



Timed Games

UPPAAL-TIGA

Alexandre David

1.2.05

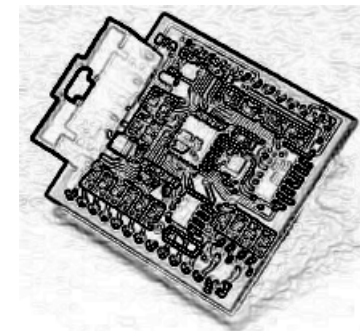
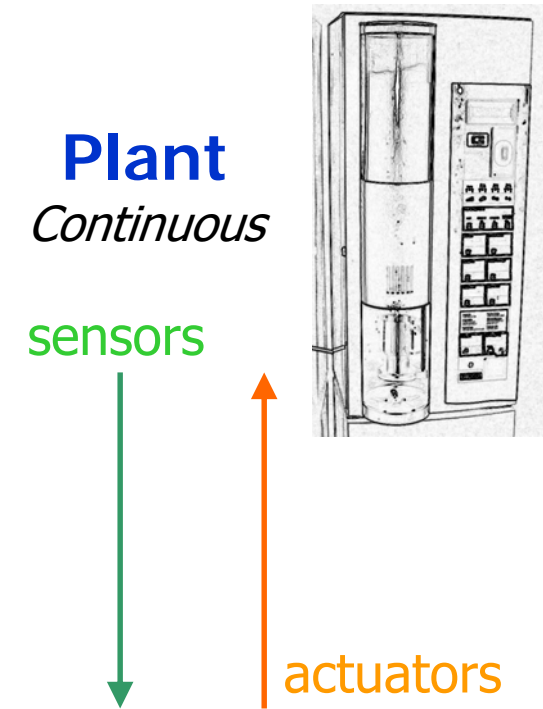


Overview

- Timed Games.
 - Algorithm (CONCUR'05).
 - Strategies.
 - Code generation.
 - Architecture of UPPAAL-TIGA.
 - Interactive game.
- Timed Games with Partial Observability.
 - Algorithm (ATVA'07).

Why Timed Games?

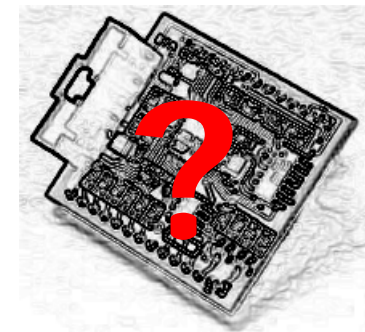
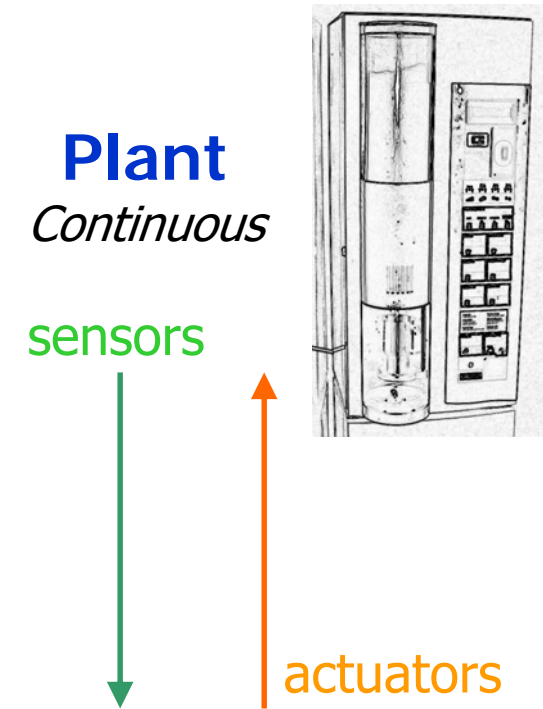
- **Real-time systems:**
 - Systems where correctness depends on the logical order of events and on their **timings!**
 - ... in addition to correct computation.
- **Real Time Model-checking:**
 - Model the environment + the tasks.
 - **Model $\models \varphi$? Automated proof.**



Controller Program
Discrete

Why Timed Games?

- **Controller synthesis:**
 - Model the environment + what a controller *can* do.
 - Generate the controller so that **controller $\models \varphi!$**
Generate the *right* code automatically.
 - 2-player timed game: environment moves vs. controller moves.
 \Rightarrow **Timed Game Automata.**



Controller Program
Discrete



Controller Synthesis/TGA

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

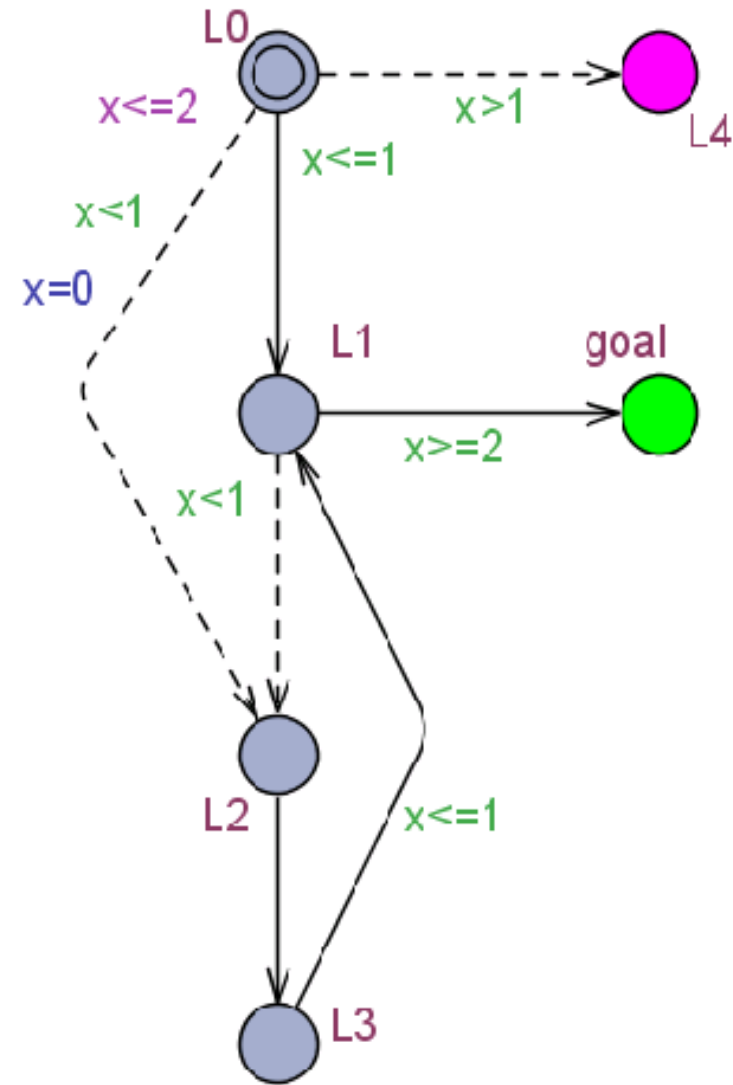


Timed Game Automata

- 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: $A \langle \rangle$ TGA.goal
 - control: $A[]$ not TGA.L4
- Memoryless strategy:
 - state \rightarrow action.



TGA – Let's Play!

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

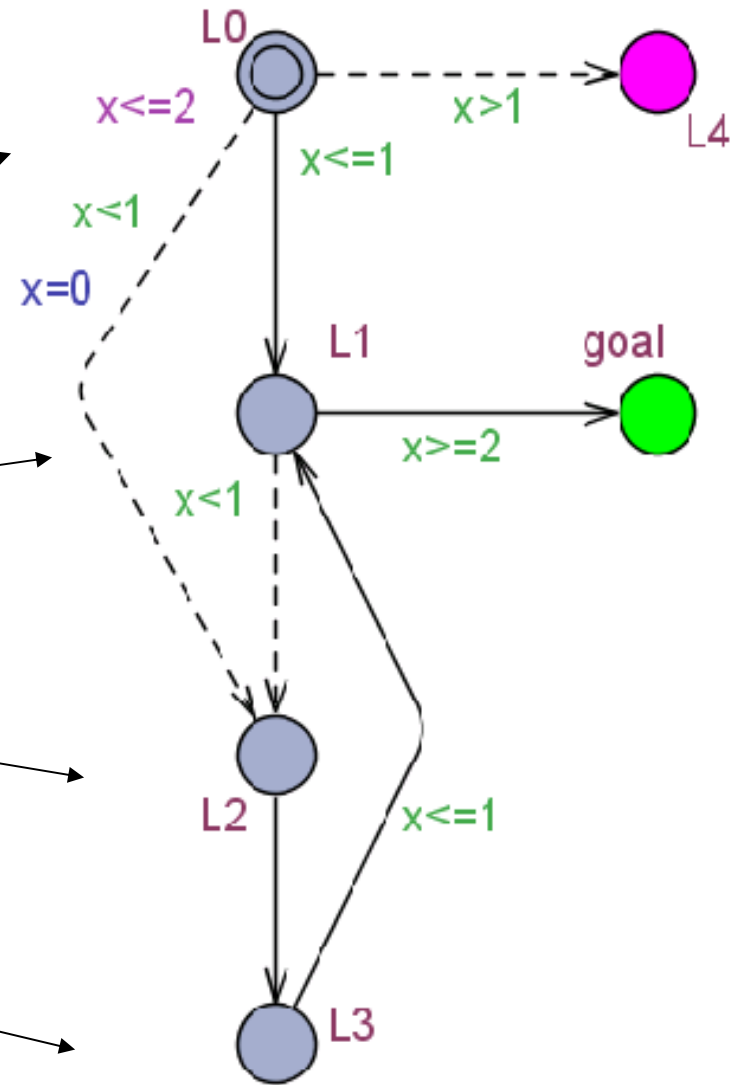
$x < 1$: λ
 $x == 1$: c

$x < 2$: 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.



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**.
- [Cassez & al. '05]
Efficient **on-the-fly algorithm** for safety & reachability games.



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 $S_0 = \{(l_0, \vec{0})\}^{\nearrow}$;
 $Waiting \leftarrow \{(S_0, \alpha, S') \mid S' = \text{Post}_\alpha(S_0)^{\nearrow}\}$;
 $Win[S_0] \leftarrow S_0 \cap (\{\text{Goal}\} \times \mathbb{R}_{\geq 0}^X)$;
 $Depend[S_0] \leftarrow \emptyset$;

Main:

while $((Waiting \neq \emptyset) \wedge (s_0 \notin Win[S_0]))$ **do**

$e = (S, \alpha, S') \leftarrow \text{pop}(Waiting)$;

if $S' \notin Passed$ **then**

$Passed \leftarrow Passed \cup \{S'\}$;

$Depend[S'] \leftarrow \{(S, \alpha, S')\}$;

$Win[S'] \leftarrow S' \cap (\{\text{Goal}\} \times \mathbb{R}_{\geq 0}^X)$;

$Waiting \leftarrow Waiting \cup \{(S', \alpha, S'') \mid S'' = \text{Post}_\alpha(S')^{\nearrow}\}$;

if $Win[S'] \neq \emptyset$ **then** $Waiting \leftarrow Waiting \cup \{e\}$;

else $(* \text{reevaluate} *)^a$

$Win^* \leftarrow \text{Pred}_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} \text{Pred}_c(Win[T]),$
 $\bigcup_{S \xrightarrow{u} T} \text{Pred}_u(T \setminus Win[T])) \cap S$;

if $(Win[S] \subsetneq Win^*)$ **then**

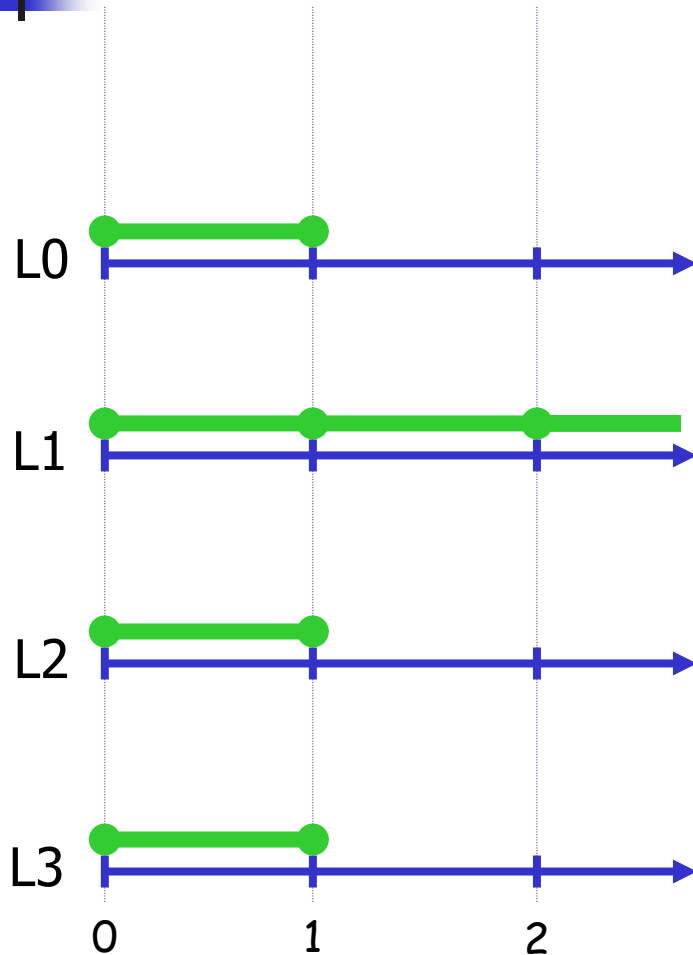
$Waiting \leftarrow Waiting \cup Depend[S]$; $Win[S] \leftarrow Win^*$;

$Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;

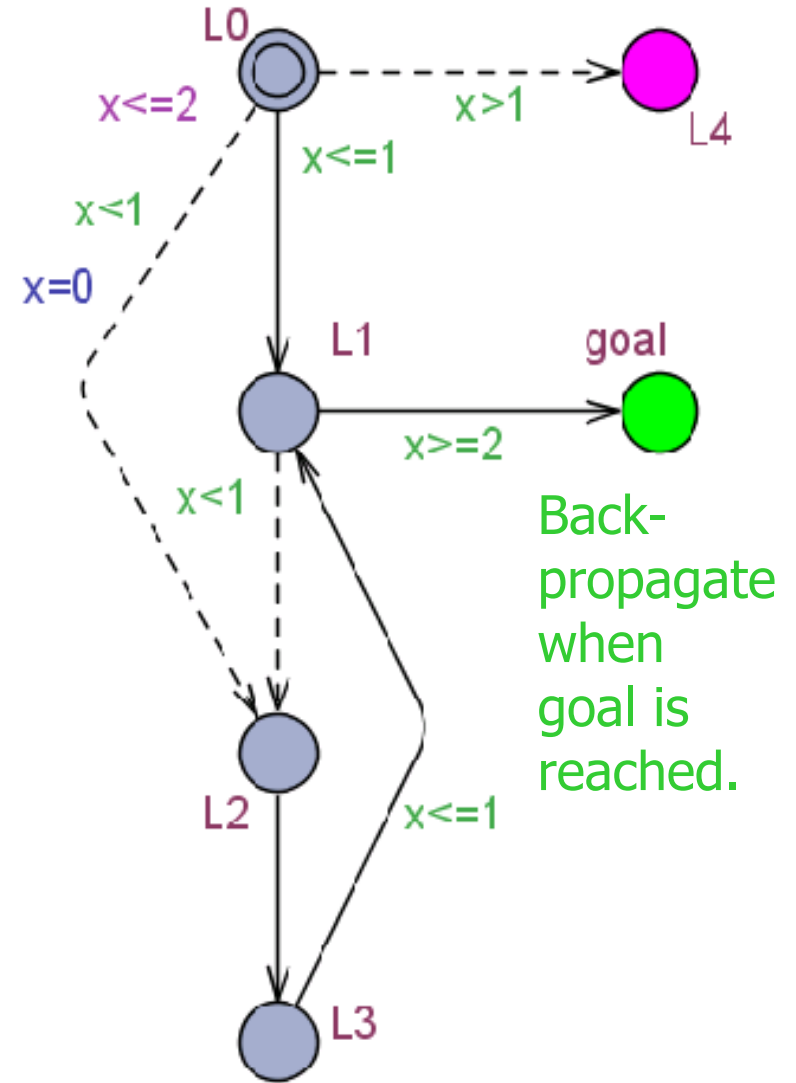
endif

endwhile

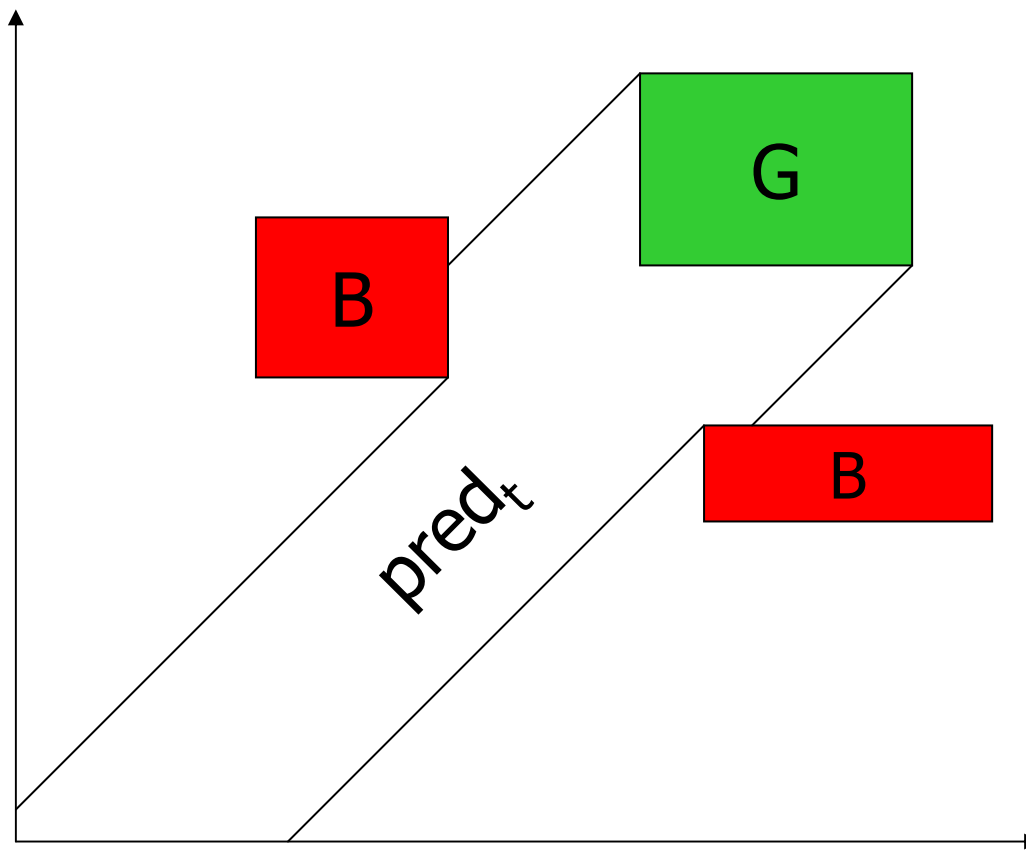
Backward Propagation



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



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

$$pred_t(G, B) = (G^\downarrow \setminus B^\downarrow) \cup ((G \cap B^\downarrow) \setminus B)^\downarrow$$



Query Language (1)

- Reachability properties:

- control: $A[p U q]$
- control: $A \langle \rangle q \Leftrightarrow \text{control: } A[\text{true} U q]$

Back-propagate winning states & BFS+DFS.

- Safety properties:

- control: $A[p W q]$
- control: $A[] p \Leftrightarrow \text{control: } A[p W \text{false}]$

Back-propagate losing states & BFS-BFS.

- Tuning:

- change search ordering,
- add back-propagation of winning+losing states.



Query Language (2)

- Time-optimality

- $\text{control_t}^*(u,g): A[p \cup q]$

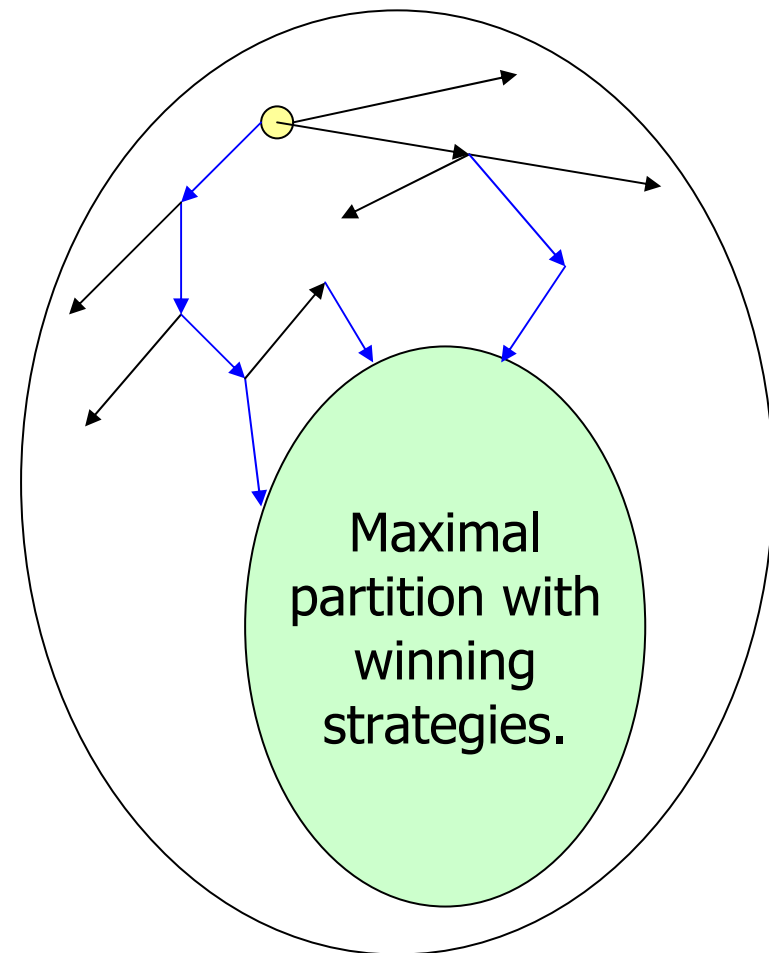
- u is an upper-bound to prune the search, act like an invariant but on the path = expression on the current state.
 - g is the time to the goal from the current state (a lower-bound in fact), also used to prune the search. States with $t+g > u$ are pruned.

- Cooperative strategies.

- $E \langle \rangle$ 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.

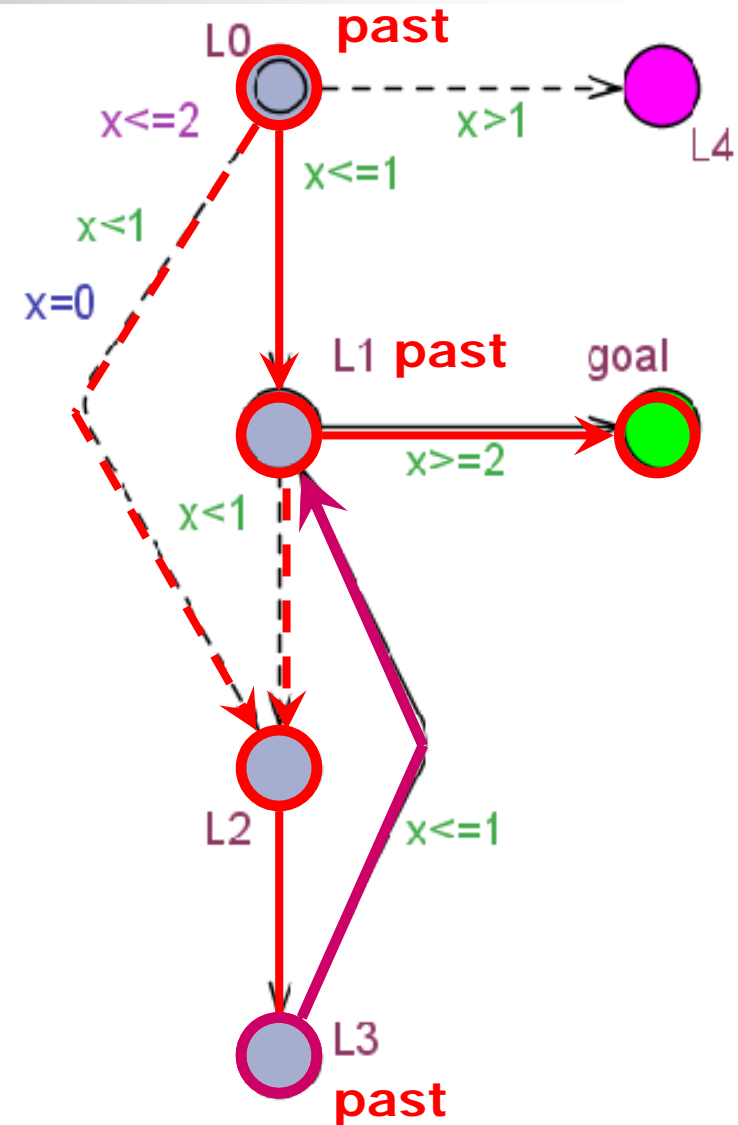
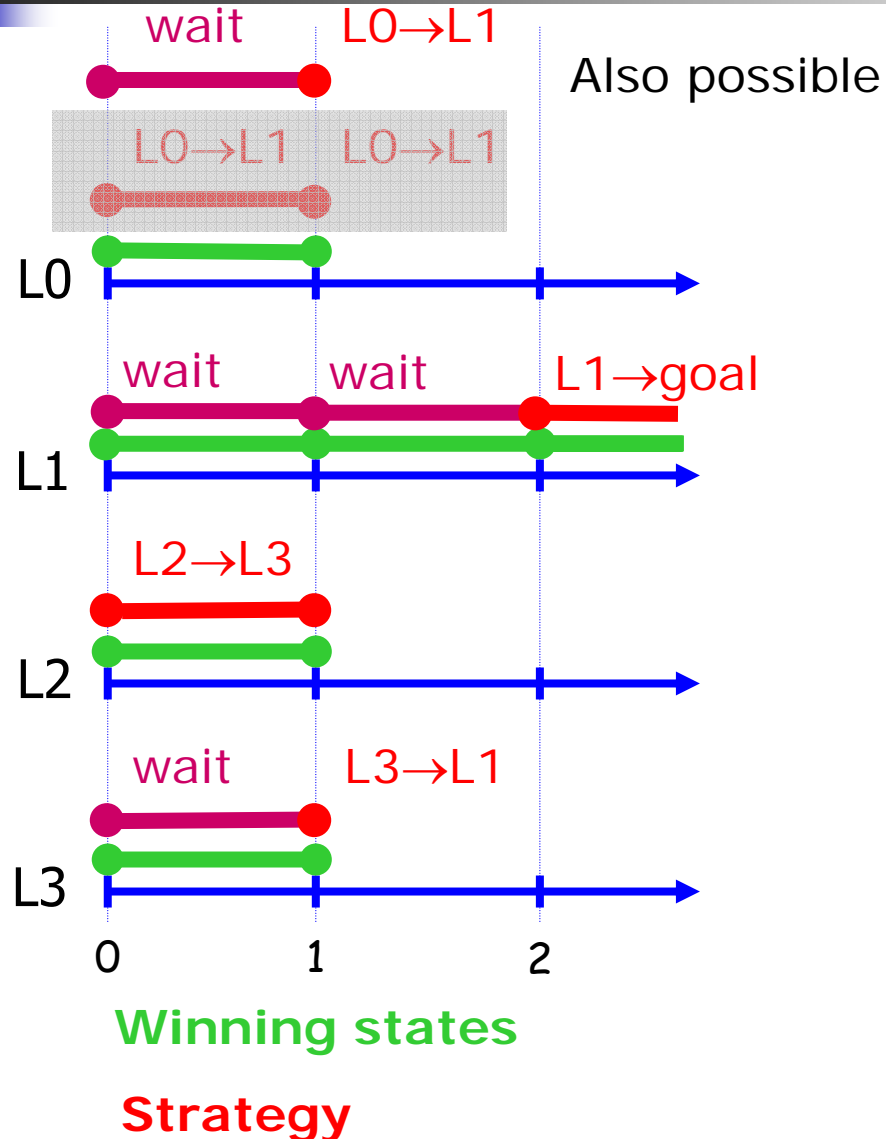




Strategies?

- 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





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.*

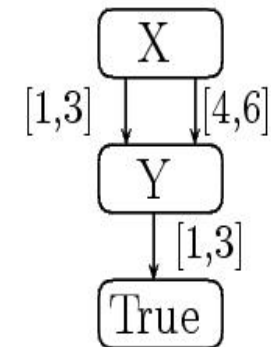
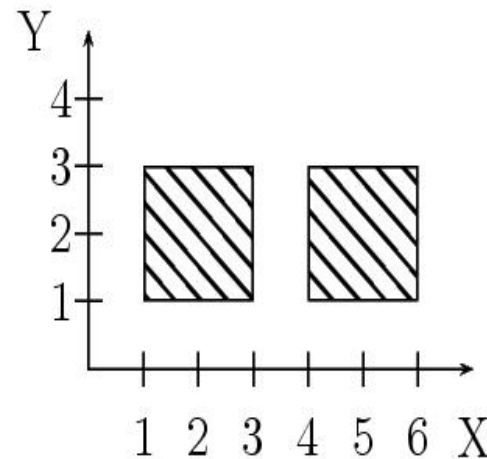
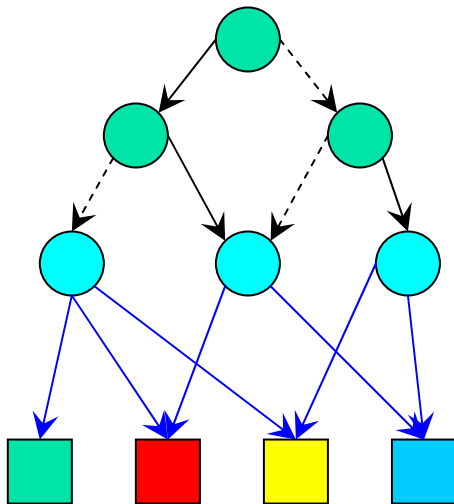
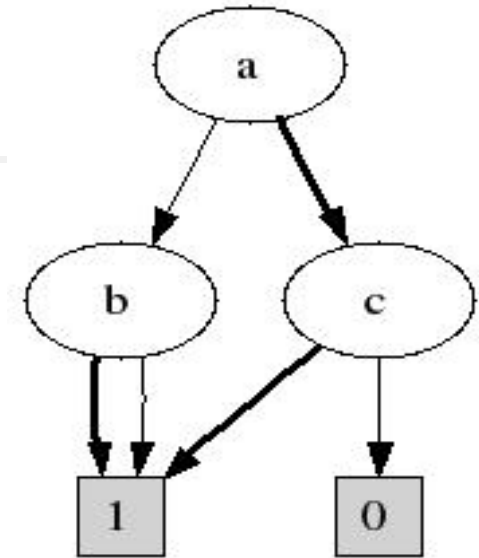


Code Generation

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

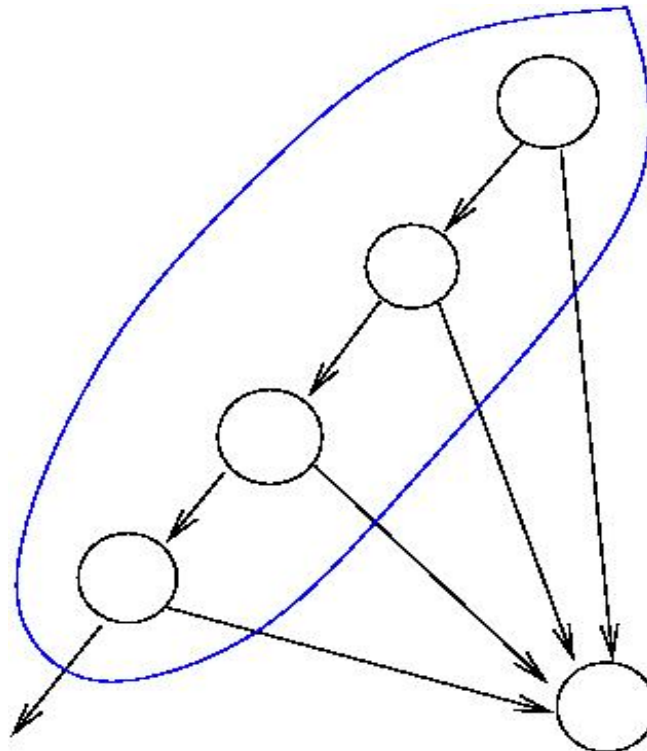
Decision Graph

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

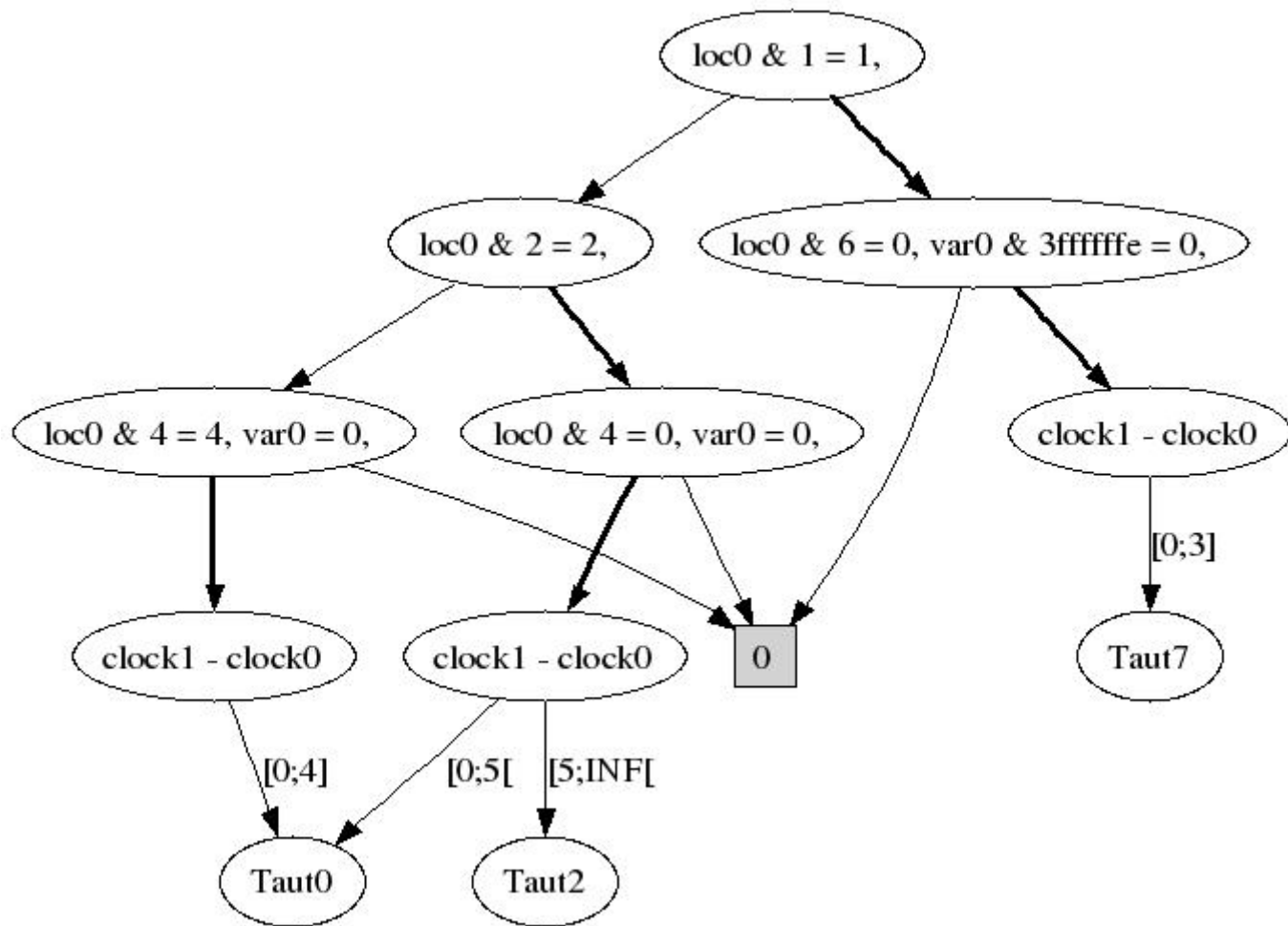


Graph Reduction

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



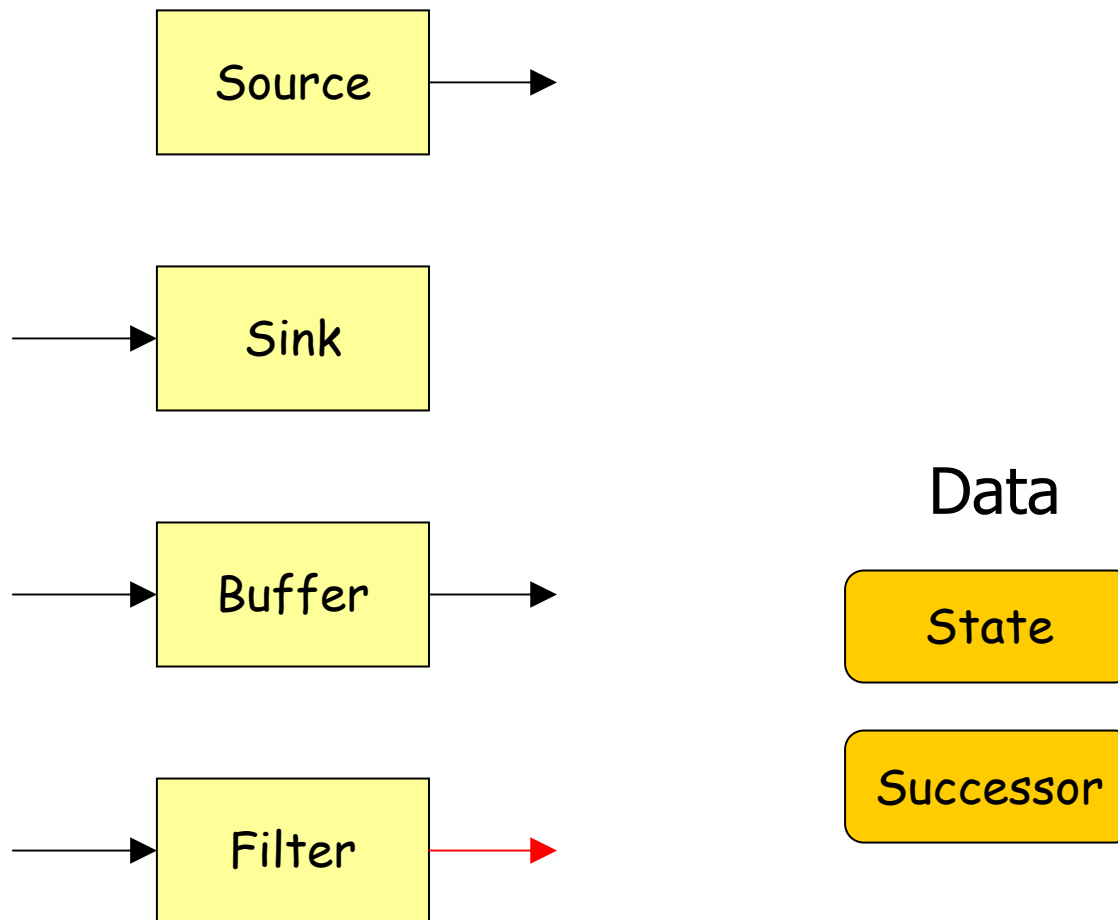
Decision Graph



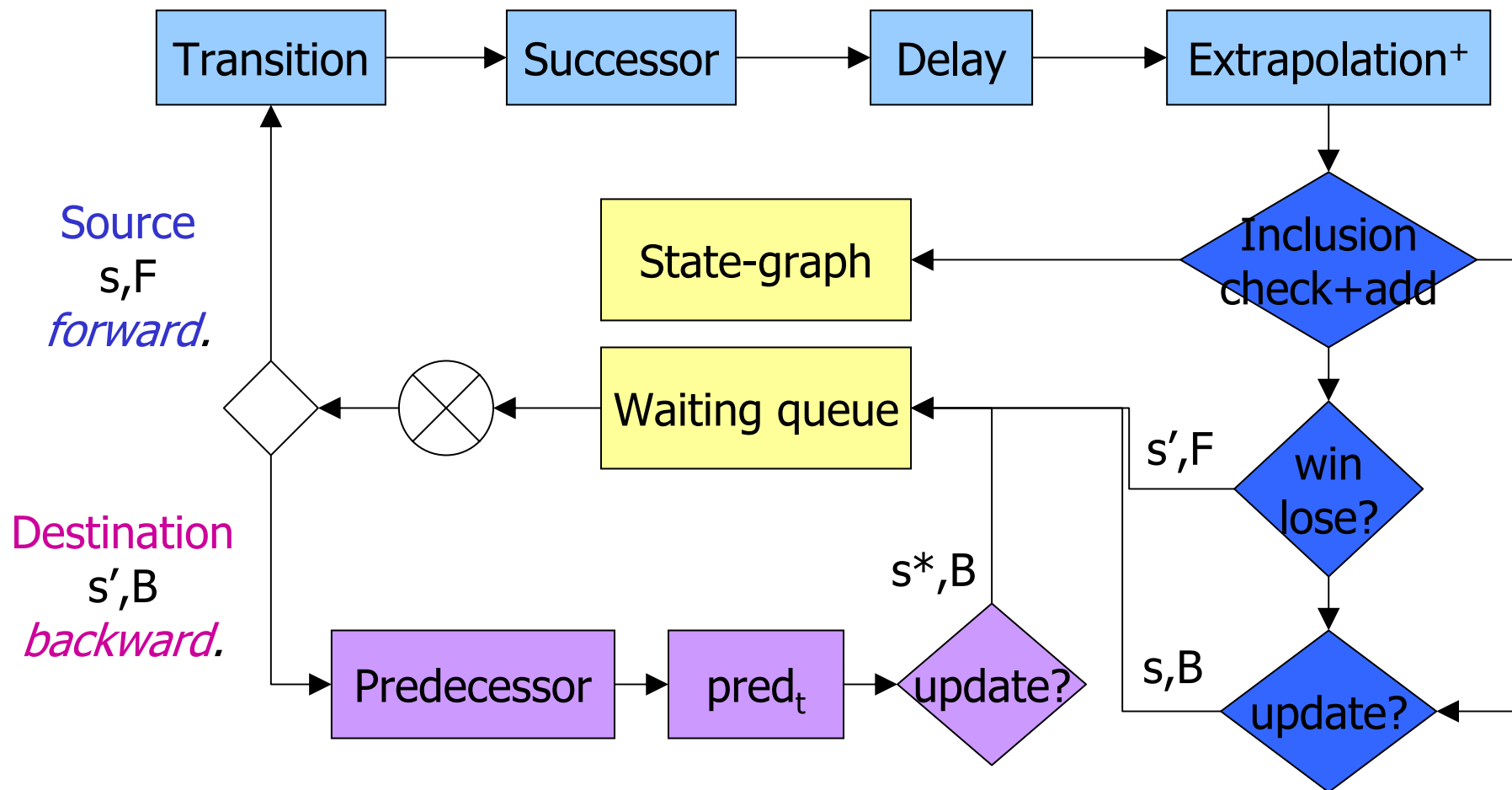


Pipeline Architecture

Pipeline Components



Pipeline Architecture



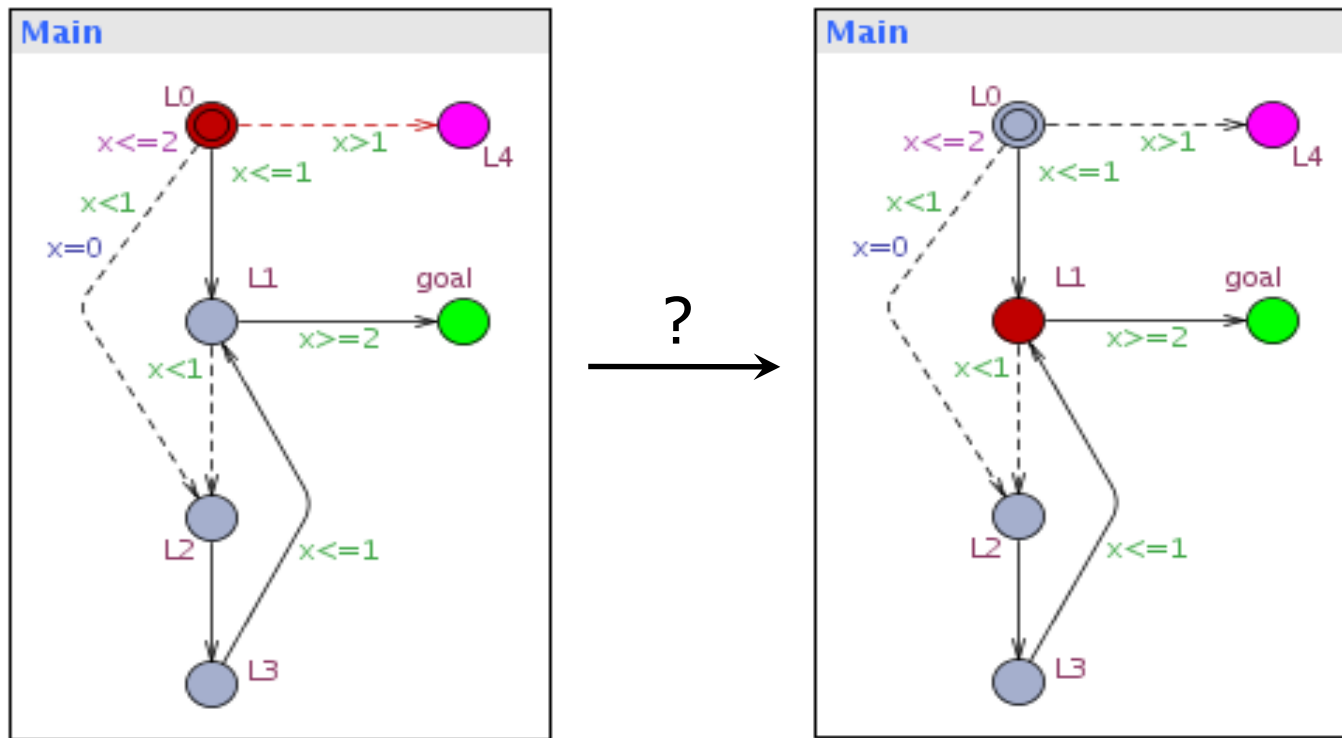


Interactive Game

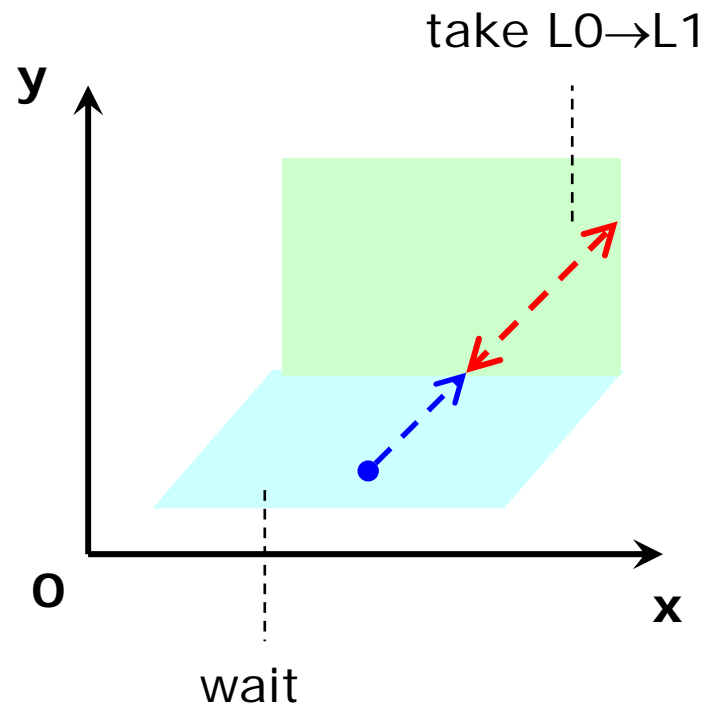
- How to play a (timed) strategy against the user?
 - Concrete simulator.
 - Actions depend on the point in time.
 - Allowed delays depend on the actions.
 - The GUI has limited feedback for showing *counter-actions*.

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

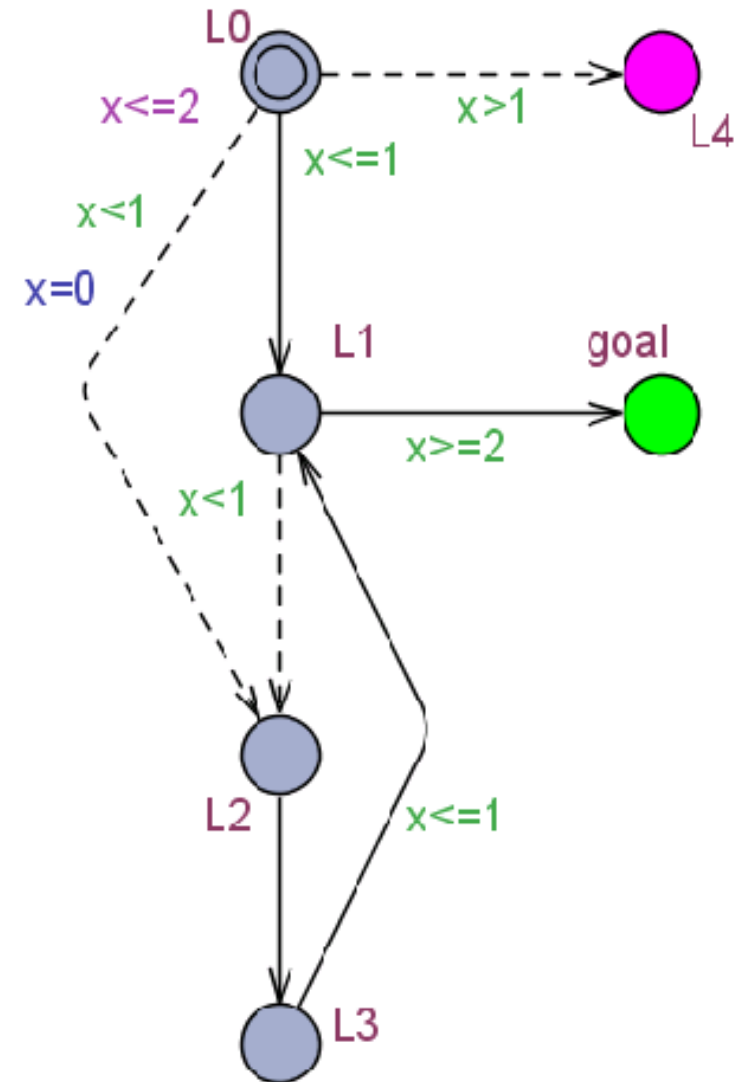
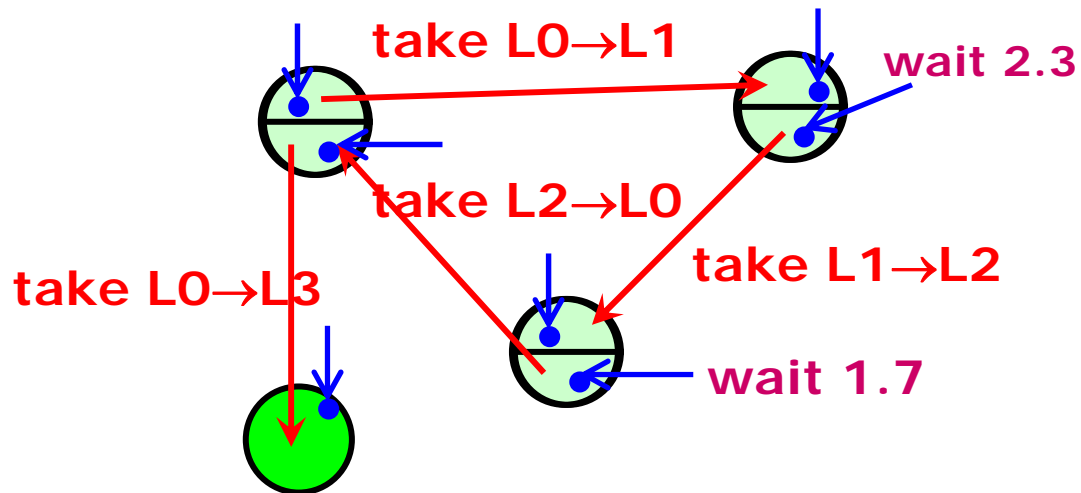


Valid interval for taking action.

How long to wait before taking action.

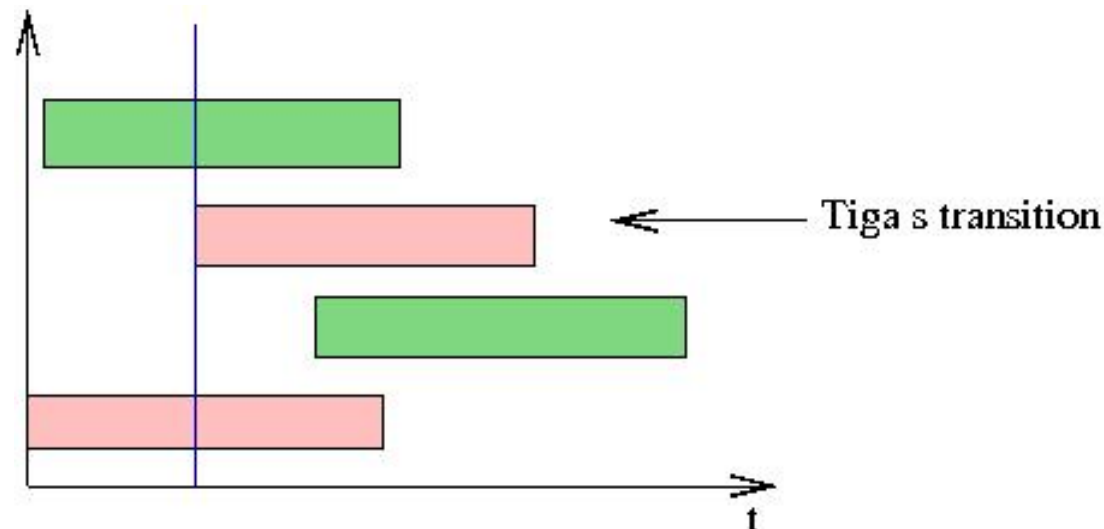
From Symbolic to Concrete

- 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

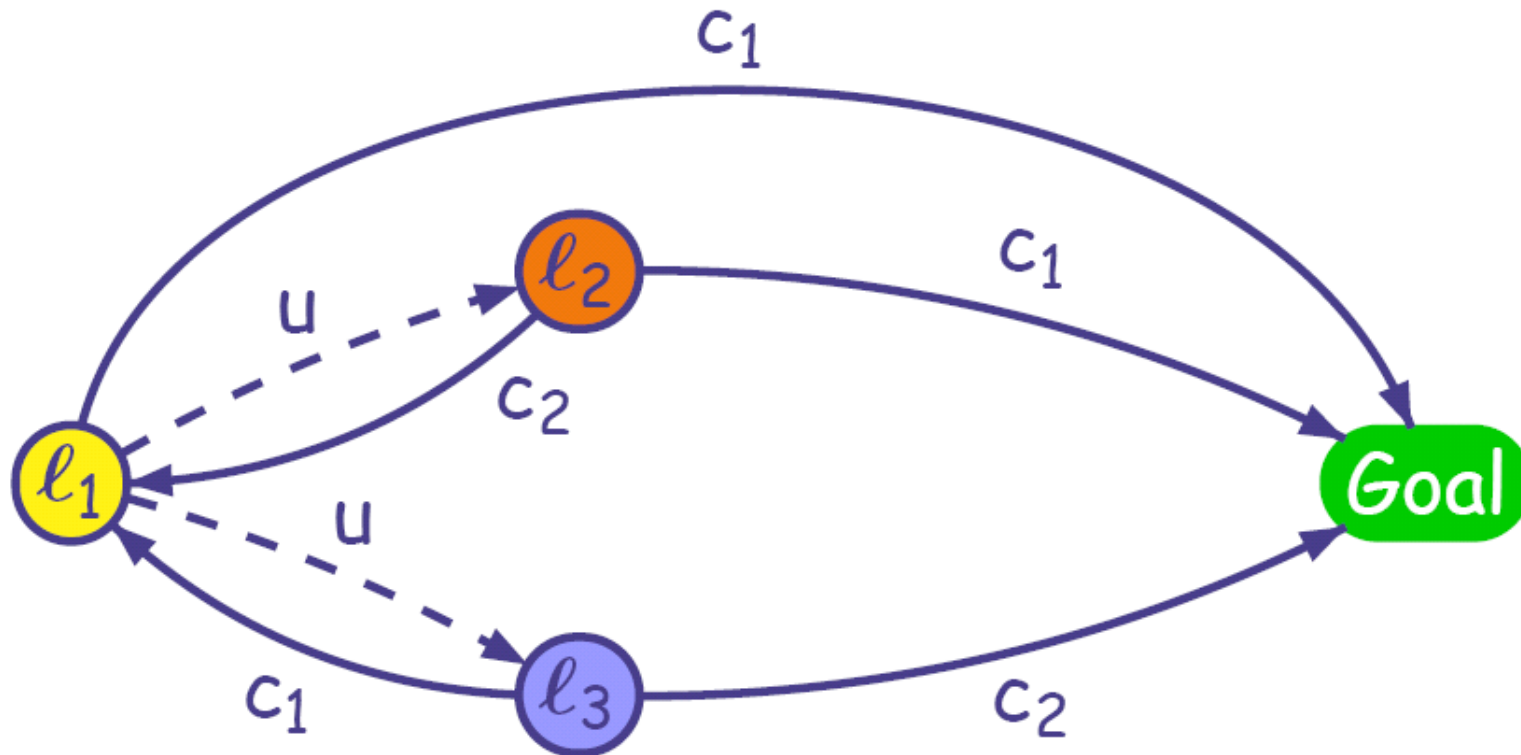
- Previous: Perfect information.
 - Not always suitable for controllers.
- Partial observation.
 - States or events, here states.
 - Distinguish states w.r.t. observations.
 - Strategy keeps track of states w.r.t. observations.
 - Observations = predicates over states.



Results

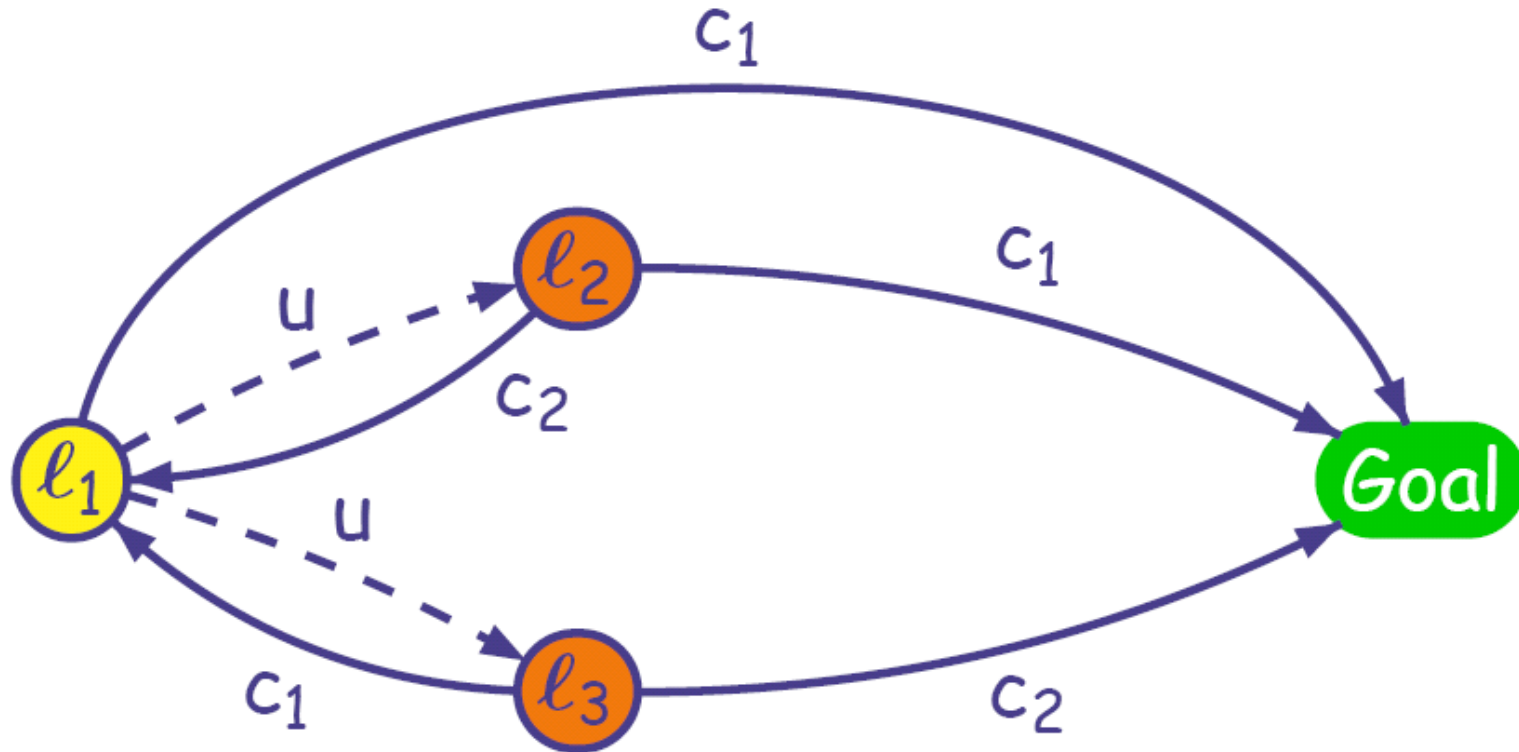
- **Discrete** event systems
 - [Kupferman & Vardi '99, Reif '84, Arnold & al. '03]. Game given as modal logic formula: Full-observation as hard as partial observation.
 - [Chatterjee & al. '06, De Wulf & al. '06]. Game given as explicit graph: Full-observation **P**TIME, partial observation **EX**PTIME.
- **Timed** systems, game given as a TA
 - [Cassez & al. '07] Efficient on-the-fly algorithm, **EX**PTIME.

State Based Full Observation



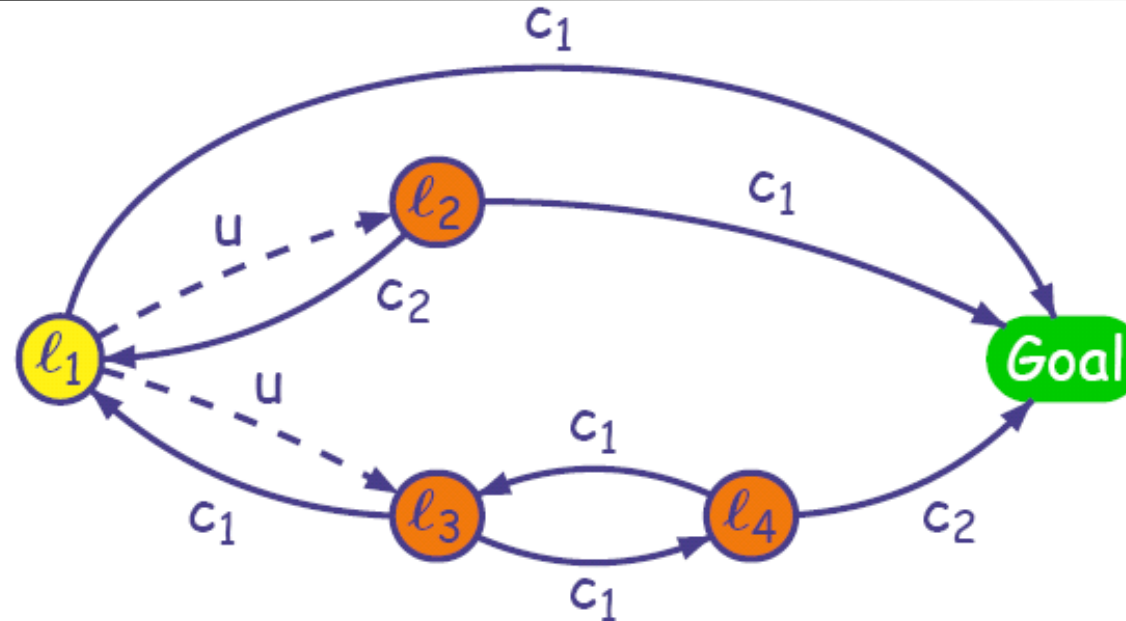
- 2-player reachability game, controllable + uncontrollable actions.
- Full observation: in l_2 do c_1 , in l_3 do c_2 .

State Based Partial Observation



- Partition the state-space $l_2=l_3$.
- Can't win here.

State Based Partial Observation



Winning Strategy:

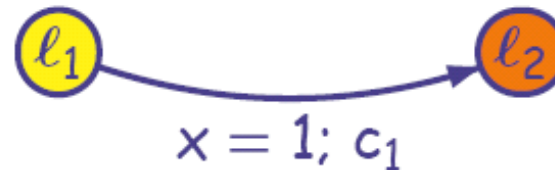
- after: ● play c_1
- after: ● ● play c_1
- after: ● ● ● play c_2

The controller **can observe** each state's change

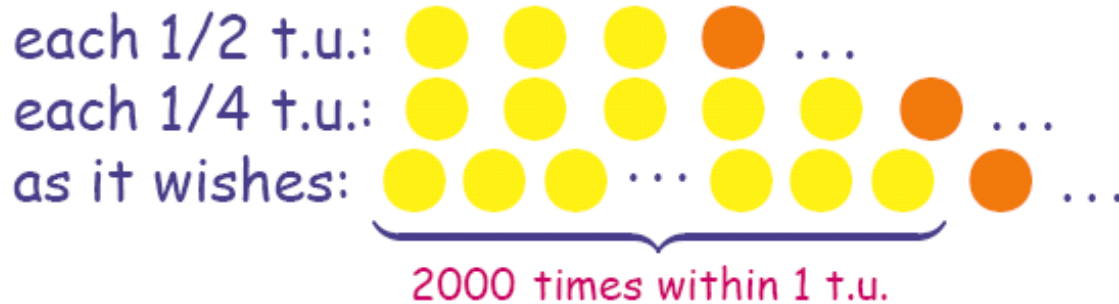
Observation For Timed Systems

In **Continuous** Timed Systems, "next state" is reached by:

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



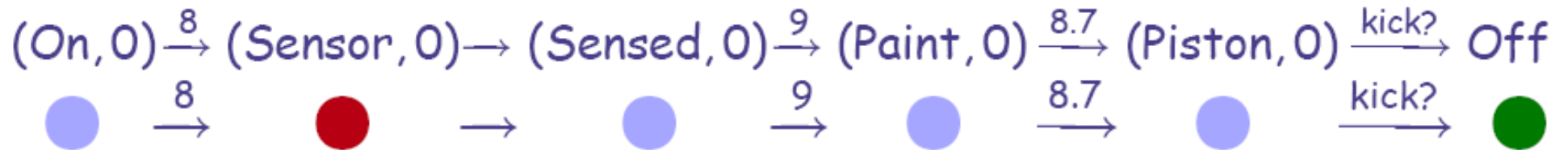
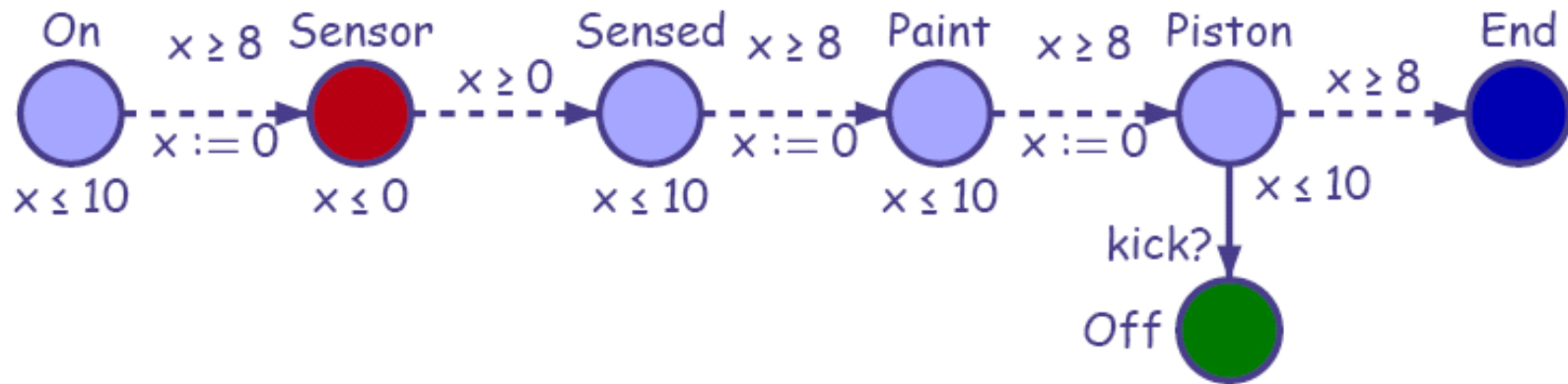
- ▶ Possible Observations:



- ▶ 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

Rules of the Game

Requirements

- ▶ Observations have **special shapes**:
must become true at some first instant / partition the state space
e.g. $20 \leq y < 24$ or $1 \leq x < 2$
- ▶ The control objective Φ is **stuttering closed**

Observation-Based Stuttering Invariant Strategies

f is a **OBSI strategy** if:

$$\text{Observation}(\rho) \equiv_{\text{stutt}} \text{Observation}(\rho') \text{ implies } f(\rho) = f(\rho')$$

Control under Partial Observation Problem

Input: a TGA automaton G
 a finite set of **observations** \mathcal{O} ,
 a **control objective** $\Phi \subseteq \mathcal{O}^\omega$.

Problem: Is there an OBSI **winning** strategy for (G, Φ) ?

On-the-Fly Algorithm

Initialization:

```

Passed ← {{s0}};
Waiting ← {{(s0, α, W') | α ∈ Σ1, o ∈ O, W' = Nextα(s0) ∩ o ∧ W' ≠ ∅}};
Win[{s0}] ← (s0 ⊆ γ(Goal) ? 1 : 0);
Losing[{s0}] ← (s0 ⊈ γ(Goal) ∧ (Waiting = ∅ ∨ ∀α ∈ Σ1, Sinkα(s0) ≠ ∅) ? 1 : 0);
Depend[{s0}] ← ∅;

```

Main:

```

while ((Waiting ≠ ∅) ∧ Win[{s0}] ≠ 1 ∧ Losing[{s0}] ≠ 1) do

```

```

  e = (W, α, W') ← pop(Waiting);

```

```

  if s' ∉ Passed then

```

```

    Passed ← Passed ∪ {W'};

```

```

    Depend[W'] ← {(W, α, W')};

```

```

    Win[W'] ← (W' ⊆ γ(Goal) ? 1 : 0);

```

```

    Losing[W'] ← (W' ⊈ γ(Goal) ∧ Sinkα(W') ≠ ∅ ? 1 : 0);

```

```

    if (Losing[W'] ≠ 1) then (* if losing it is a deadlock state *)

```

```

      NewTrans ← {(W', α, W'') | α ∈ Σ, o ∈ O, W' = Nextα(W) ∩ o ∧ W' ≠ ∅};

```

```

      if NewTrans = ∅ ∧ Win[W'] = 0 then Losing[W'] ← 1;

```

```

      if (Win[W'] ∨ Losing[W']) then Waiting ← Waiting ∪ {e};

```

```

      Waiting ← Waiting ∪ NewTrans;

```

```

    else (* reevaluate *)

```

```

      Win* ← ∨c ∈ Enabled(W) ∧W →c W'' Win[W''];

```

```

      if Win* then

```

```

        Waiting ← Waiting ∪ Depend[W]; Win[W] ← 1;

```

```

        Losing* ← ∧c ∈ Enabled(W) ∨W →c W'' Losing[W''];

```

```

        if Losing* then

```

```

          Waiting ← Waiting ∪ Depend[W]; Losing[W] ← 1;

```

```

          if (Win[W'] = 0 ∧ Losing[W'] = 0) then Depend[W'] ← Depend[W'] ∪ {e};

```

```

        endif

```

```

      endwhile

```

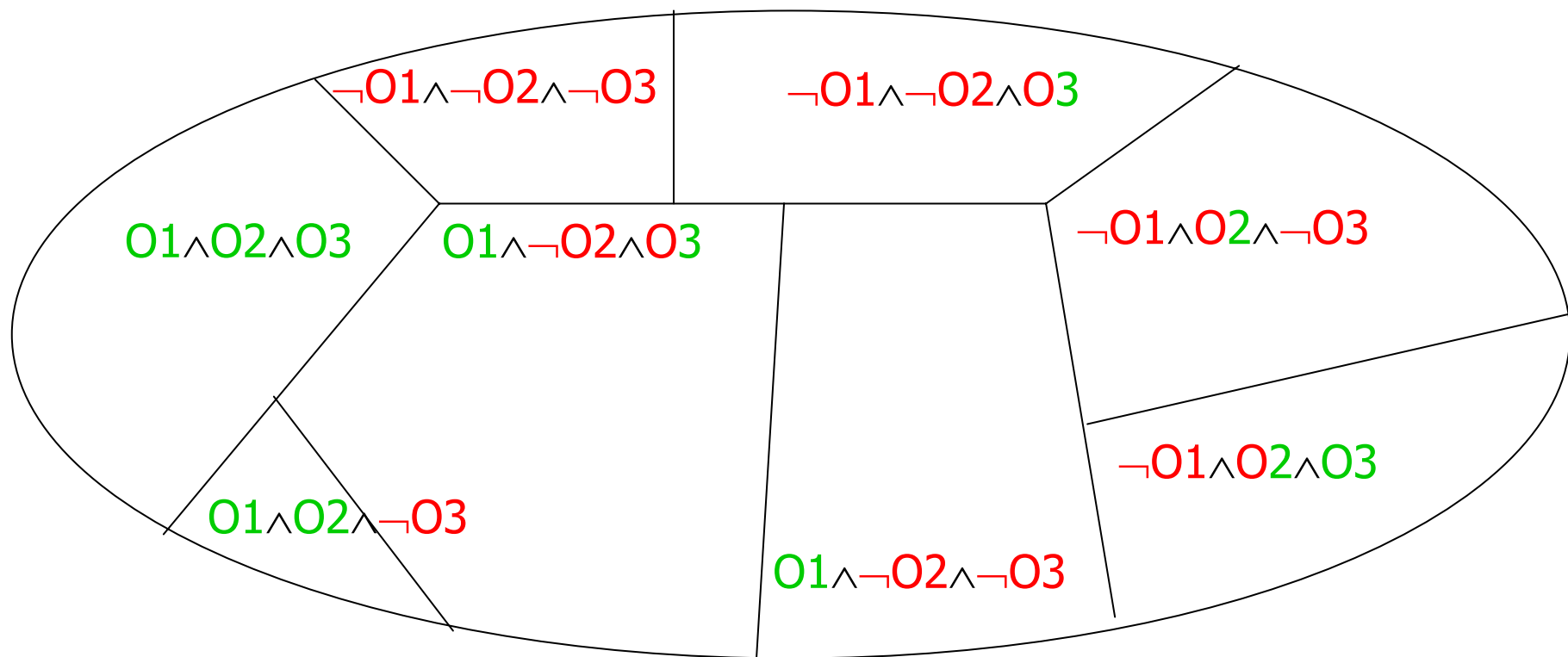


Algorithm

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

Observations $O1$ $O2$ $O3$.

Winning/losing is observable.

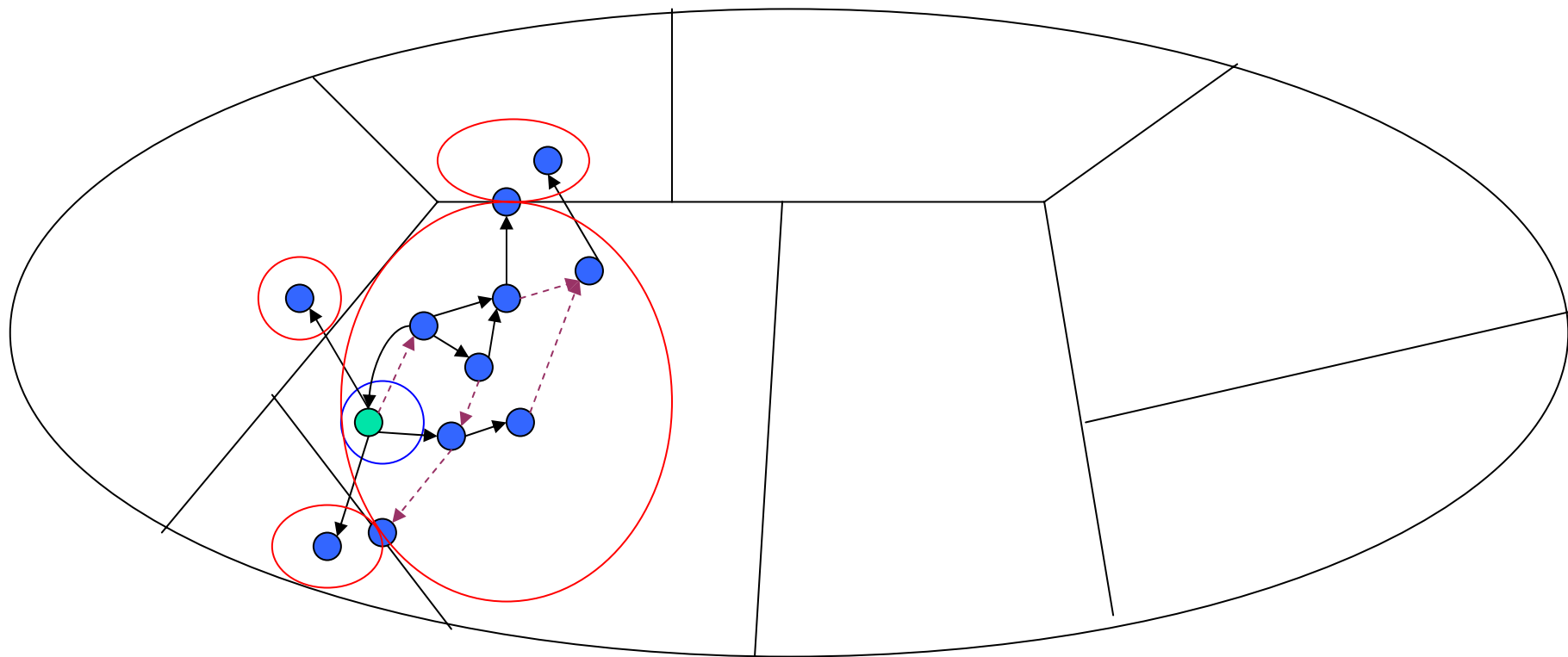


Algorithm

Initial state in some partition.

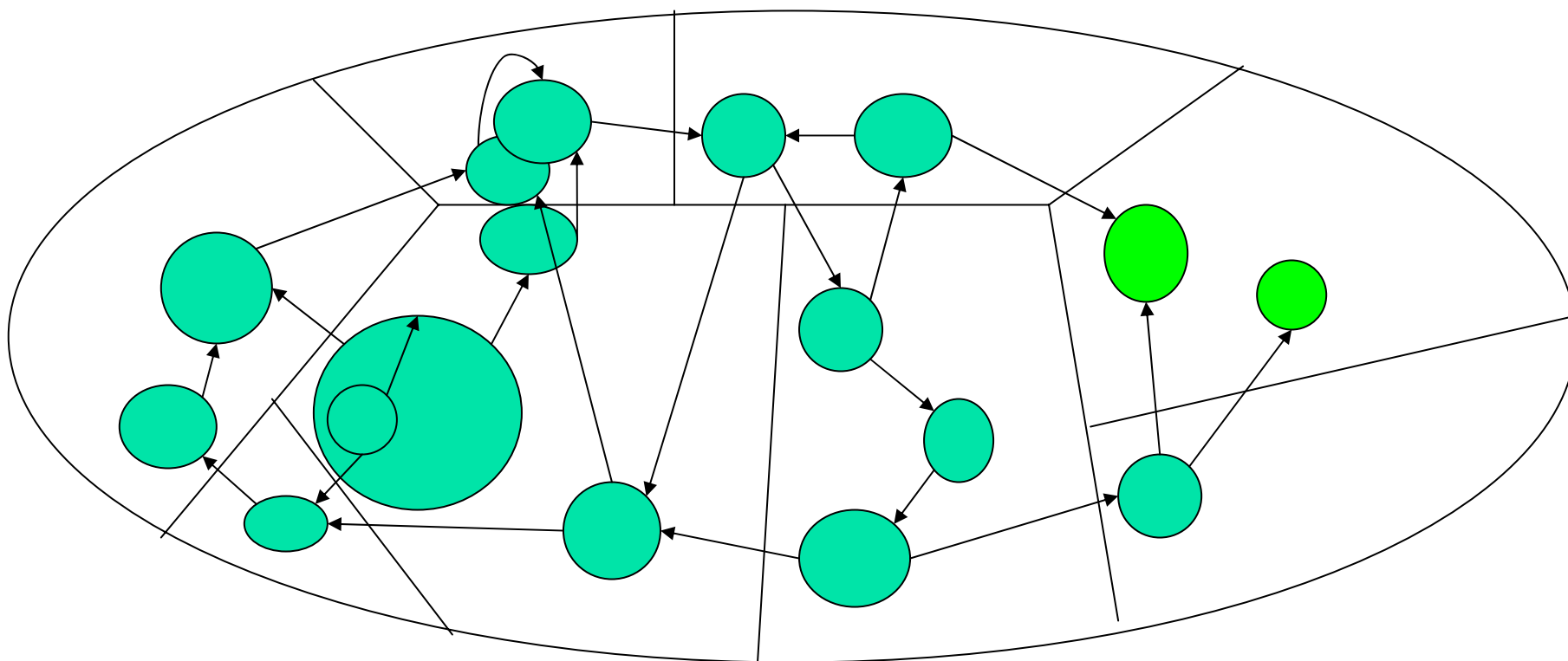
Compute successors { set of states } w.r.t. a controllable action.

Successors distinguished by observations.



Algorithm

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

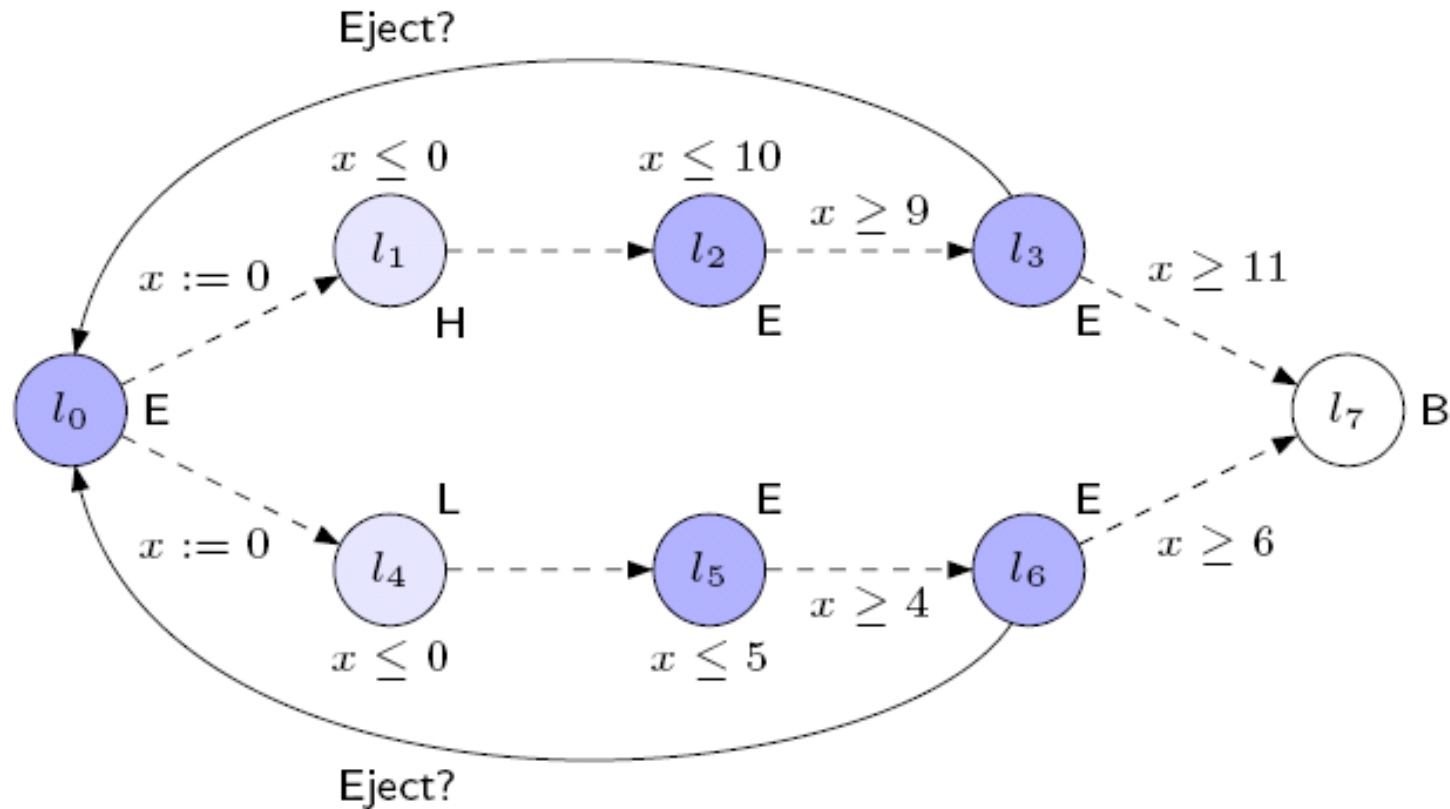




Algorithm

- 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.

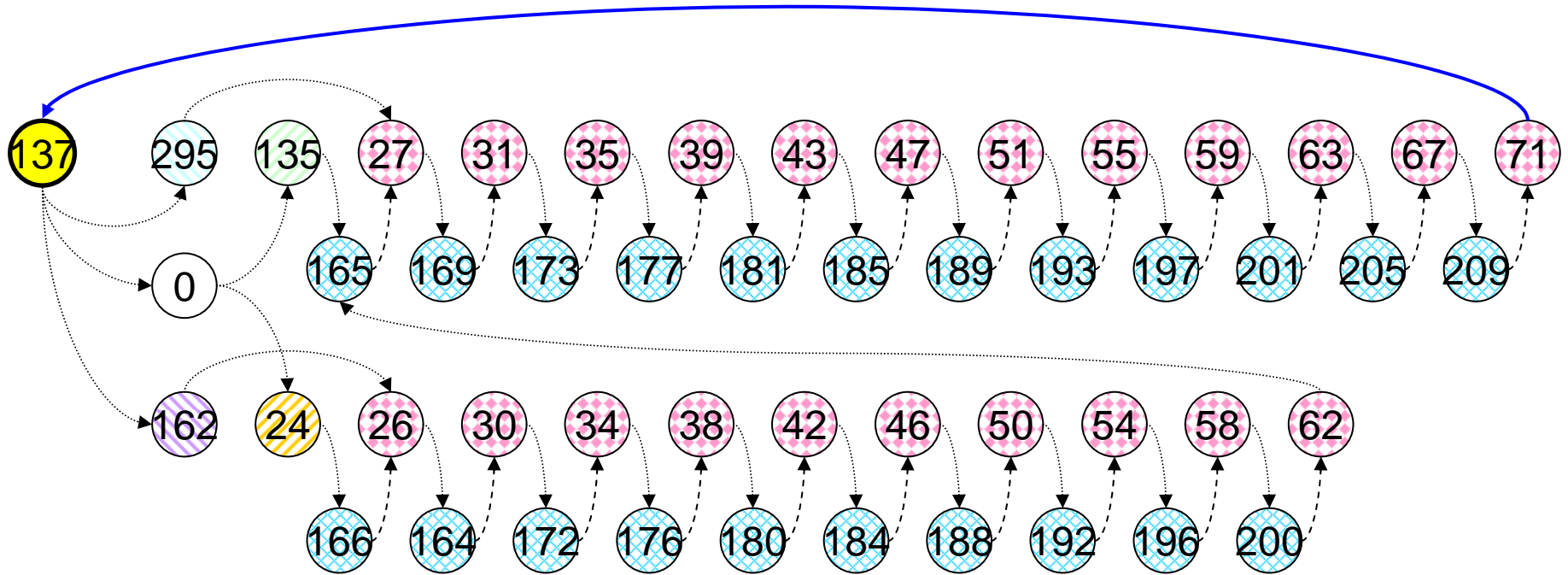
Example



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



Example



Partition:



Actions:

