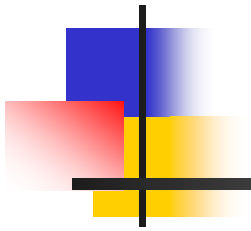
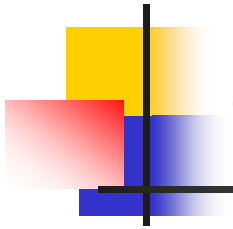


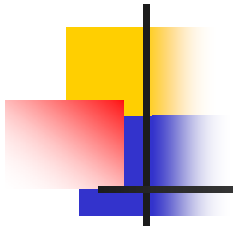
Towards Realizable Strategies

Alexandre David
Kim G. Larsen
Didier Lime



Overview

- Motivation
- Automatic generation of non-zeno strategies
  - More general approach using timed games with Büchi control objectives
- Timed games with partial observability

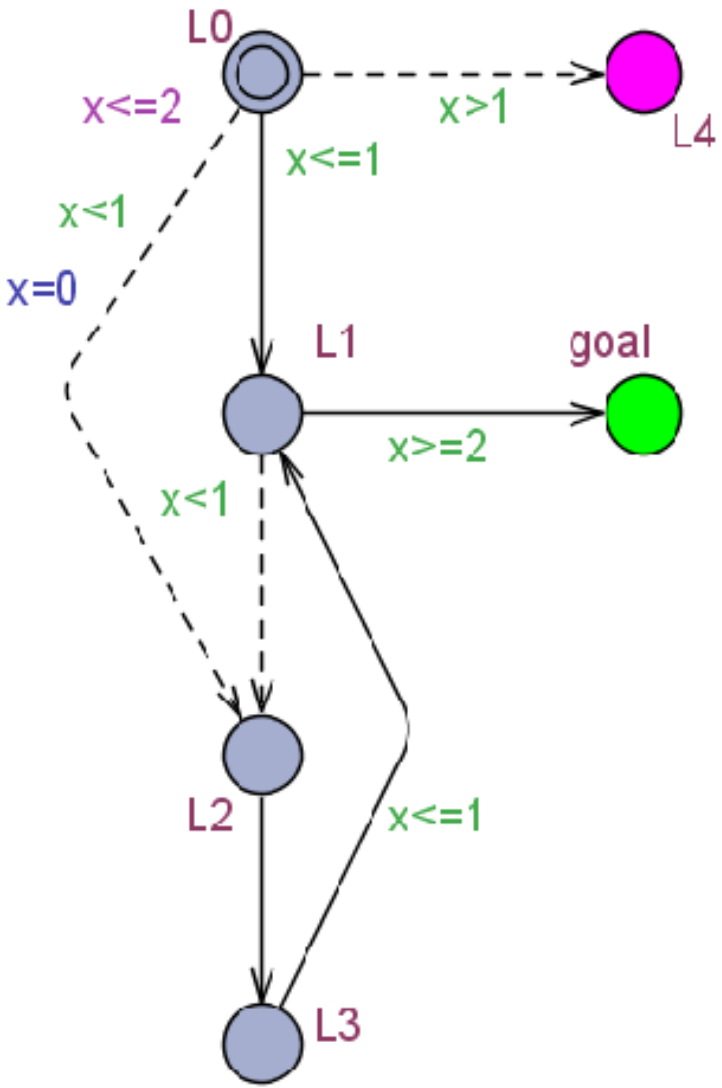


Motivation

- Goal: Generate controllers closer to reality
  - For safety: Good behaviors where time diverges. (Non-zeno controllers).
    - Use a more general approach using games with Büchi control objective.
    - Use an observer to generate automatically non-zeno strategies.
  - Real controllers have partial observability through limited sensors.
    - Take that partial observability into account.
    - ATVA'07

TGA in a Nutshell

- Timed automata with controllable and uncontrollable transitions.
- Reachability & safety games.
  - control: A<> TGA.goal
  - control: A[] not TGA.L4
- Memoryless strategy:
  - state → action.



Strategies in a Nutshell

- control: A<> TGA.goal

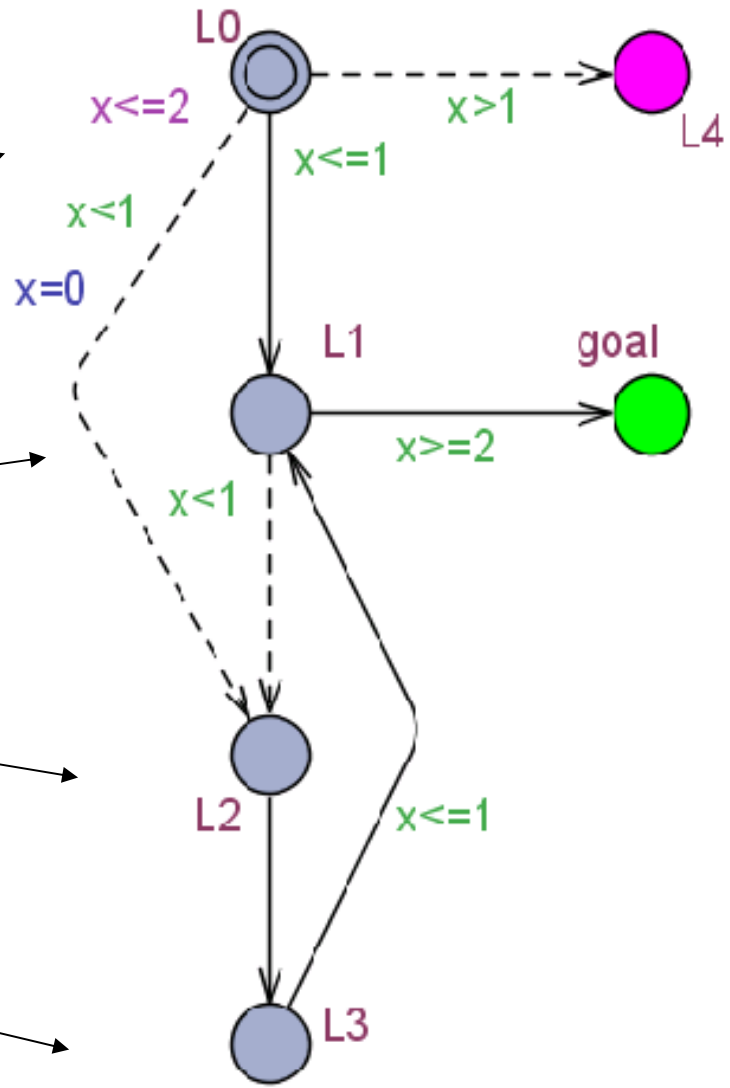
Strategy

x<1	: λ
x==1	: c

x<2	: λ
x≥2	: c

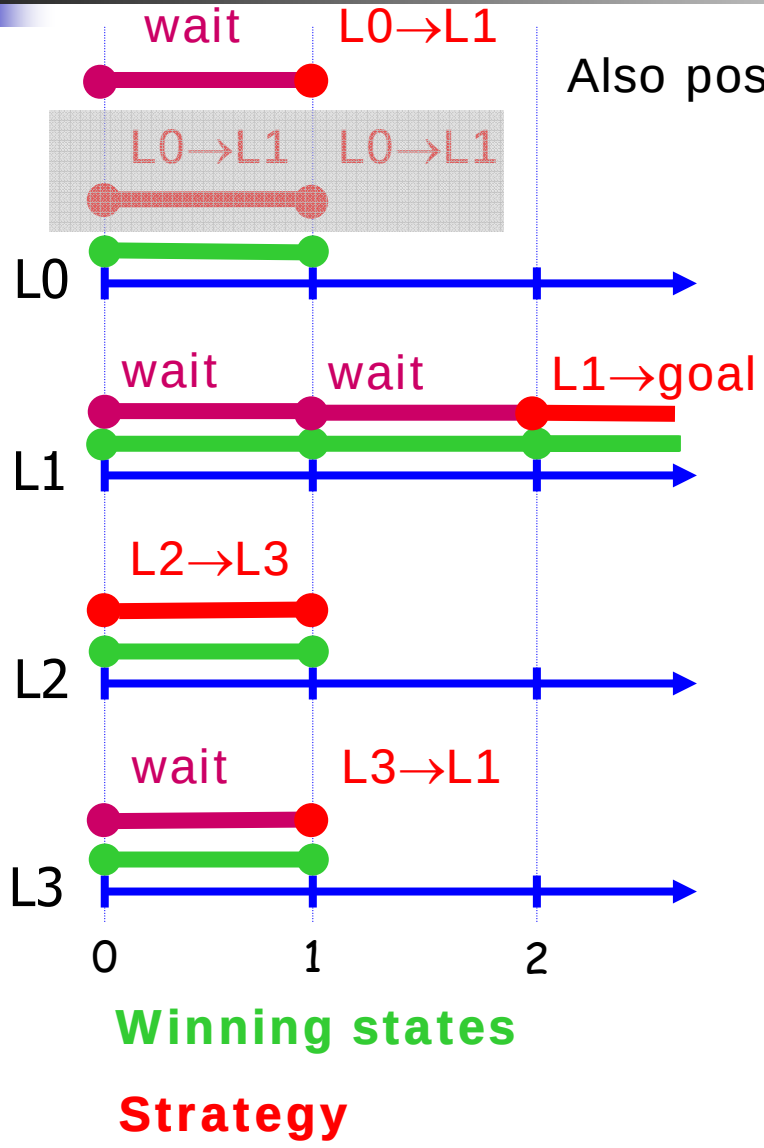
x≤1	: c

x<1	: λ
x==1	: c

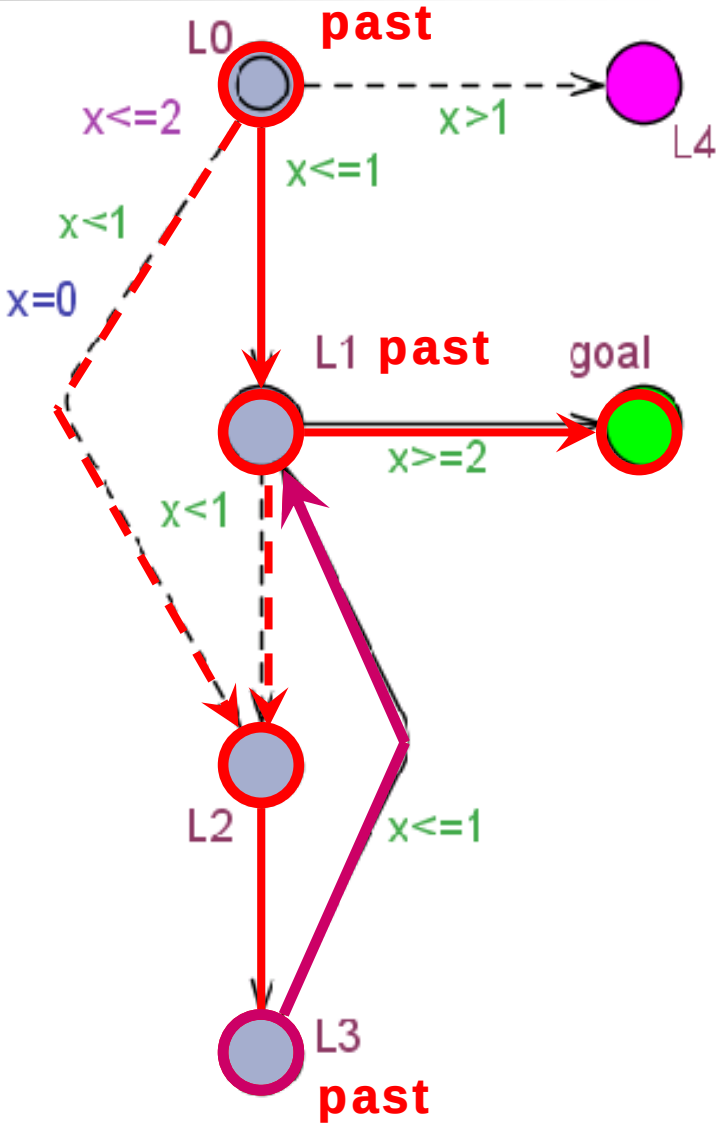


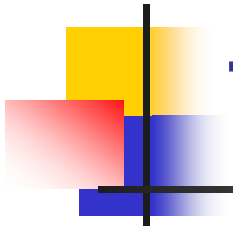
Note: This is *one* strategy.
There are other solutions.

Winning States → Strategy



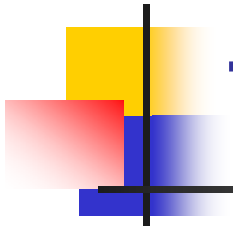
Also possible





TIGA in a Nutshell

- Editor + simulator + verifier – TGA
- Properties:
  - control: A[p U q]
  - control: A<> q ⇔ control: A[true U q]
  - control: A[p W q]
  - control: A[] p ⇔ control: A[p W false]
  - control_t*(u,g): A[p U q]
  - E<> control: φ
  - control: A[] (p && A<> q)
  - control: A[] A<> q ⇔ control: A[] (true && A<> q)
  - { obs1, obs2 … } control: reachability | safety
  - { A, B … } <= { C, D … }



Today

- Games with Buchi accepting states.
  - control: A[] (p && A<> q)
  - control: A[] A<> q
- Partial observable systems:
  - { obs1, obs2 … } control: reachability | safety
  - Generate a controller based on partial observations of the system (obs1, obs2…).
  - [Cassez & al. ’07] Efficient on-the-fly algorithm, EXPTIME.

TGA with Buchi Accepting States

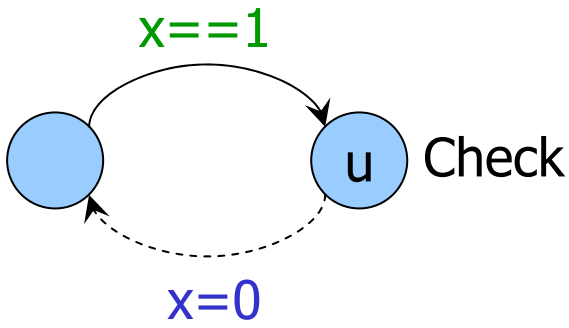
- Use TIGA's algorithm as an intermediate steps in a fixpoint, but modified:
  - winning states are states that can reach goals
  - goal states defined by queries, not necessarily winning
  - Fixpoint:

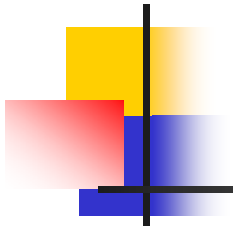
```
Win = SOTF(goal)
while (Win ∩ goal) ≠ goal
    goal = Win ∩ goal
    Win = SOTF(goal)
done
return S0 ∈ Win
```

Simple algorithm. Complexity in line with other known algorithms.

Application: Non-zeno Strategies

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

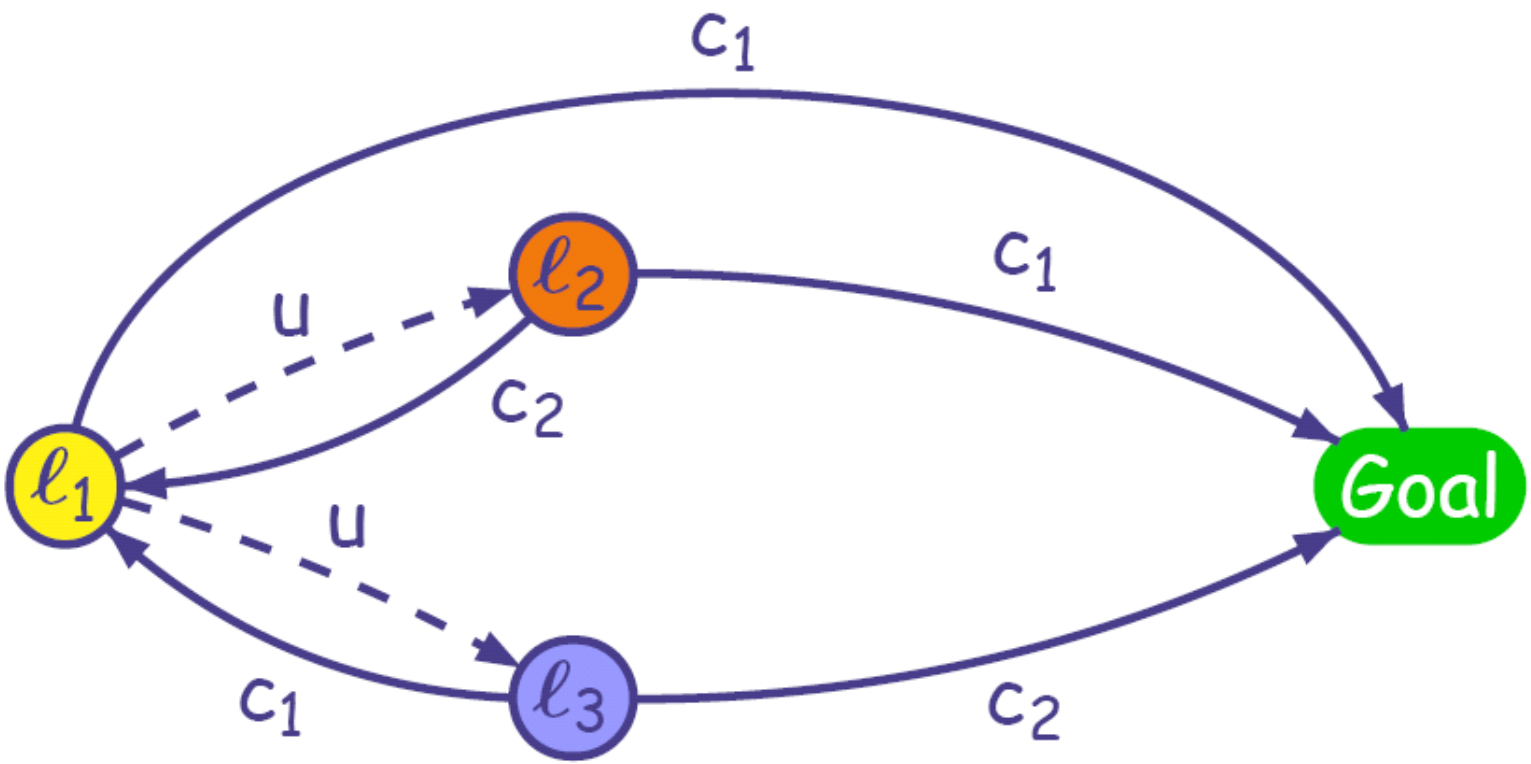




Timed Games *with Partial Observability*

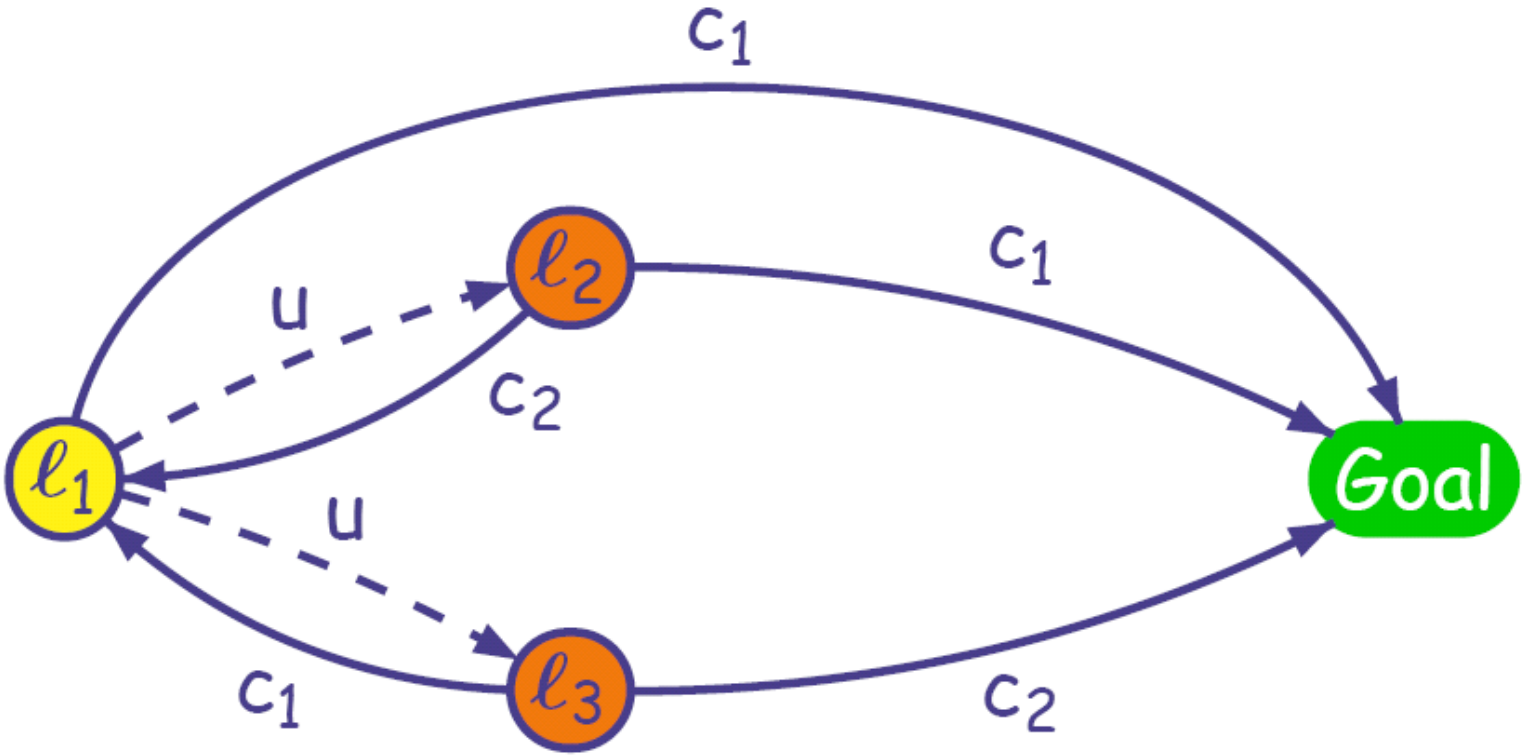
- Previous: Perfect information.
  - Not always suitable for controllers.
- Partial observation.
  - States or events, here states.
  - Distinguish states w.r.t. observations.
  - Strategy keeps track of states w.r.t. observations.
  - Observations = predicates over states.

State Based Full Observation



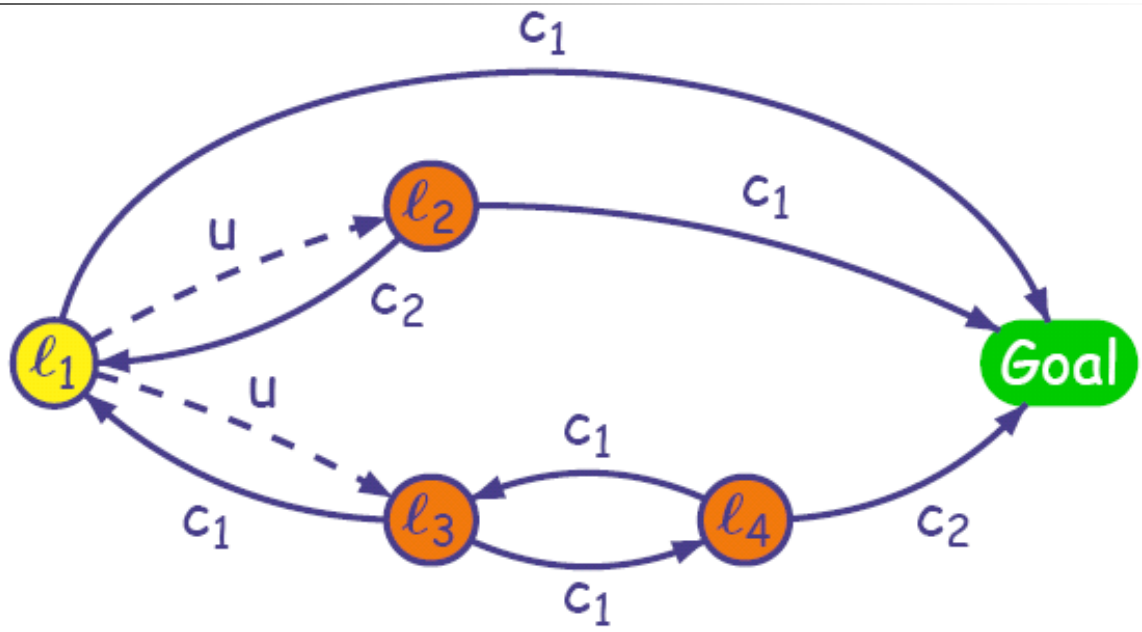
- 2-player reachability game, controllable + uncontrollable actions.
- Full observation: in l_2 do c_1, in l_3 do c_2.

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

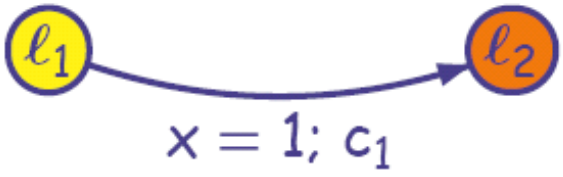
after: play c_2

The controller can observe each state's change

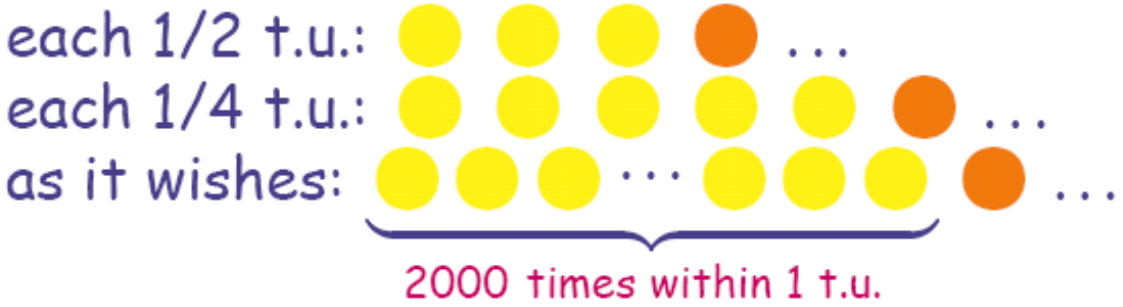
Observation For Timed Systems

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

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



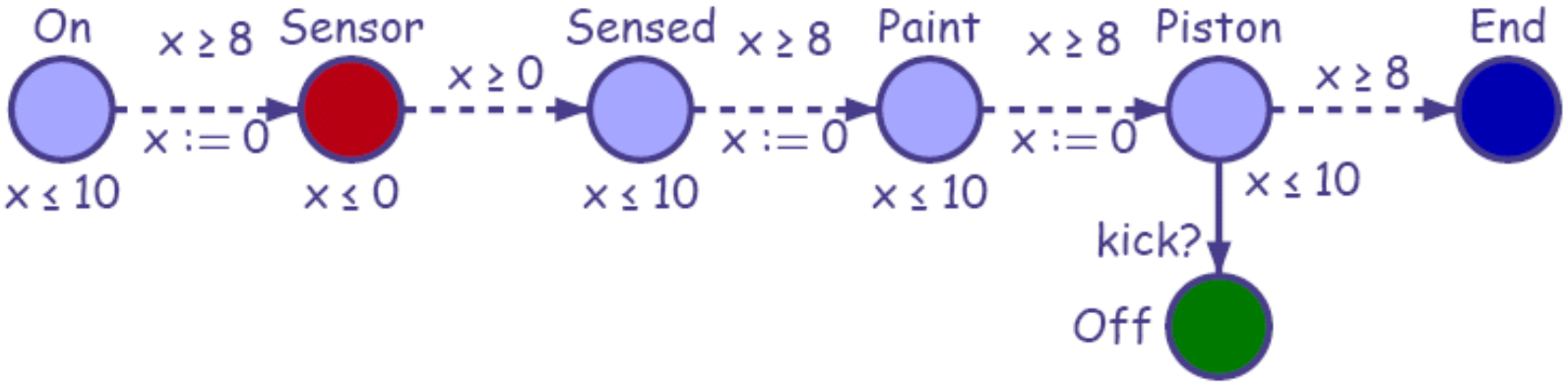
- Possible Observations:



- the controller cannot observe each state's change

Issue: When does the controller observe the system ?

Stuttering-Free Invariant Observations

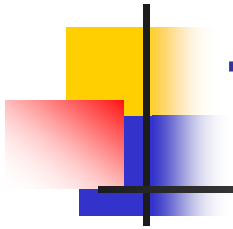


$(On, 0) \xrightarrow{8} (Sensor, 0) \rightarrow (Sensed, 0) \xrightarrow{9} (Paint, 0) \xrightarrow{8.7} (Piston, 0) \xrightarrow{kick?} Off$

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:

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

Main:

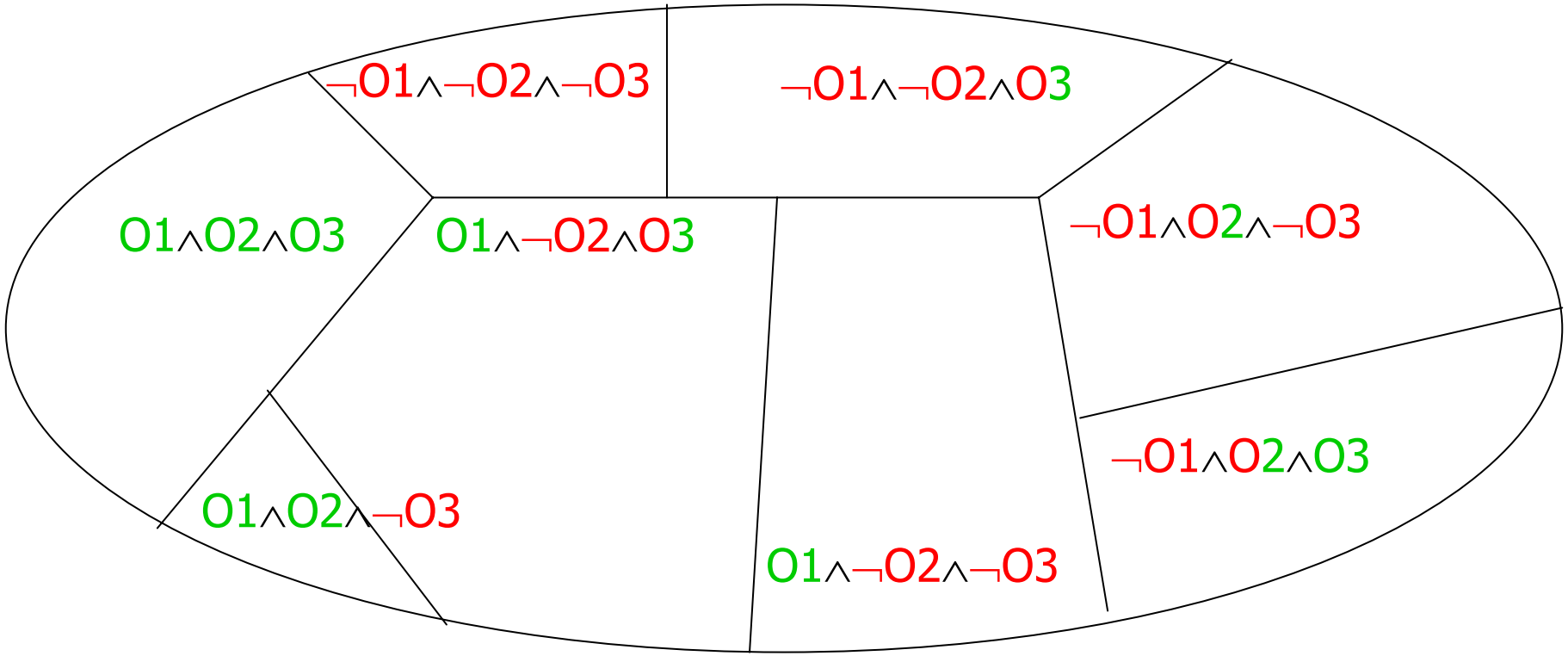
while $((Waiting \neq \emptyset) \wedge Win[\{s_0\}] \neq 1 \wedge Losing[\{s_0\}] \neq 1))$ **do**
  $e = (W, \alpha, W') \leftarrow pop(Waiting)$;
  if $s' \notin Passed$ **then**
    $Passed \leftarrow Passed \cup \{W'\}$;
    $Depend[W'] \leftarrow \{(W, \alpha, W')\}$;
    $Win[W'] \leftarrow (W'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

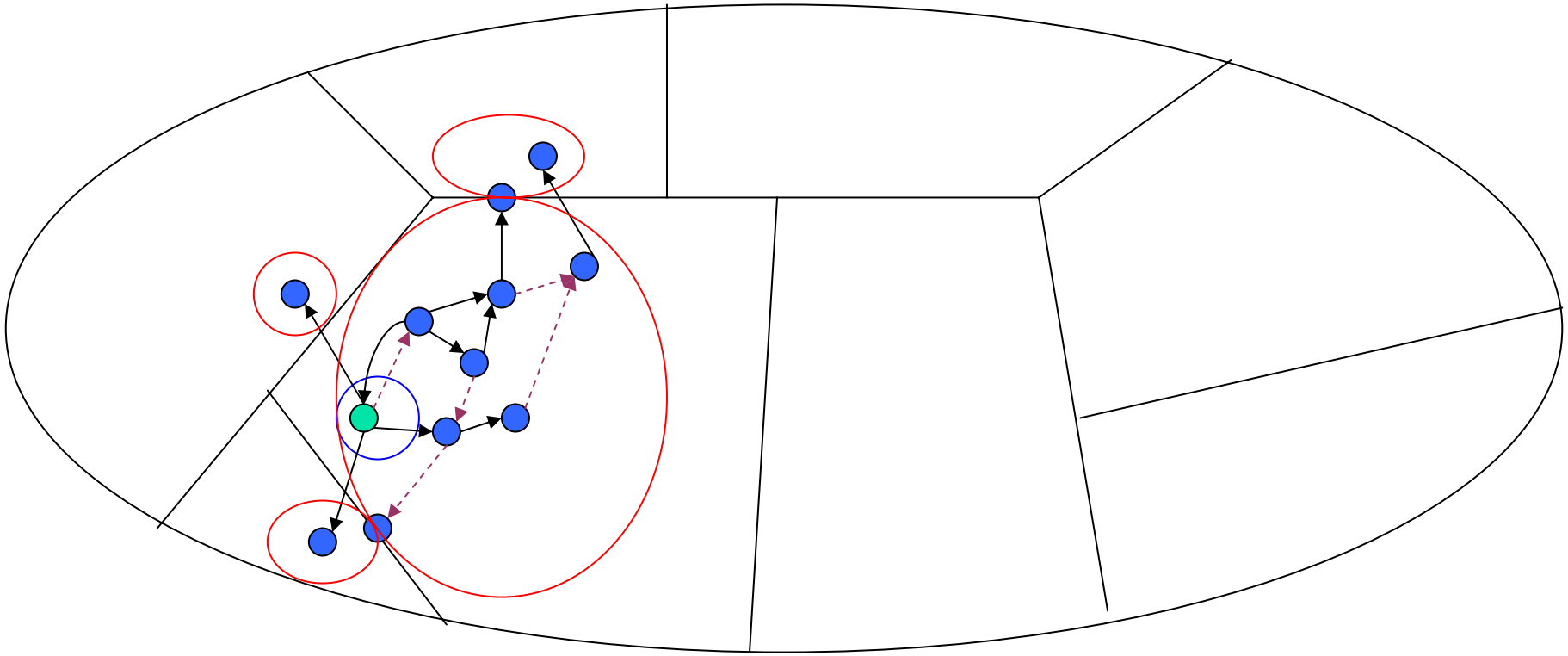
Observations O1 O2 O3.

Winning/losing is observable.



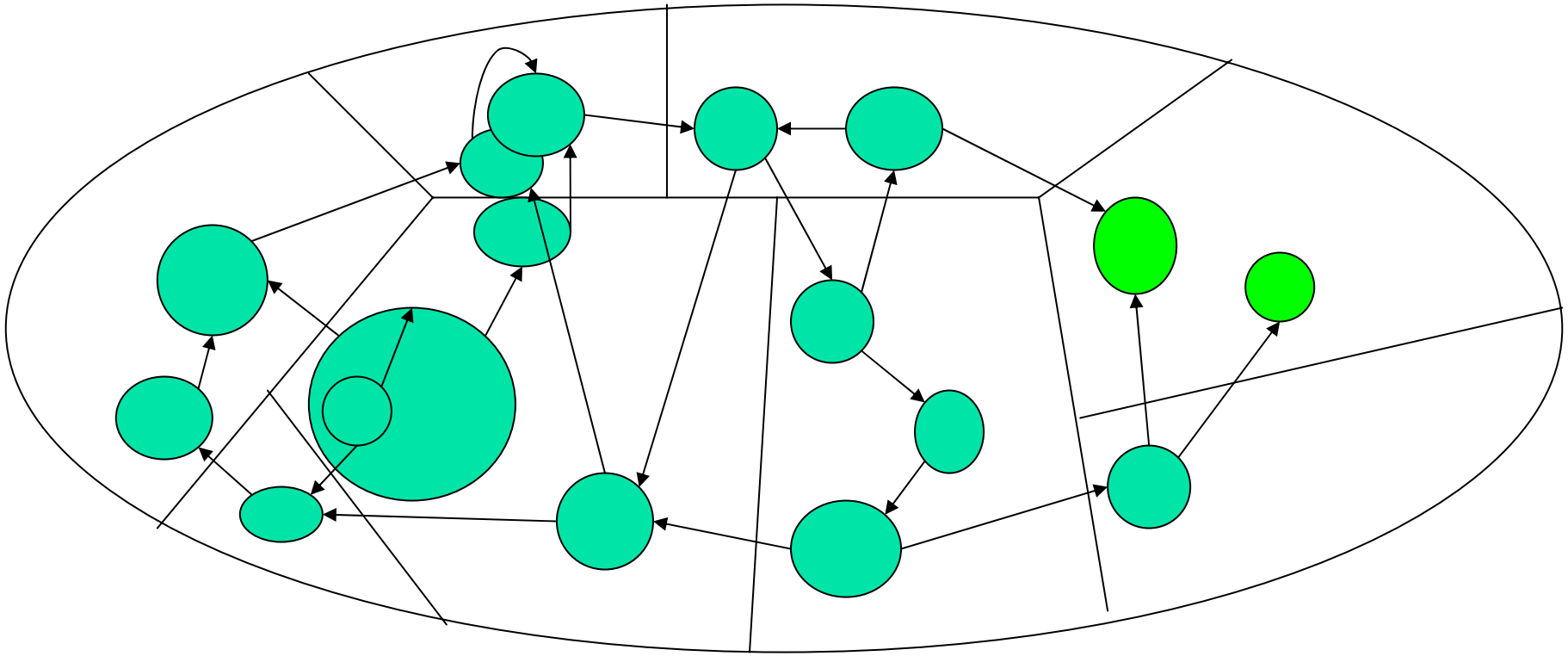
Algorithm

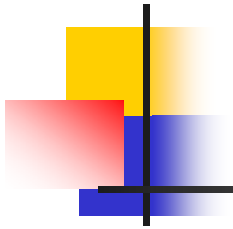
Initial state in some partition.
Compute successors { set of states } w.r.t. a controllable action.
Successors distinguished by observations.



Algorithm

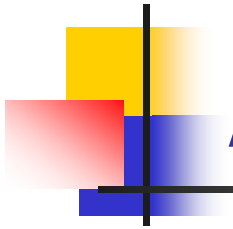
Construct the graph of sets of symbolic states.
Back-propagate winning/losing states.





Notes

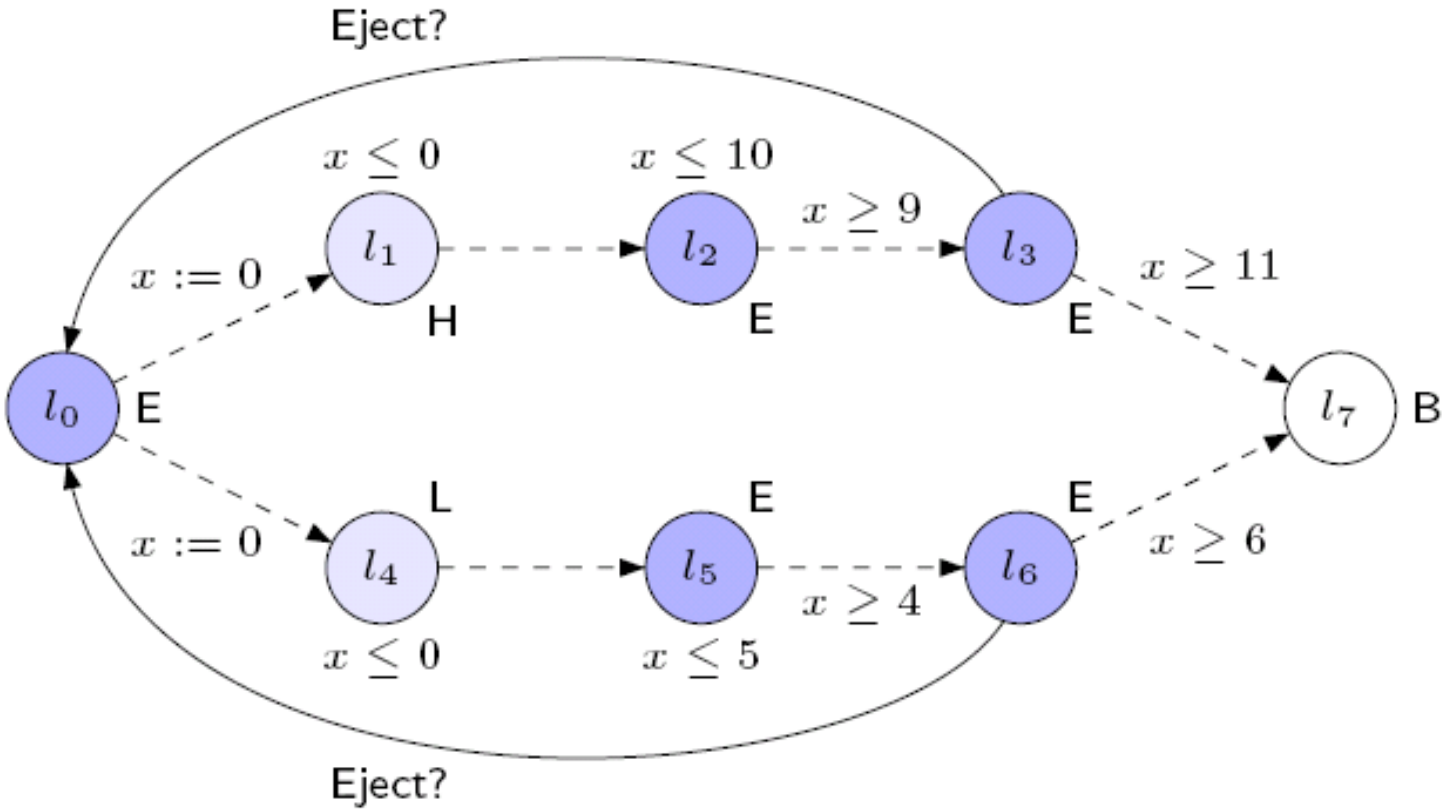
- Stuttering-free invariant observations
  - sinks possible in observations (deadlock/livelock/loop)
- Actions are urgent
  - delay until actions are enabled (or observation changes)



Algorithm

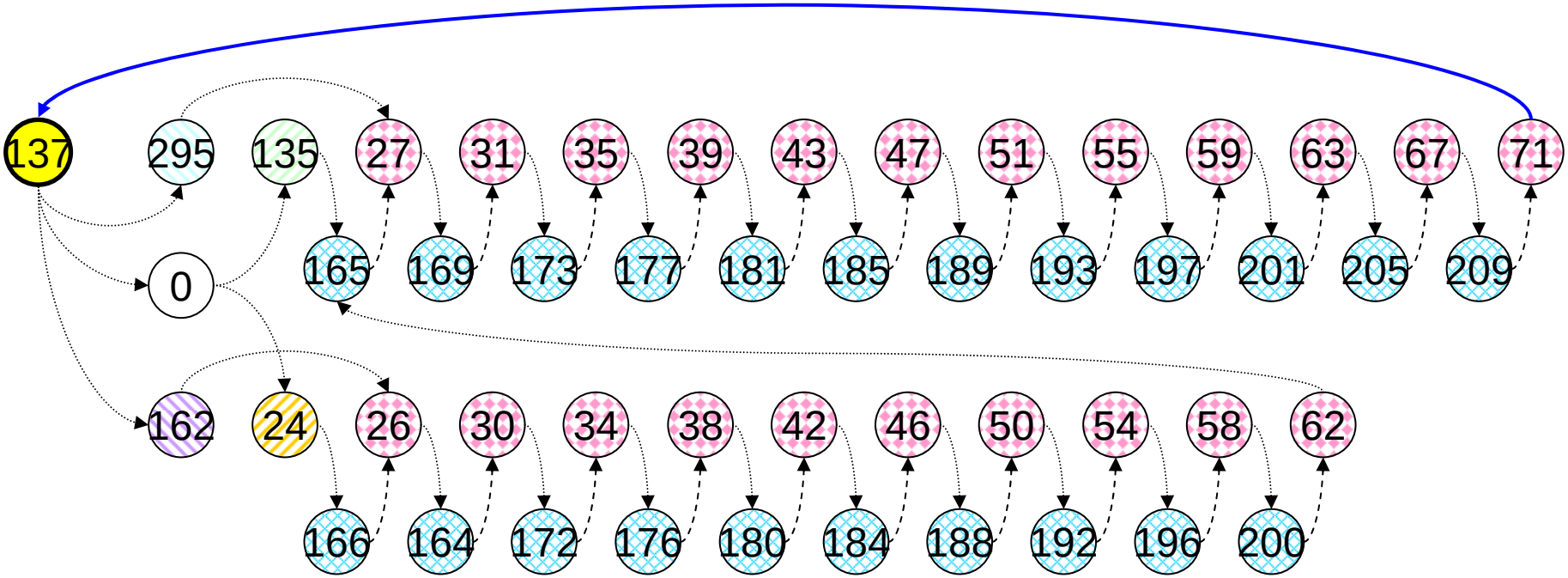
- Forward exploration
  - constrained by action + observations
  - delay special
- Back-propagation.
  - If all successors^a are winning,
    declare current state winning,
    strategy: take action *a*.
  - If one successor^a is losing,
    avoid action *a*.
    If no action is winning the current state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

Example



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

Example



Partition:

Actions:

delay

y=0

eject!