$$\text{if } (Win[W'] = 0 \land Losing[W'] = 0) \text{ then } \text{Depend}[W'] \leftarrow \text{Depend}[W'] \cup \{e\};$$
Algorithm

Initial state in some partition.
Compute successors \{ set of states \} w.r.t. a controllable action.
Successors distinguished by observations.
Algorithm

Construct the graph of sets of symbolic states. Back-propagate winning/losing states.
Notes

- Stuttering-free invariant observations
  - sinks possible in observations (deadlock/livelock/loop)
- Actions are urgent
  - delay until actions are enabled (or observation changes)
Algorithm

- Forward exploration
  - constrained by action + observations
  - delay special
- Back-propagation.
  - If all successors\(^a\) are winning, declare current state winning, strategy: take action \(a\).
  - If one successor\(^a\) is losing, avoid action \(a\).
  - If no action is winning the current state is losing.
Example

Observations: L, H, E, B, y in [0,1[
Example – Memoryful Strategy

Partition:
- \( y \), \( Ly \), \( Hy \), \( Ey \)

Actions:
- Delay
- \( y = 0 \)
- Eject!

Controllable with \( y \in [0, \frac{1}{2}] \),
not with \( y \in [0,1] \).
Case-Studies


Automatic Controller Synthesis

Getting from models to production code

Case study:
Climate controller for livestock production
Real-Life Pig Stable in Northern Jutland
The Scenario

Uppaal TIGA model + query

Strategy

Parser

S-function

Controller

C-code

MATLAB/Simulink + Real Time Workshop
Step 1: Generating a Strategy

- Uppaal TIGA model + query
- Strategy
- Parser
- S-function
- Controller
- C-code
- MATLAB/Simulink + Real Time Workshop
Step 1: Generating the Strategy

Uppaal TIGA models and Control Query

Control Query:

control : A[] ZC.Decided imply forall (c0 : choice_t)\nforall (c1 : choice_t)\nforall (in : intbool_t)\nforall (out : intbool_t)\nforall (heat : intbool_t) (flow_balance(c0,c1,in,out) imply\nobj_func >= compute_objective_function(c0,c1,in,out,heat))
Step 1: Generating the Strategy

Uppaal TIGA models and Control Query

Control Query:

control : A[] ZC.Decided imply forall (c0 : choice_t) \forall (c1 : choice_t) \forall (in : intbool_t) \forall (out : intbool_t) \forall (heat : intbool_t) (flow_balance(c0,c1,in,out) imply obj_func >= compute_objective_function(c0,c1,in,out,heat))

Strategy to avoid losing:

State: ( Neighbor(0)._id0 Neighbor(1)._id0 ZC.Decide StateChanger._id4 ) n[0]=2 n[1]=2 temp[0]=1 temp[1]=0 humid[0]=1 humid[1]=0 objective=1 hottest=0
morehumid=0 decrease_humidity=0 c[0]=0 c[1]=0 inlet=0 heater=0 outlet=0 obj_f
unc=0

When you are in true, take transition ZC.Decide->ZC.Decided { flow_balance(1, 0, 0, 1), tau, c[0] = 1, c[1] = 0, heater := 1, inlet := 1, obj_func := compute_objective_function(1, 0, 0, 1) }

State: ( Neighbor(0)._id0 Neighbor(1)._id0 ZC.Init StateChanger._id4 ) n[0]=1 n[1]=1 temp[0]=1 humid[0]=0 humid[1]=0 objective=0 hottest=1 morehumid=1 decrease_humidity=1 c[0]=0 c[1]=0 inlet=0 heater=0 outlet=0 obj_f

c=0

...
Step 2: From Strategy to S-function

- **Uppaal TIGA model + query**

- **Strategy** → **Parser** → **S-function**

- **Controller**

- **C-code**

- **MATLAB/Simulink + Real Time Workshop**
Step 2: From Strategy to S-function

Uppaal TIGA Strategy

Ruby Script

Simulink S-function

{
    INLET = OFF;
    OUTLET = ON;
    HEATER = ON;
    GiveLeft = 0;
    GetLeft = 1;
    GiveRight = 0;
    GetRight = 0;
}
{
    INLET = ON;
    OUTLET = ON;
    HEATER = OFF;
}
...
Step 3: From S-function to Production Code

- Uppaal TIGA model + query
- Controller
- Strategy
- Parser
- S-function
- C-code
- MATLAB/Simulink + Real Time Workshop
Step 3: From S-function to Production Code

Actual dynamics modelled in Simulink

Validation through simulations

Embedded S-function
Step 3: From S-function to Production Code

Actual dynamics modelled in Simulink

Production Code

dstatic void gamecontrol2_output(int T tid)
{
    SimStruct *rts = gamecontrol2_M->childSfunctions[4];
    sfcnOutputs(rts, 0);
}
SimStruct *rts = gamecontrol2_M->childSfunctions[5];
    sfcnOutputs(rts, 0);
}
SimStruct *rts = gamecontrol2_M->childSfunctions[6];
    sfcnOutputs(rts, 0);
}

int32_T i1;
for(i1 = 0; i1 < 18; i1++) {
    gamecontrol2_B.Sum[i1] =
    gamecontrol2_B.SFunction1[i1] -
    gamecontrol2_P.Constant_Value;
    gamecontrol2_B.Gain[i1] =
    gamecontrol2_B.Sum[i1] *
    gamecontrol2_P.Gain_Gain;
}

....
Step 4: Executing the Production Code

- Uppaal TIGA model + query
- Strategy
- Parser
- S-function
- Controller
- C-code
- MATLAB/Simulink + Real Time Workshop
Generalizing this Work

- The case-study
  - ad-hoc translation
  - ad-hoc script
  - ad-hoc interface

- What we did:
  - Defined the interface (inputs & outputs in Simulink).
  - Rewrote the Ruby translator
    - generic inputs/outputs
    - generic strategies
    - discretizes time.
  - Integrated the workflow inside Simulink.
Workflow Between TIGA & Simulink
Results

- From a TIGA model and a Simulink model we can
  - generate the S-function that acts as the discrete controller inside Simulink,
  - simulate the model and validate the controller.