Timed Games
UPPAAL-TIGA

Alexandre David

1.2.05

Overview

- Timed Games.
  - Algorithm (CONCUR’05).
  - Strategies.
  - Code generation.
  - Architecture of UPPAAL-TIGA.
  - Interactive game.
- Timed Games with Partial Observability.
  - Algorithm (ATVA’07).
- Timed Games with Büchi Condition
  - Algorithm (CAV’09)

Controller Synthesis/TGA

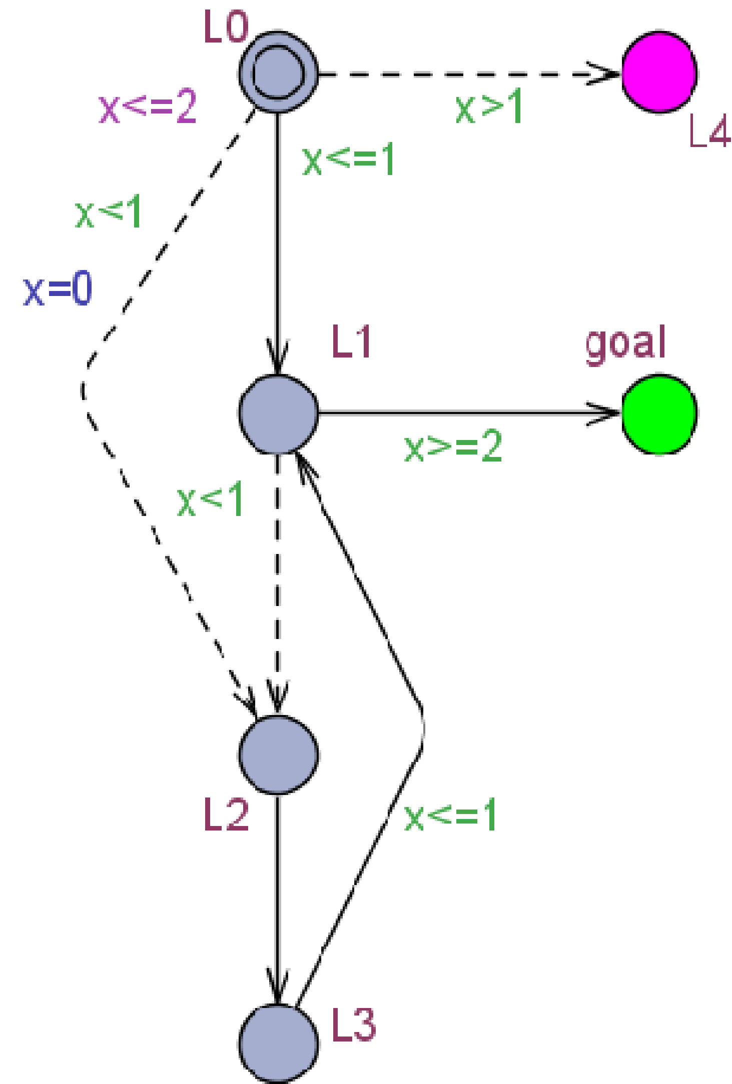
- Given
  - System moves S,
  - Controller moves C,
  - and a property φ,
- find
  - a strategy S_c s.t. $S_c || S \models \varphi$,
  - or prove there is no such strategy.

Timed Game Automata

- CONCUR'05 (Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, Didier Lime)
- The controller continuously observes the system (all delays & moves are observable).
- The controller can
  - wait (delay action),
  - take a controllable move (=prevent delay).
- TIGA:
  - Priority to the environment.
  - Forced actions.

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 → action.



TGA – Let's Play!

- control: A<> TGA.goal

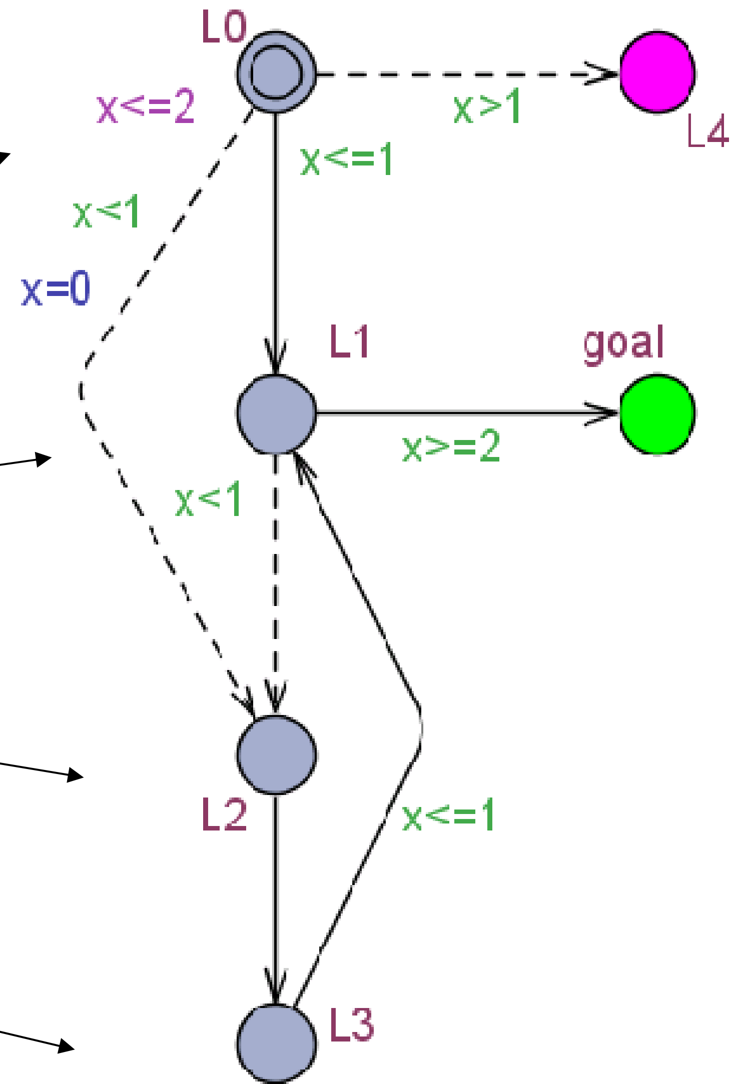
$x<1$	: λ
$x==1$	: c

$x<2$	: λ
$x\geq 2$	: c

Strategy

$x\leq 1$	: c

$x<1$	: λ
$x==1$	: c



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

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.

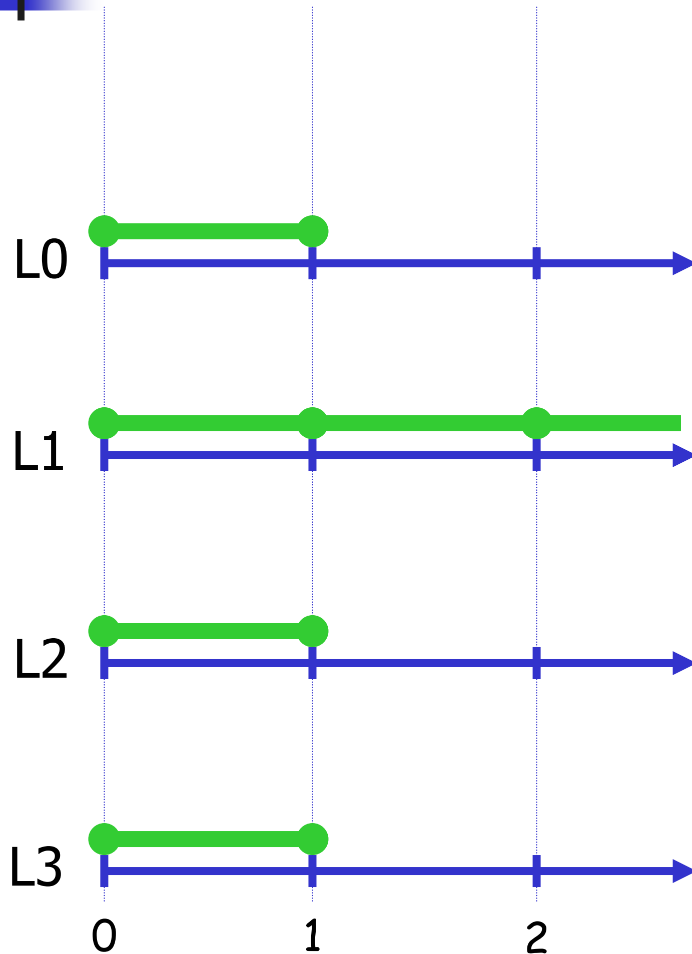
Initialization:

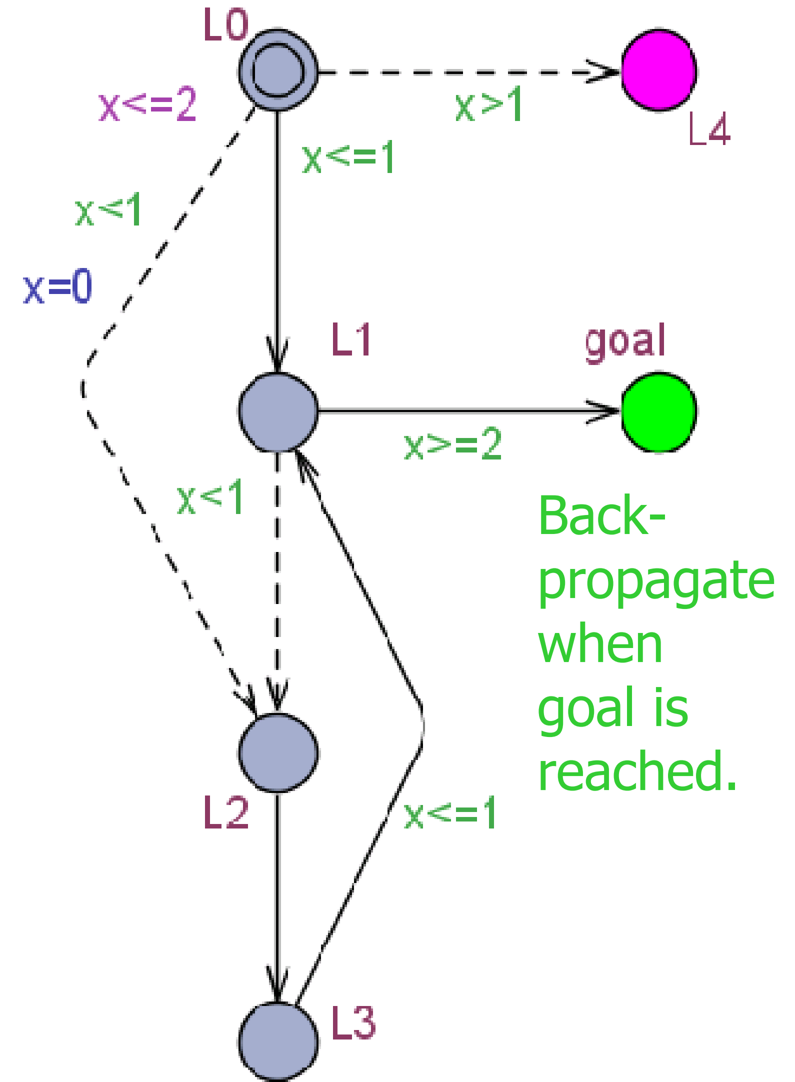
$Passed \leftarrow \{S_0\}$ **where** $S_0 = \{(\ell_0, \vec{0})\}^{\nearrow}$;
$Waiting \leftarrow \{(S_0, \alpha, S') \mid S' = \mathsf{Post}_\alpha(S_0)^{\nearrow}\}$;
$Win[S_0] \leftarrow S_0 \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
$Depend[S_0] \leftarrow \emptyset$;

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**
$Passed \leftarrow Passed \cup \{S'\}$;
$Depend[S'] \leftarrow \{(S, \alpha, S')\}$;
$Win[S'] \leftarrow S' \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
$Waiting \leftarrow Waiting \cup \{(S', \alpha, S'') \mid S'' = \mathsf{Post}_\alpha(S')^{\nearrow}\}$;
if $Win[S'] \neq \emptyset$ **then** $Waiting \leftarrow Waiting \cup \{e\}$;
else (* **reevaluate** *)[a]
$Win^* \leftarrow \mathsf{Pred}_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} \mathsf{Pred}_c(Win[T]),$
$\bigcup_{S \xrightarrow{u} T} \mathsf{Pred}_u(T \setminus Win[T])) \cap S$;
if $(Win[S] \subsetneq Win^*)$ **then**
$Waiting \leftarrow Waiting \cup Depend[S]$; $Win[S] \leftarrow Win^*$;
$Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;
endif
endwhile

Backward Propagation

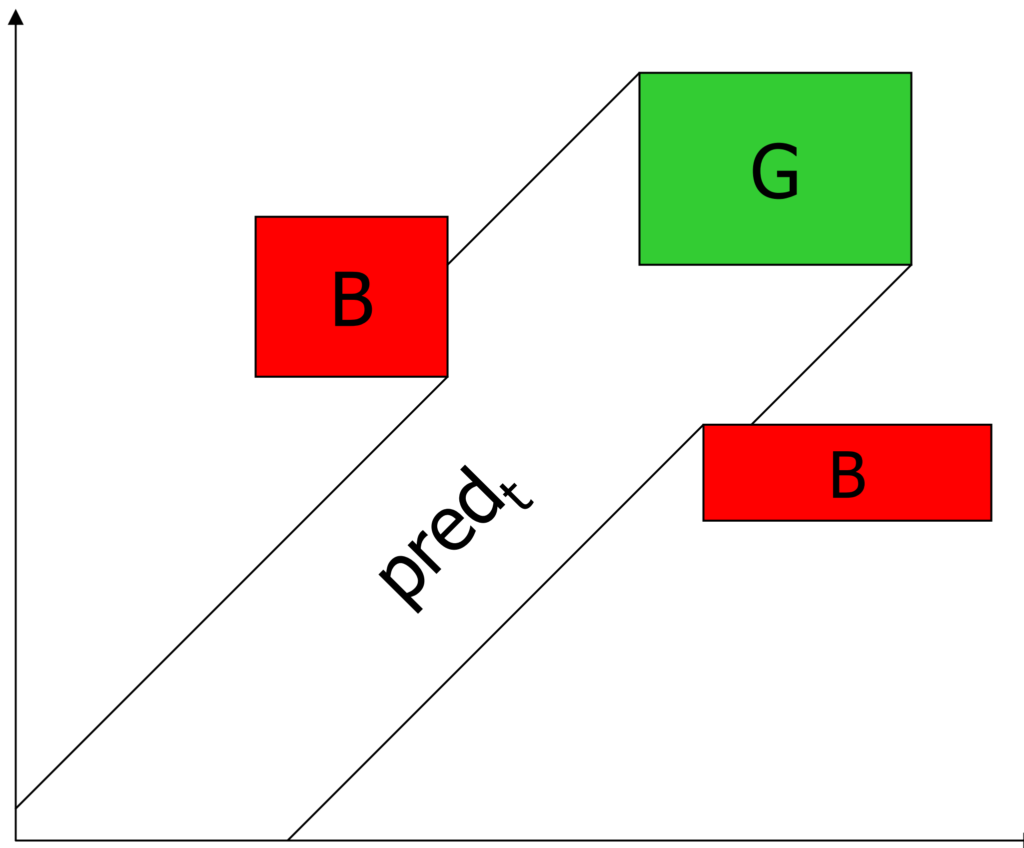




Back-propagate when goal is reached.

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

Backward Propagation



Predecessors of G avoiding B?

$pred_t$ – From Federation to Zone

$$pred_t\left(\bigcup_i G_i, \bigcup_j B_j\right) = \bigcup_i \bigcap_j pred_t(G_i, B_j)$$

$$pred_t(G, B) = (G^{\downarrow} \setminus B^{\downarrow}) \bigcup ((G \bigcap B^{\downarrow}) \setminus B)^{\downarrow}$$

Query Language (1)

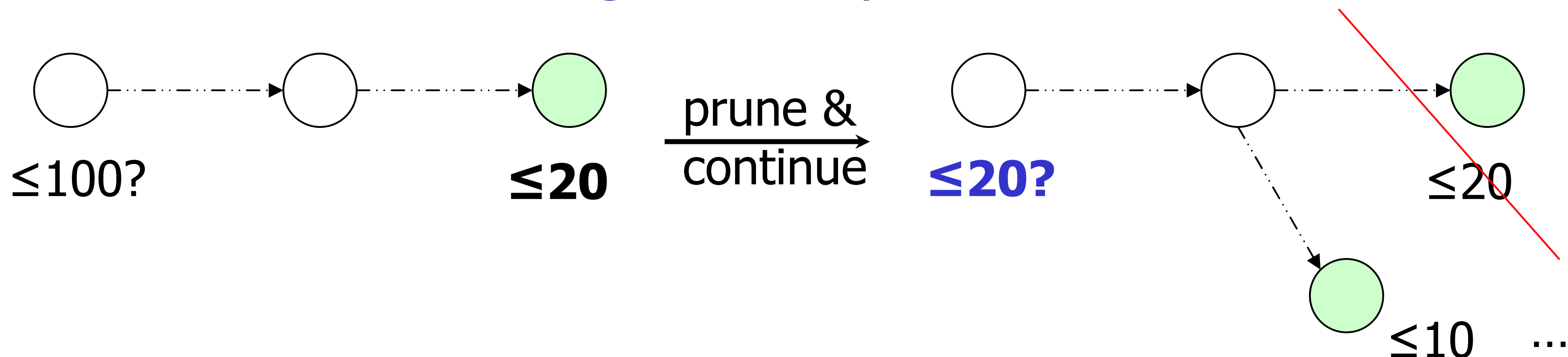
- Reachability properties:
  - control: A[p U q]
  - control: A<> q ⇔ control: A[true U q]
- Safety properties:
  - control: A[p W q]
  - control: A[] p ⇔ control: A[p W false]
- Tuning:
  - change search ordering,
  - add back-propagation of winning+losing states.

Back-propagate winning states & BFS+DFS.

Back-propagate losing states & BFS-BFS.

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. *Evaluated once, updated later.*
    - 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.

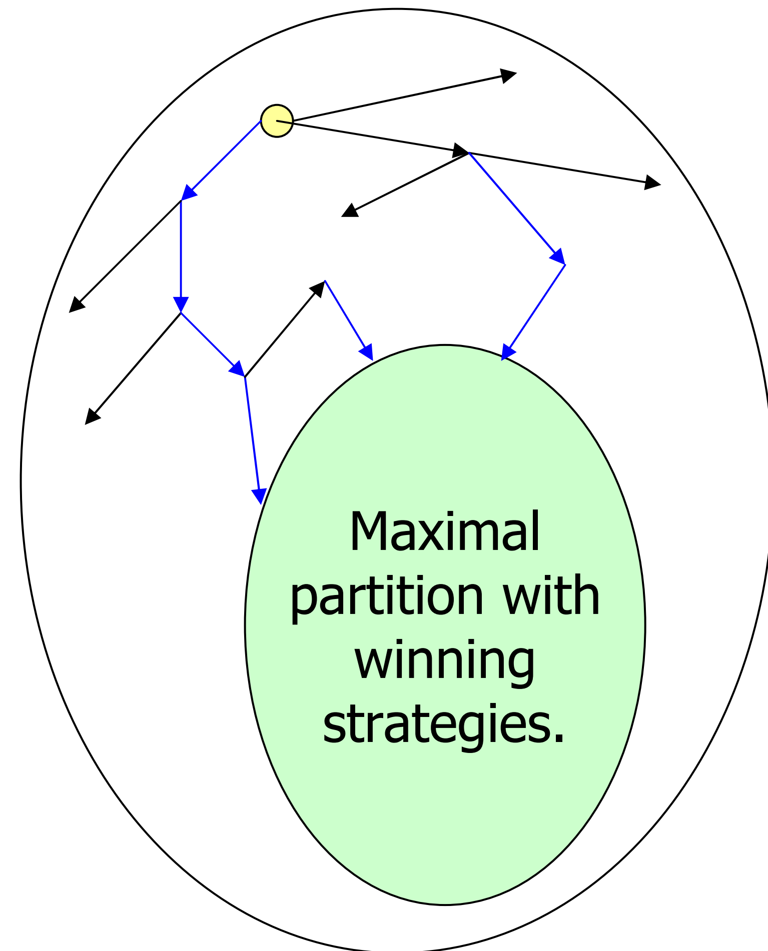


Query Language (3)

- Cooperative strategies.
  - E<> control: φ
  - Property satisfied iff φ is reachable but the obtained strategy is maximal.
  - Used in testing.
  - Useful to study faults.
- [MBT’08] Shuaho Li, Kim G. Larsen, Brian Nielsen, Alexandre David.

Cooperative Strategies

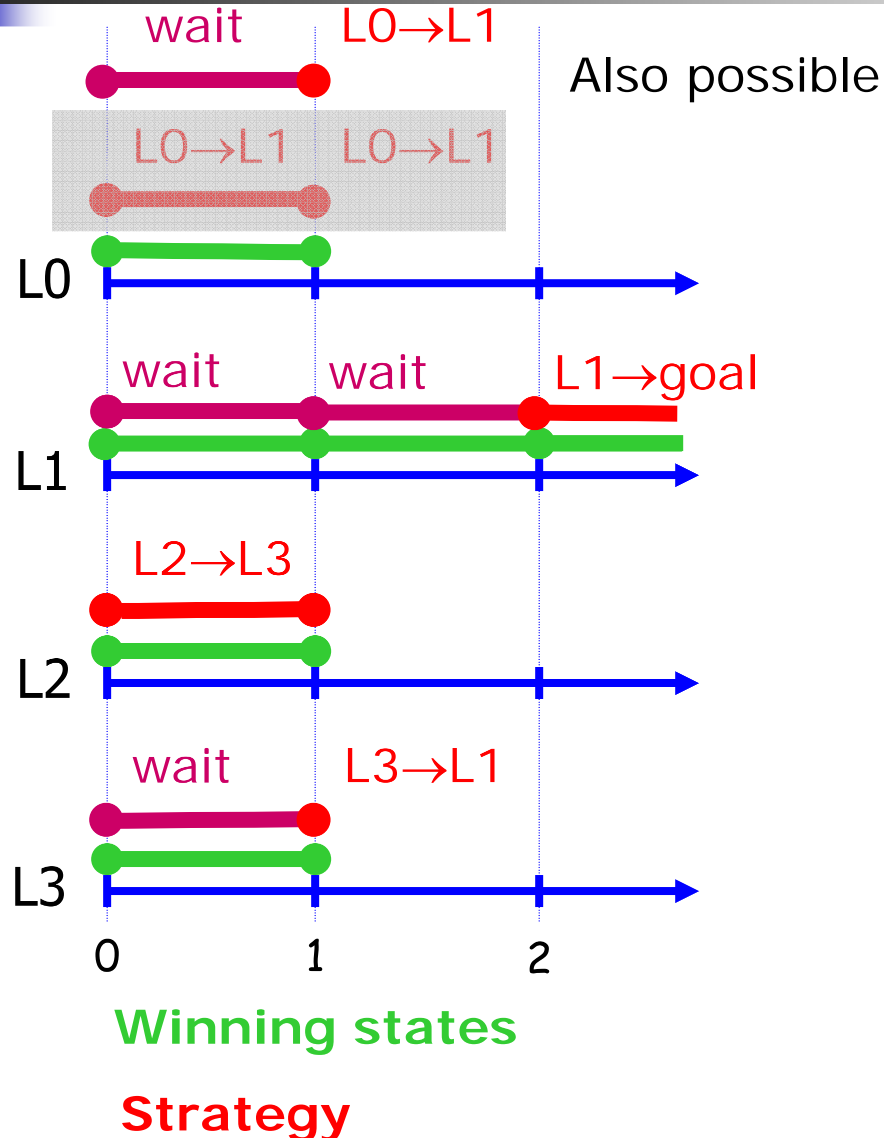
- State-space is partitioned between states from which there is a strategy and those from which there is no strategy.
- Cooperative strategy suggests moves from the opponent that would “help” the controller.
- Being used in testing.

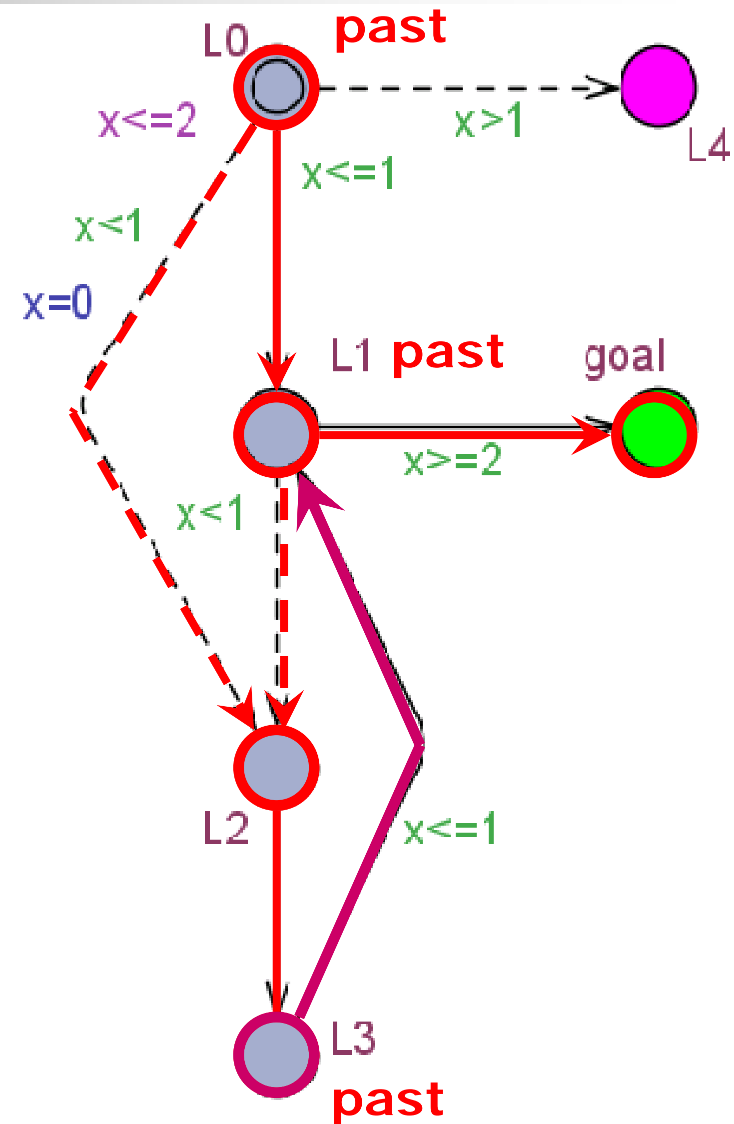


Strategies?

- 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





Strategies as Partitions

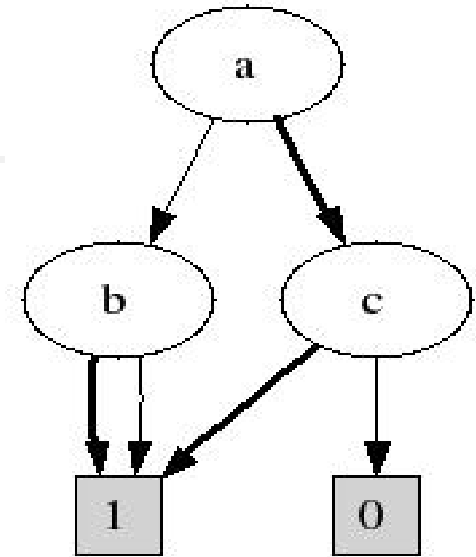
- Built on-the-fly.
- Guarantee progress in the strategy.
  - No loop.
- Deterministic strategy.
- Different problem than computing the set of winning states.
- *Different ordering searches* can give *different strategies* … with possibly the same set of winning states.

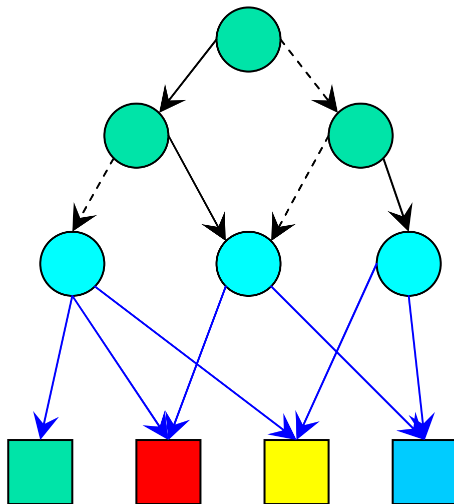
Code Generation

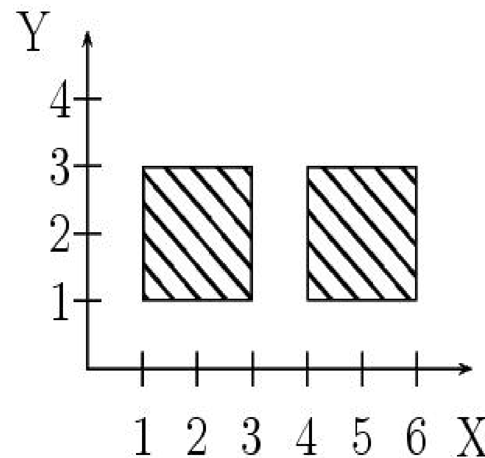
- Mapping state → action.
  - # entries = # states.
- Decision graph state → action.
  - # tests = # variables.
  - More compact.
  - Based on a hybrid BDD/CDD with multi-terminals.

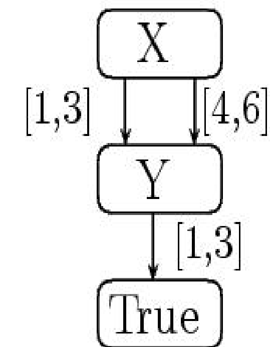
Decision Graph

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



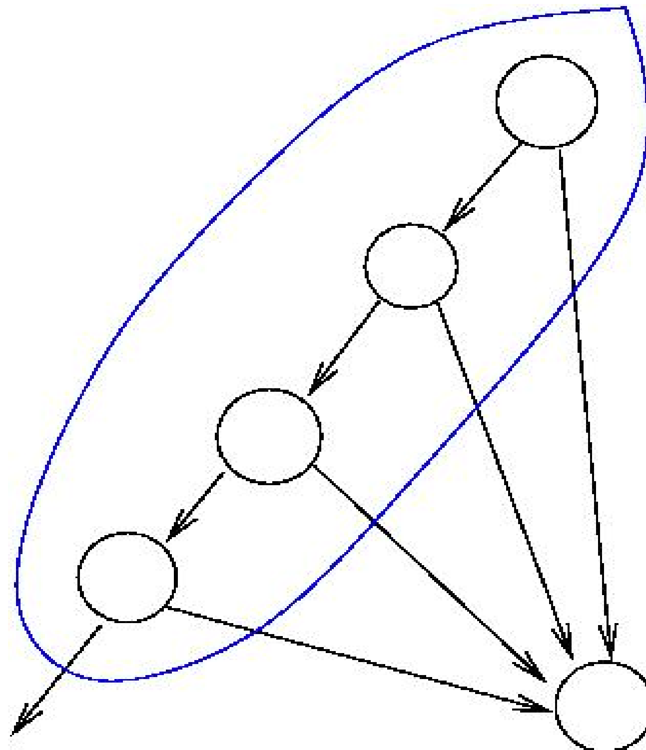




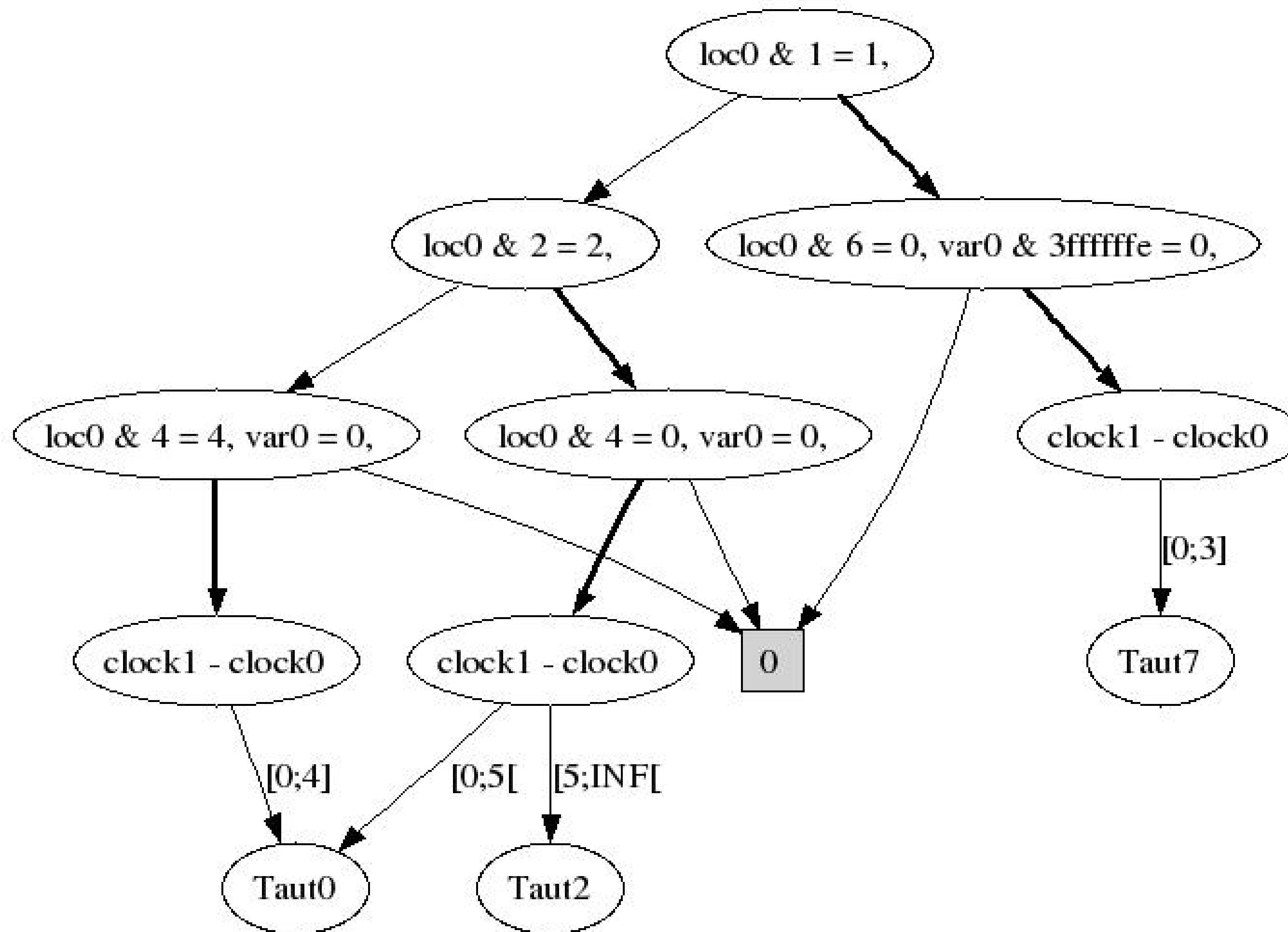


Graph Reduction

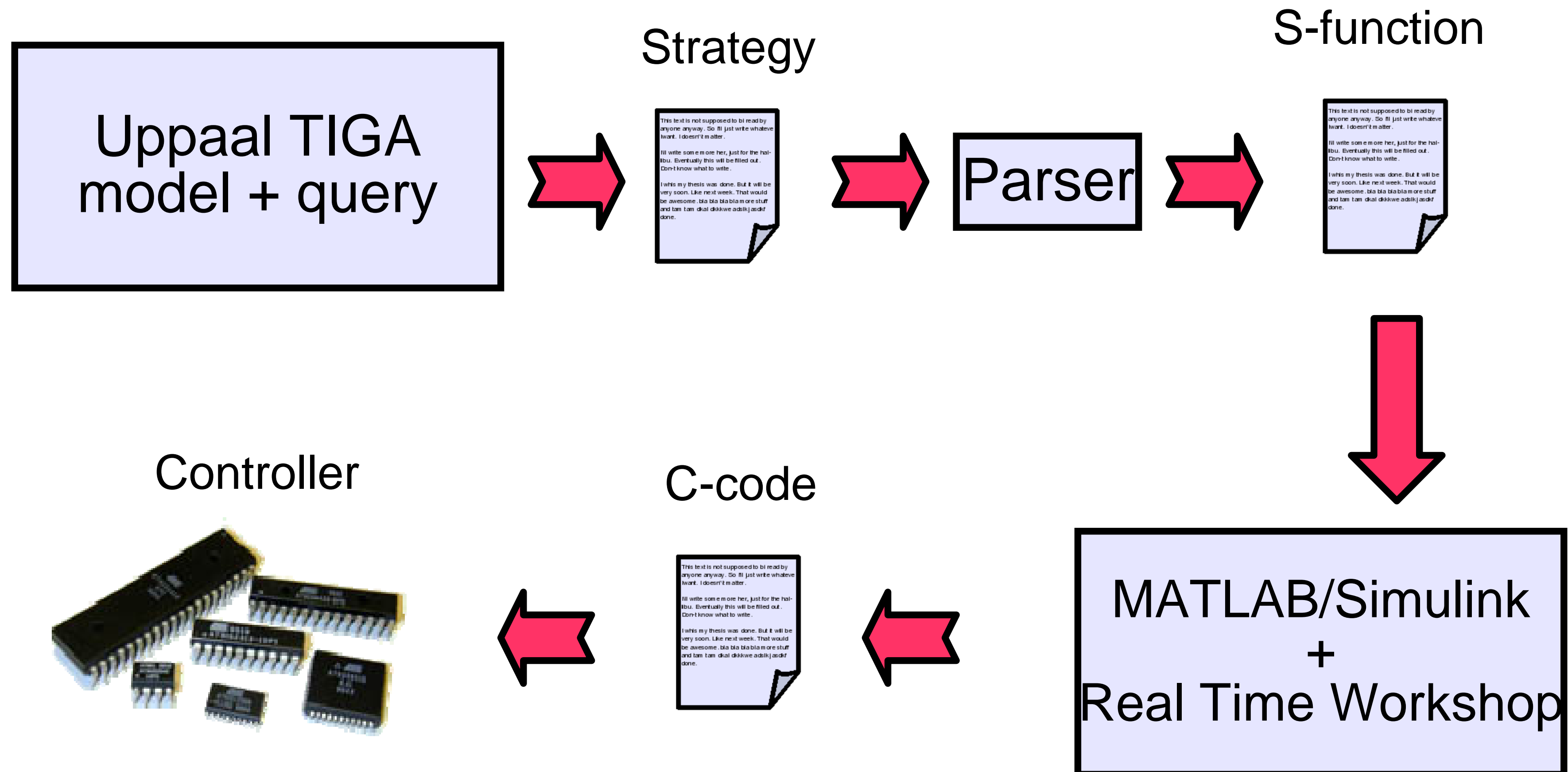
- Testing consecutive bits:
  - Replace by one testing with a mask.
  - Can span on several variables.



Decision Graph – Example

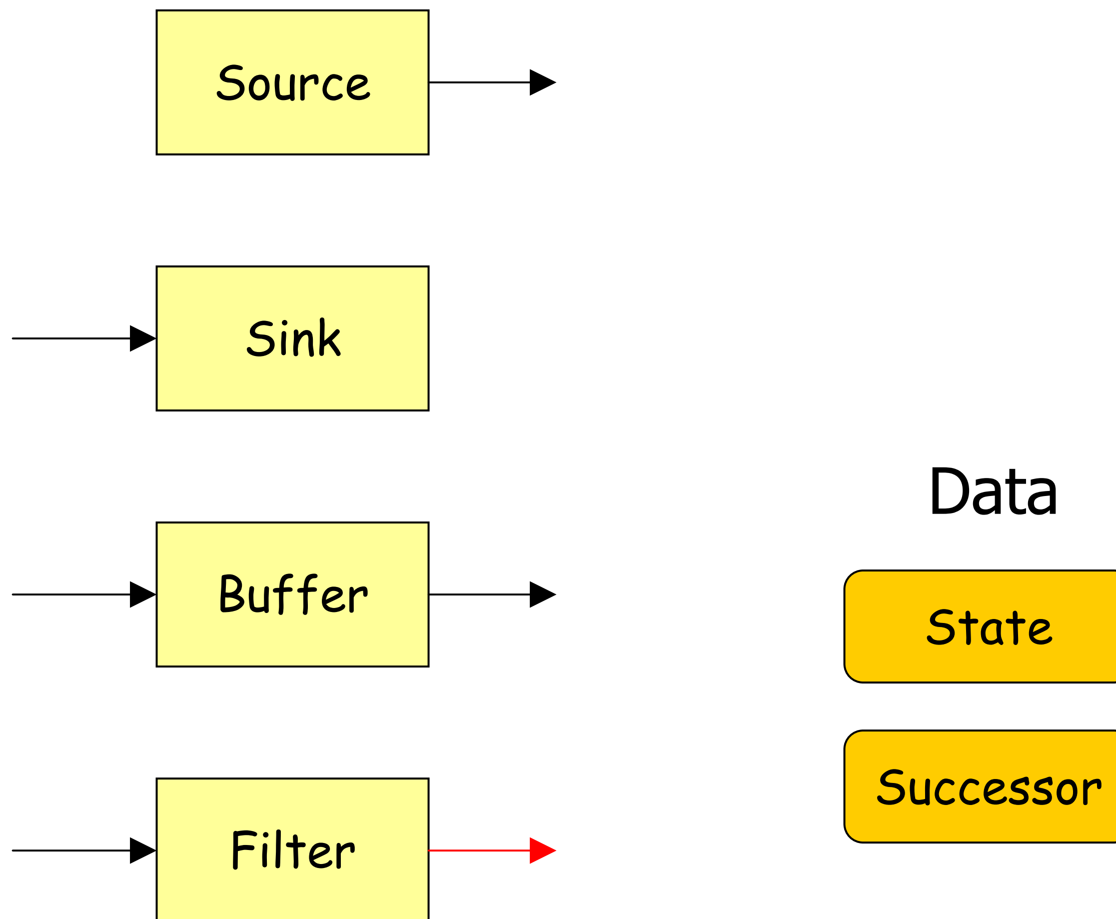


Tool Chain [FORMATS'07]

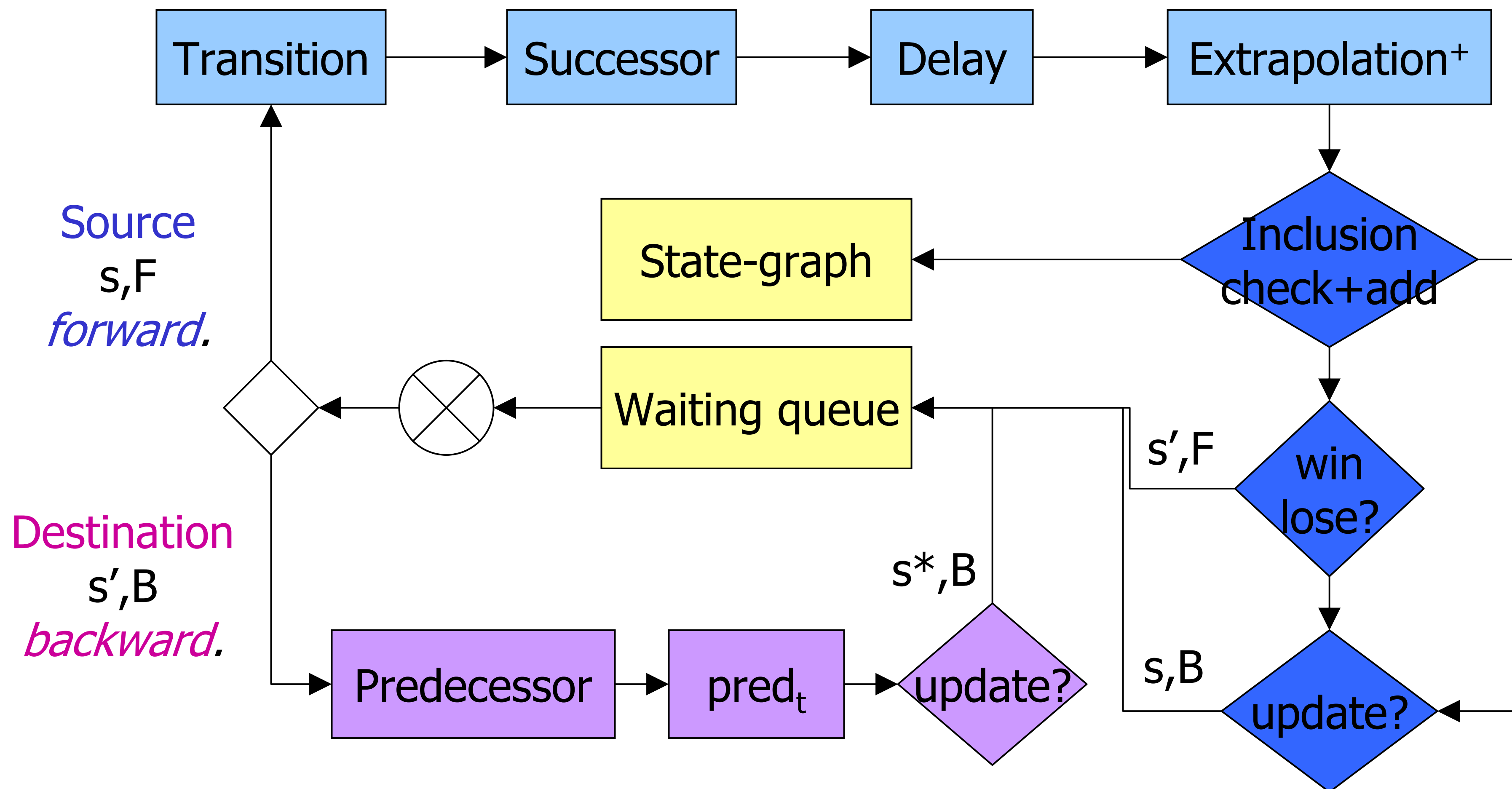


Similar tool chain for Hydac [HSCC09]

Pipeline Components



Pipeline Architecture

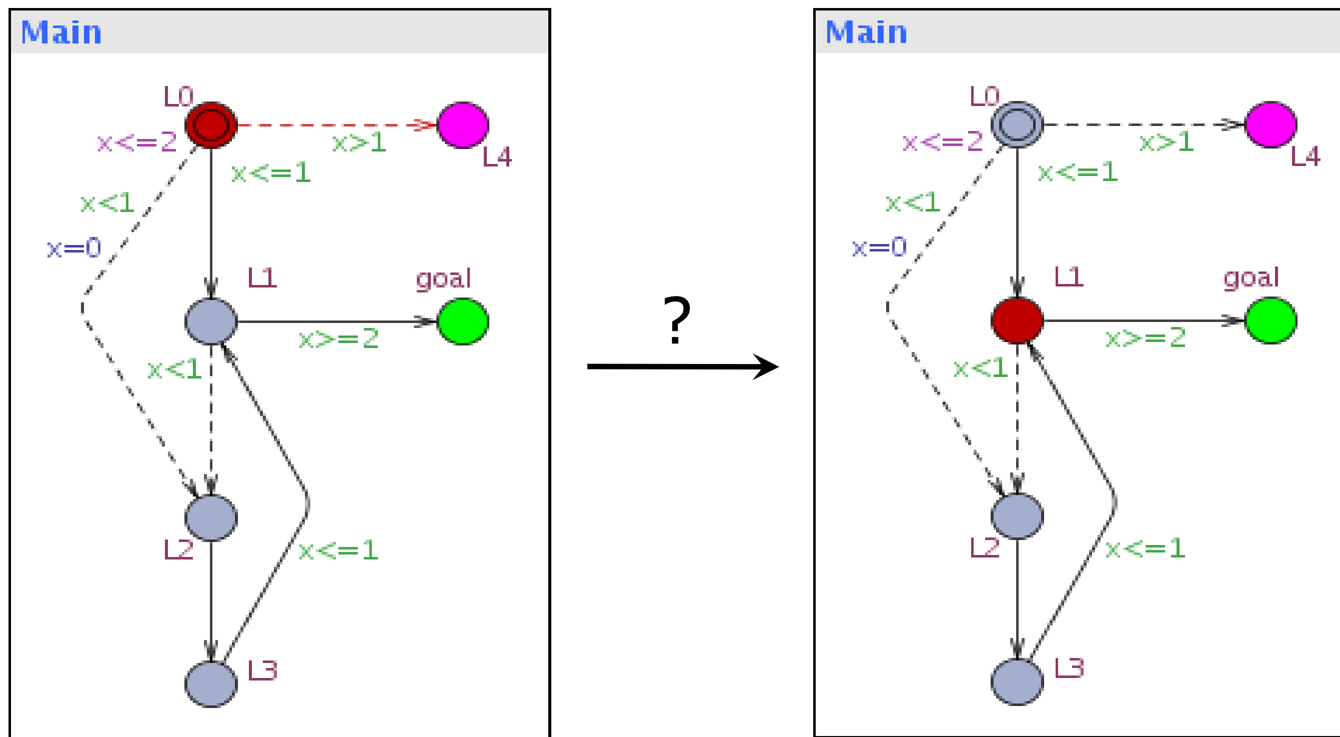


Interactive Game

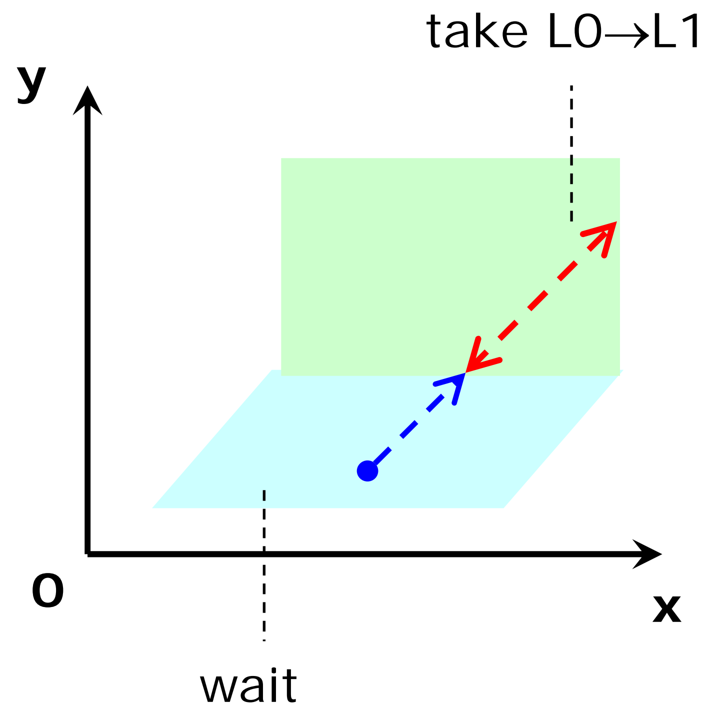
- How to play a (timed) strategy against the user?
  - Concrete simulator.
  - Actions depend on the point in time.
  - Allowed delays depend on the actions.
  - The GUI has limited feedback for showing *counter-actions.*

Interactive Game

- Goal: Play the game inside UPPAAL GUI.
- Problem: The GUI is not as talkative as a command line simulator !



From Symbolic to Concrete

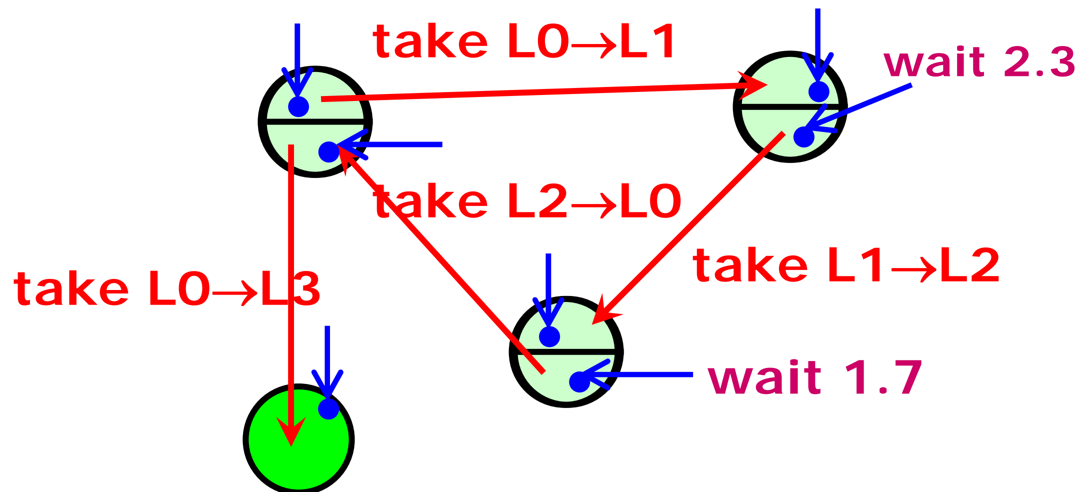


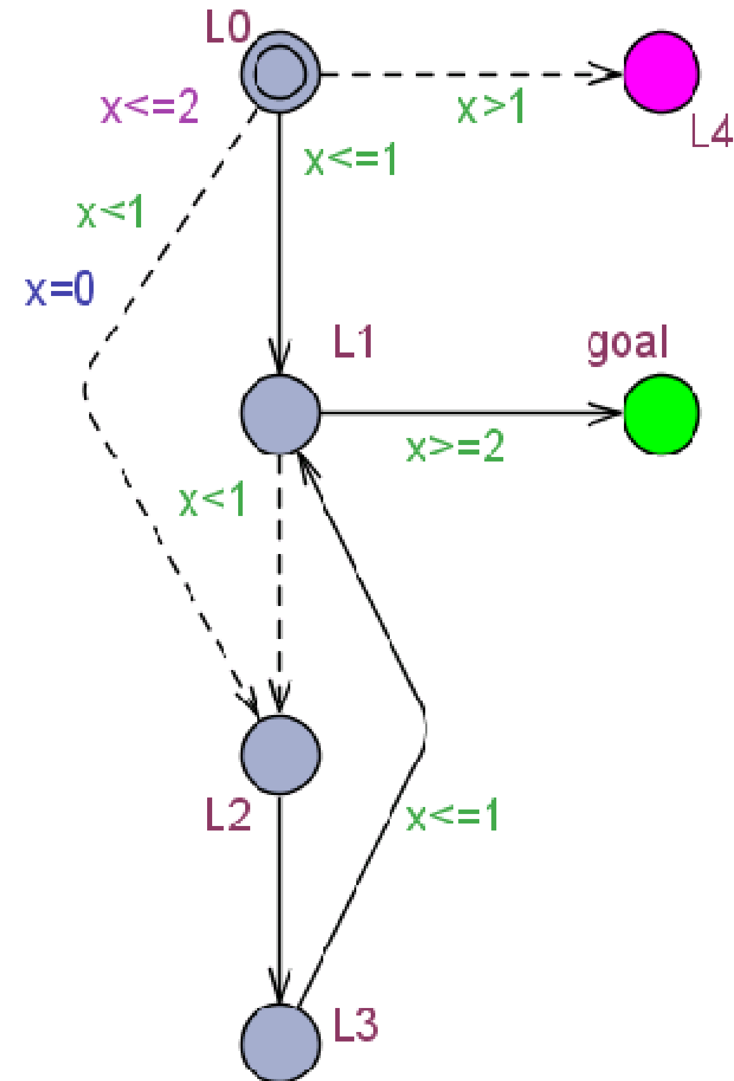
Valid interval for taking action.

How long to wait before taking action.

From Symbolic to Concrete

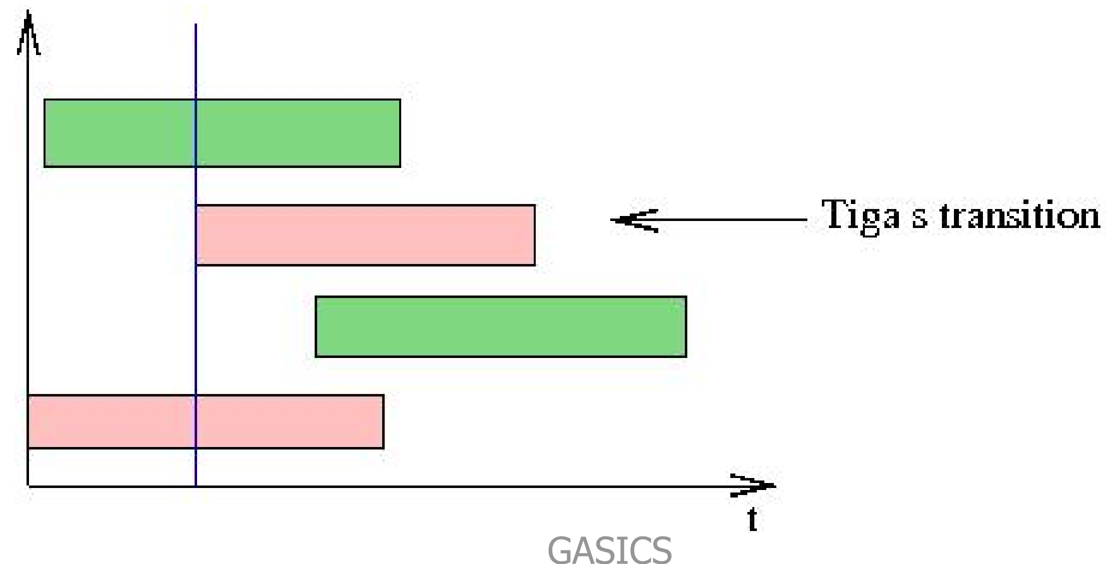
- Strategy = mapping from sets of states to actions (incl. wait).
- Simulation with a given clock valuation.





Interactive Game – GUI

- Avoid “your action has been countered”: Restrict selection w.r.t. the strategy.
- What is a “selectable action” for the user ?
  - His own transition – if can take it before TIGA
  - The choice of TIGA
- the other actions are not selectable



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.
- The game is played differently.
- [ATVA'07] – Franck Cassez, Alexandre David, Kim G. Larsen, Didier Lime, Jean-François Raskin

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.
- 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.

Notes

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

On-the-Fly Algorithm

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:

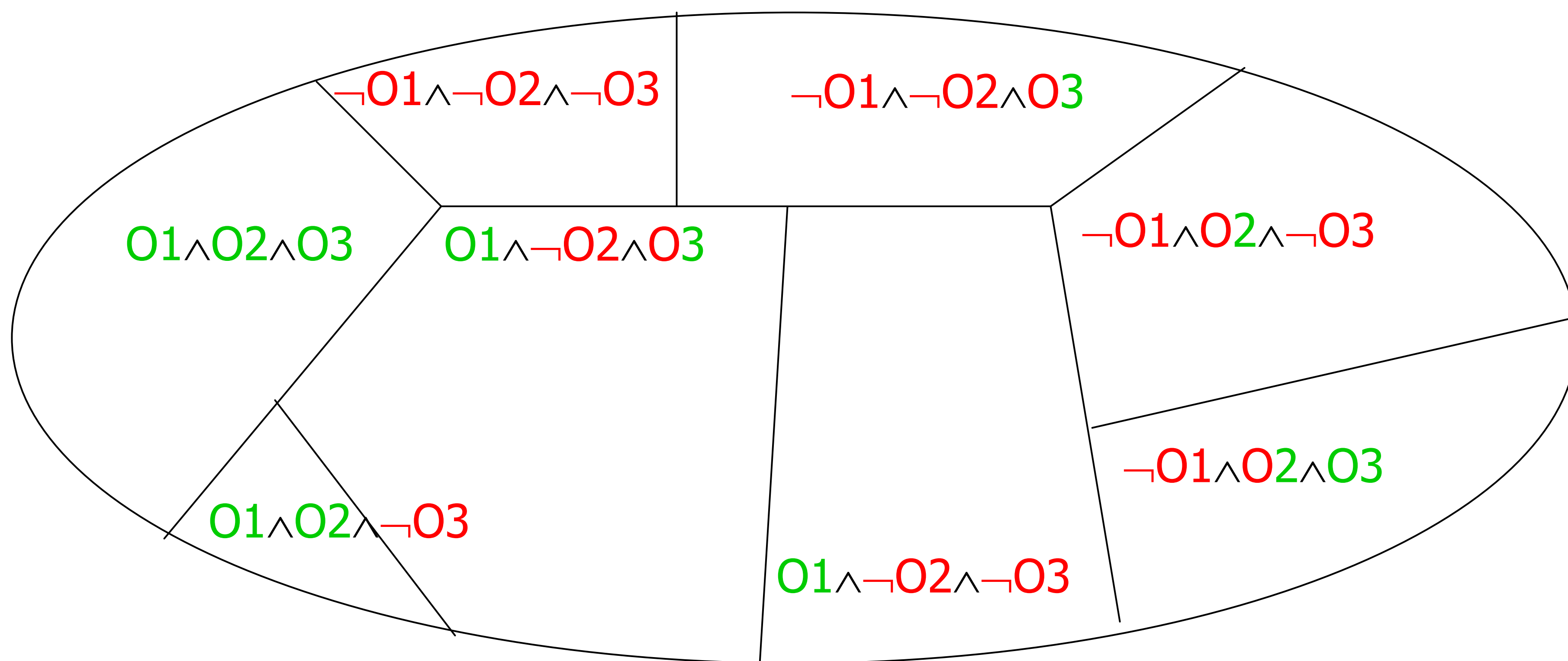
while $((Waiting \neq \emptyset) \wedge Win[\{s_0\}] \neq 1 \wedge Losing[\{s_0\}] \neq 1))$ **do**
  $e = (W, \alpha, W') \leftarrow pop(Waiting)$;
  if $s' \notin Passed$ **then**
    $Passed \leftarrow Passed \cup \{W'\}$;
    $Depend[W'] \leftarrow \{(W, \alpha, W')\}$;
    $Win[W'] \leftarrow (W' \subseteq \gamma(\mathsf{Goal}) \; ? \; 1 : 0)$;
    $Losing[W'] \leftarrow (W' \not\subseteq \gamma(\mathsf{Goal}) \wedge \mathsf{Sink}_\alpha(W') \neq \emptyset \; ? \; 1 : 0)$;
    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

Algorithm

Partition the state-space w.r.t. observations.

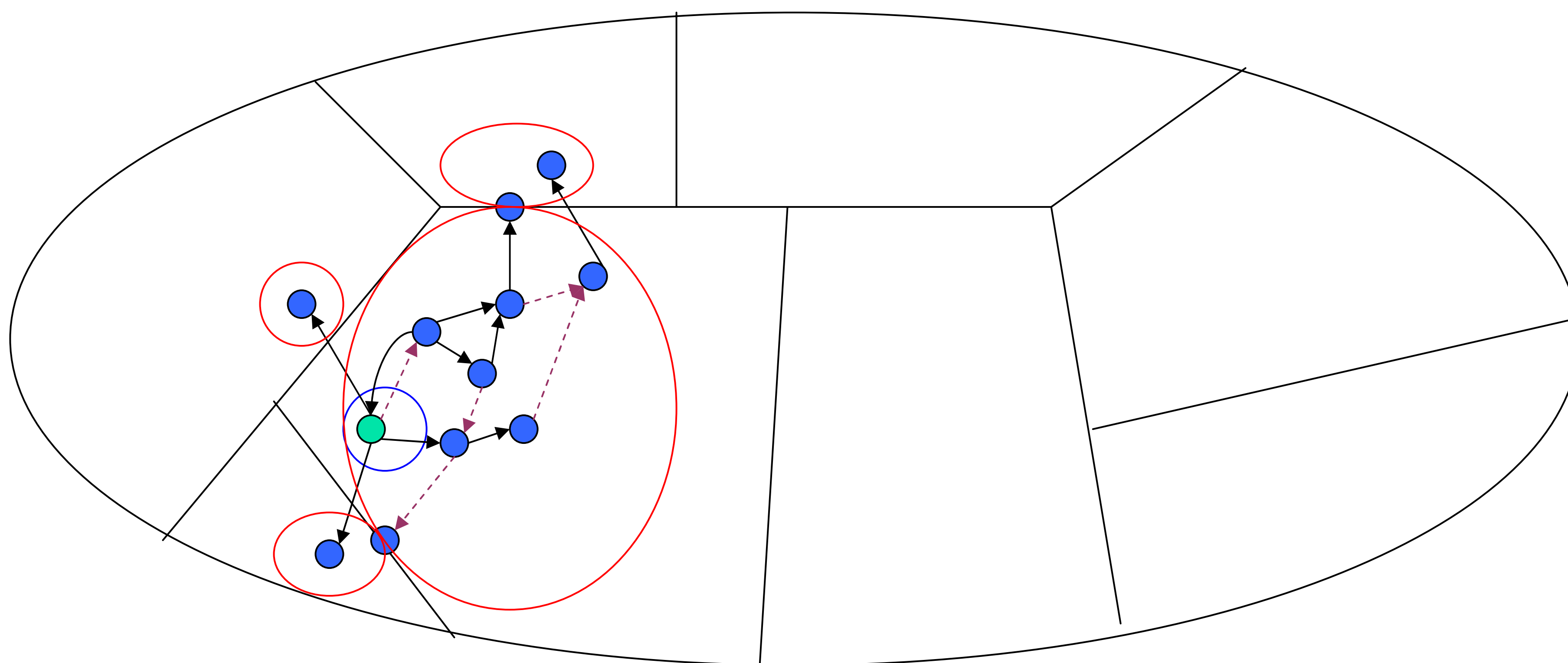
Observations O1 O2 O3.

Winning/losing is observable.



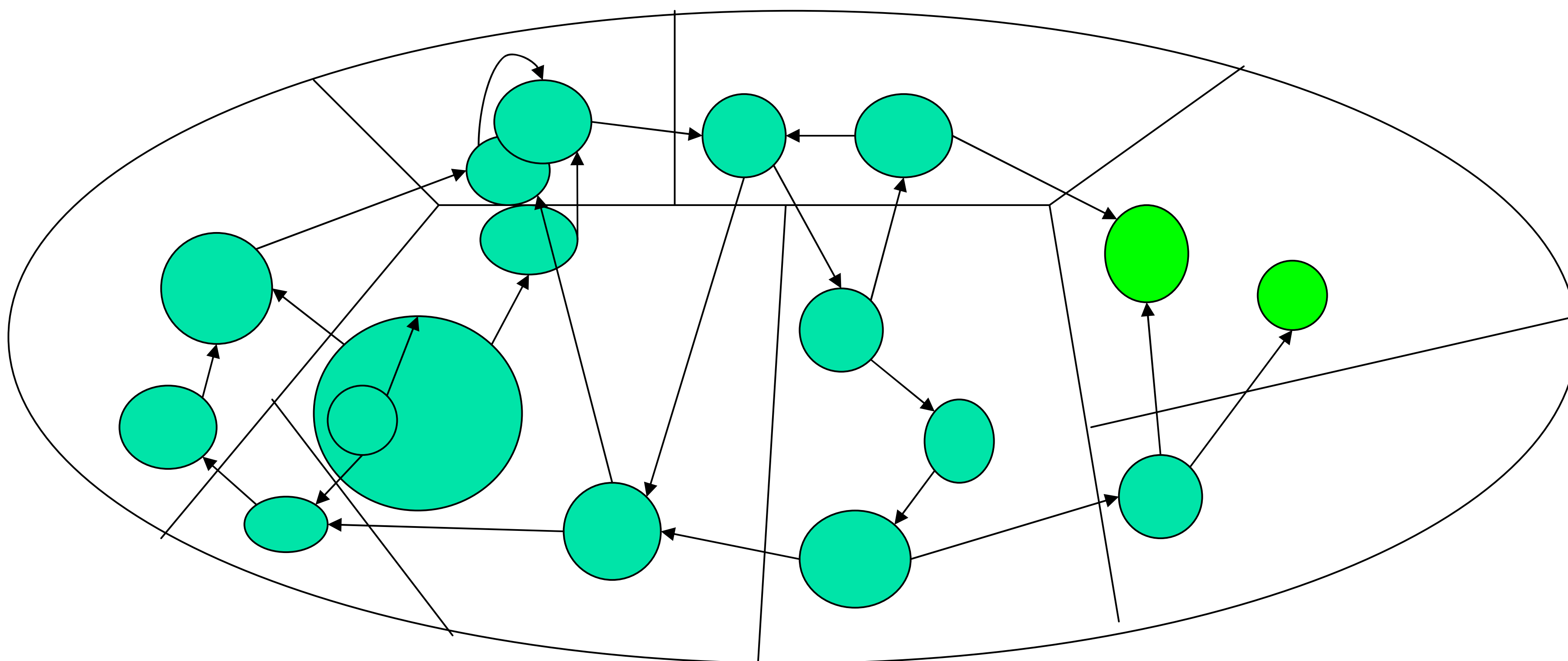
Algorithm

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



Algorithm

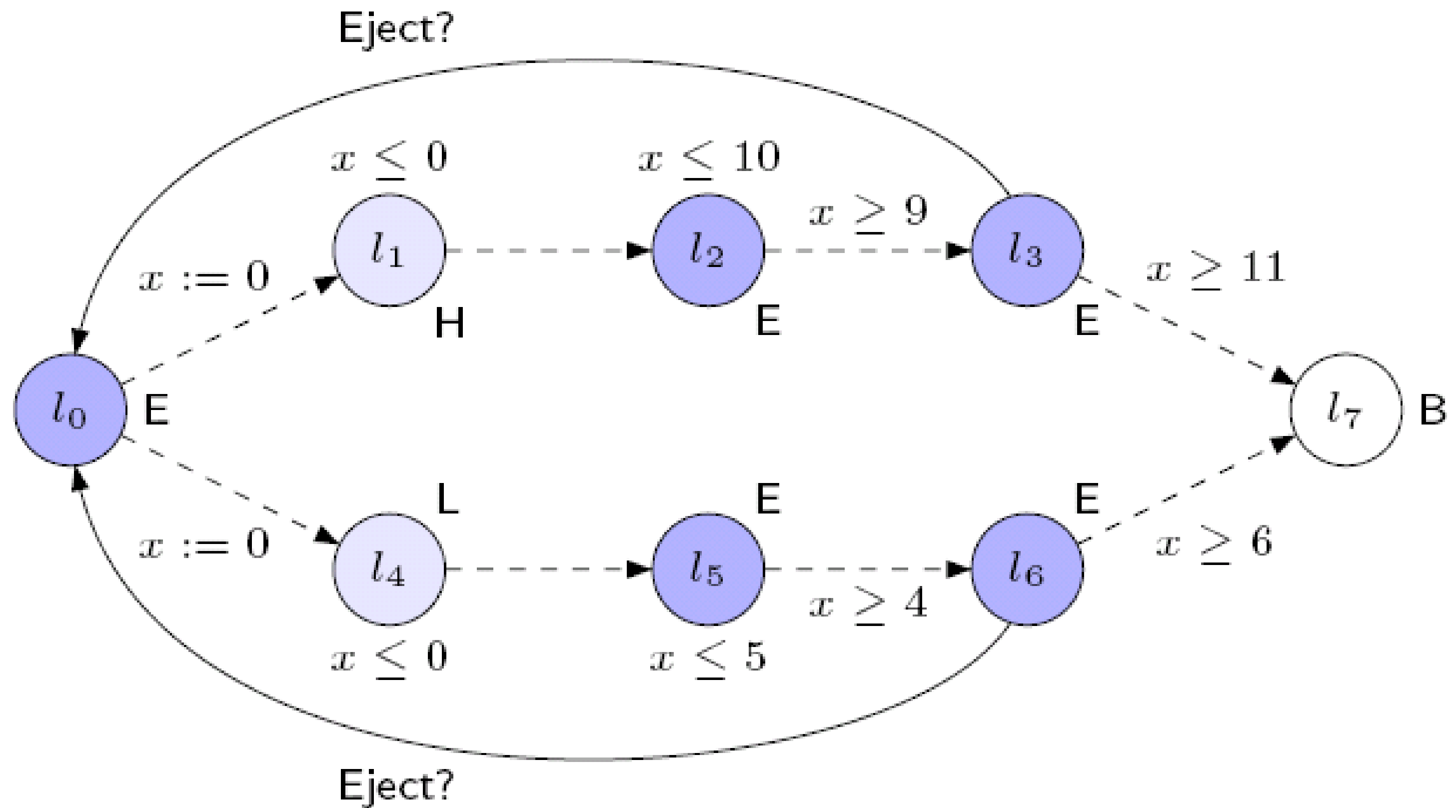
Construct the graph of sets of symbolic states.
Back-propagate winning/losing states.



Algorithm

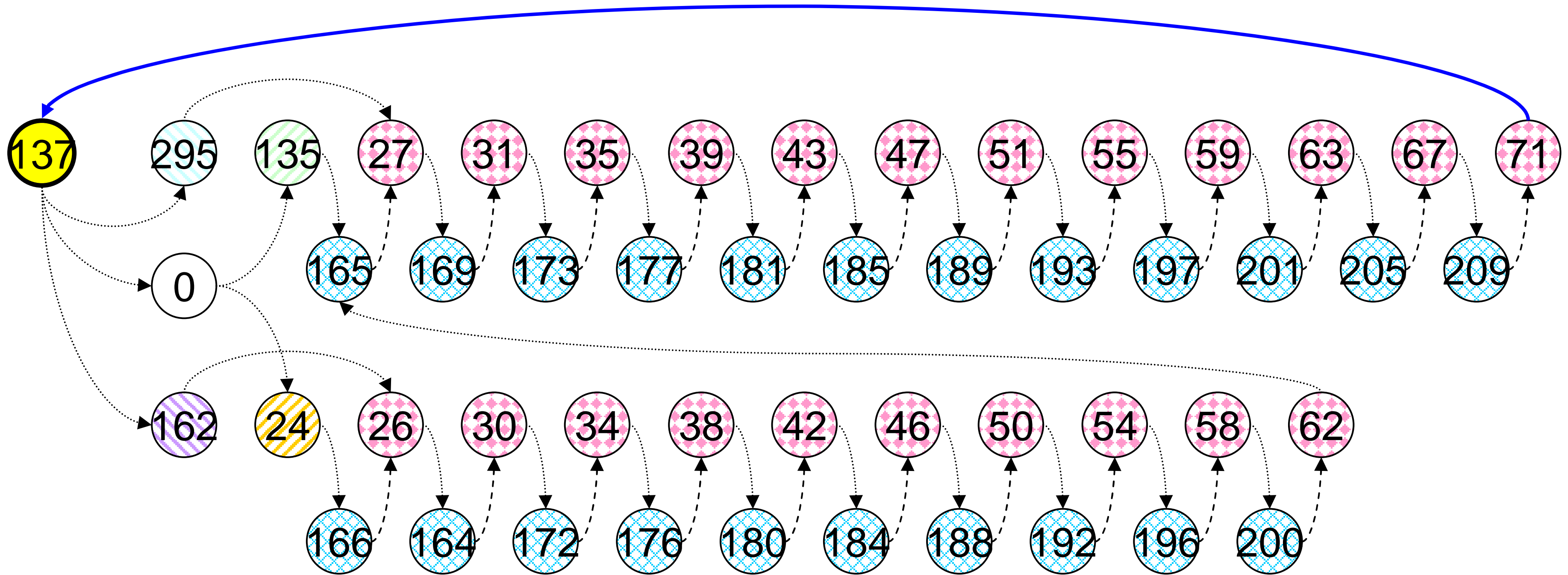
- Forward exploration
  - constrained by action + observations
  - delay special
- Back-propagation.
  - If all successors^a are winning,
    declare current state winning,
    strategy: take action *a*.
  - If one successor^a is losing,
    avoid action *a*.
    If no action is winning the current state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

Example



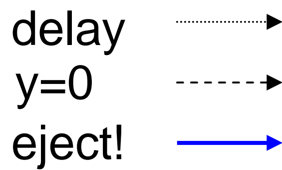
Observations: L, H, E, B, y in [0,1[

Example



Partition:

Actions:



TGA with Buchi Accepting States

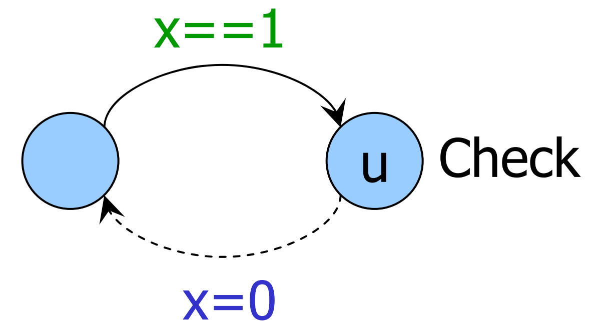
- Use TIGA's algorithm as an intermediate steps in a fixpoint, but modified:
  - winning states are states that can reach goals
  - goal states defined by queries, not necessarily winning
  - Fixpoint:

```
Win = SOTF(goal)
while (Win ∩ goal) ≠ goal
    goal = Win ∩ goal
    Win = SOTF(goal)
done
return S0∈Win
```

Didier Lime

Application: Non-zeno Strategies

- Add a monitor with the rest of the system.
- Ask
  control: A[] (p && A<> Monitor.Check)



TODO List for TIGA

- Improve PO-TIGA – 0.13.
- Checking simulation of TA & TGA – 0.14.
  - with Peter Bulychev + Thomas Chatain
- Improving the simulator (PO-TIGA).
- Work with lattice of observations.

TODO List for UPPAAL

- Specifying properties with live sequence charts
  - start in April – Sandie Balaguer
- Gantt chart in the simulator
  - in-progress – Morten Kuhnrich
- Plugin architecture for the tool
  - student programmer working on TRON
- Concrete simulator

Restrictions on TGA

- TIGA:
  - weak invariants (≤)
  - priorities not supported
  - stop watches not supported
- + TIGA-PO
  - weak lower bounds on (controllable) guards (≥)
  - clock intervals in the observations of the form [a,b)
  - no clock differences in the observations