

# Controller Synthesis Based on On-Demand Partial Observations



---

Franck Cassez  
Alexandre David  
Kim G. Larsen  
Jean-François Raskin  
Pierre-Alain Reynier



# Overview

---

- Time games with full observability.
- Time games with partial observability.
- Example and motivation.
- Time games with lattice of observations.



# Controller Synthesis/TGA

---

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



# Previous Work

---

- The controller continuously observes the system (all delays & moves are observable).
  - [CONCUR'05] (Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, Didier Lime).
- The controller observes the system through one set of predicates (observations). Actions are urgent.
  - [ATVA'07] (Franck Cassez, Alexandre David, Kim G. Larsen, Didier Lime, Jean-François Raskin).



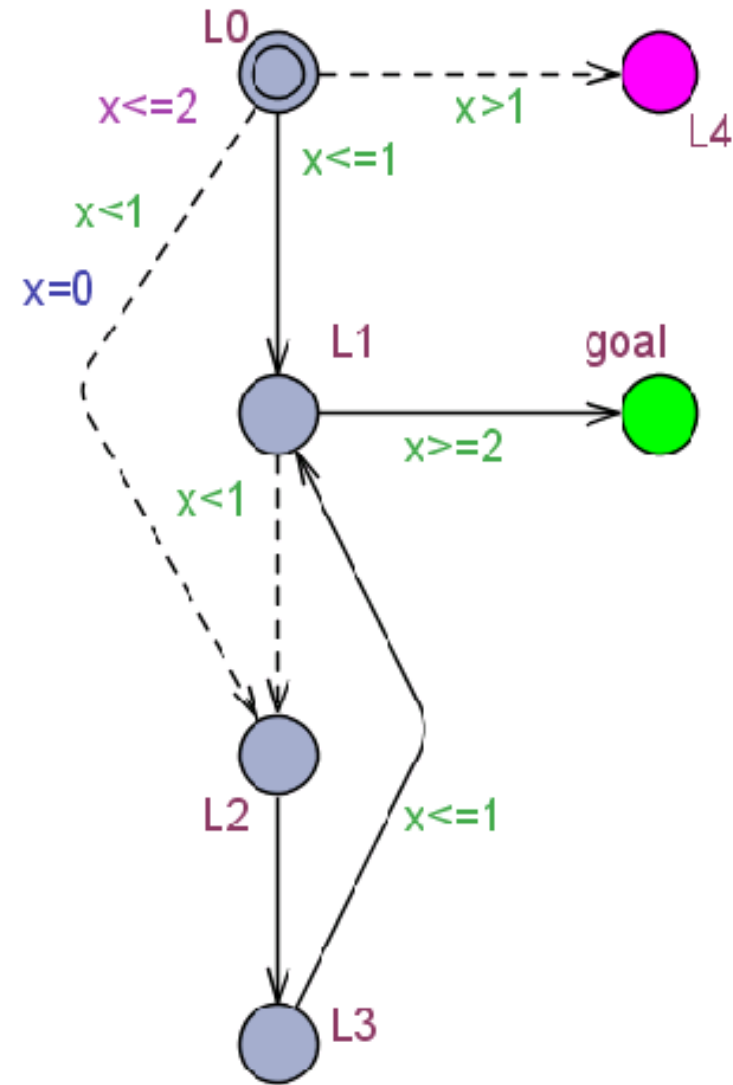
# Case-Studies

---

- [FORMATS'07] Guided Controller Synthesis Using UPPAAL-TIGA. Jan Jakob Jessen, Jacob Illum Rasmussen, Kim G. Larsen, Alexandre David.
- [HSCC'09] Automatic Synthesis of Robust and Optimal Controllers – An Industrial Case Study. Franck Cassez, J.J. Jessen, Kim G. Larsen, Jean-François Raskin, Pierre-Alain Reynier. (Hydac)

# Timed Game Automata

- Timed automata with controllable and uncontrollable transitions.
- Reachability & safety games.
  - control:  $A \langle \rangle \text{TGA.goal}$
  - control:  $A[\ ]$  not  $\text{TGA.L4}$
- Memoryless strategy:
  - state  $\rightarrow$  action.



# One Strategy

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

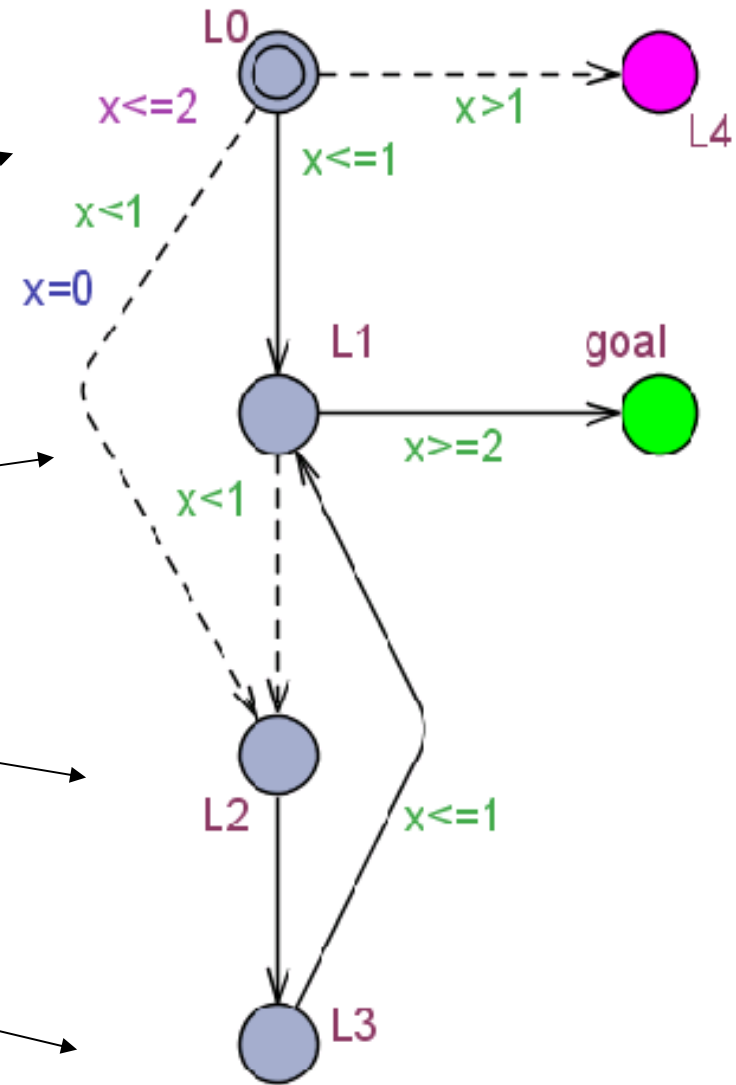
Strategy

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

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

$x \leq 1$  :  $c$

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





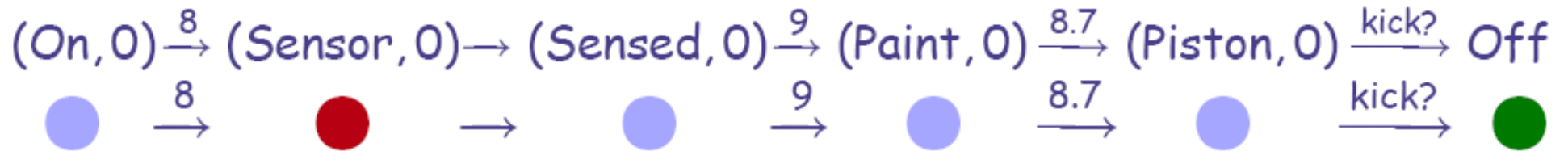
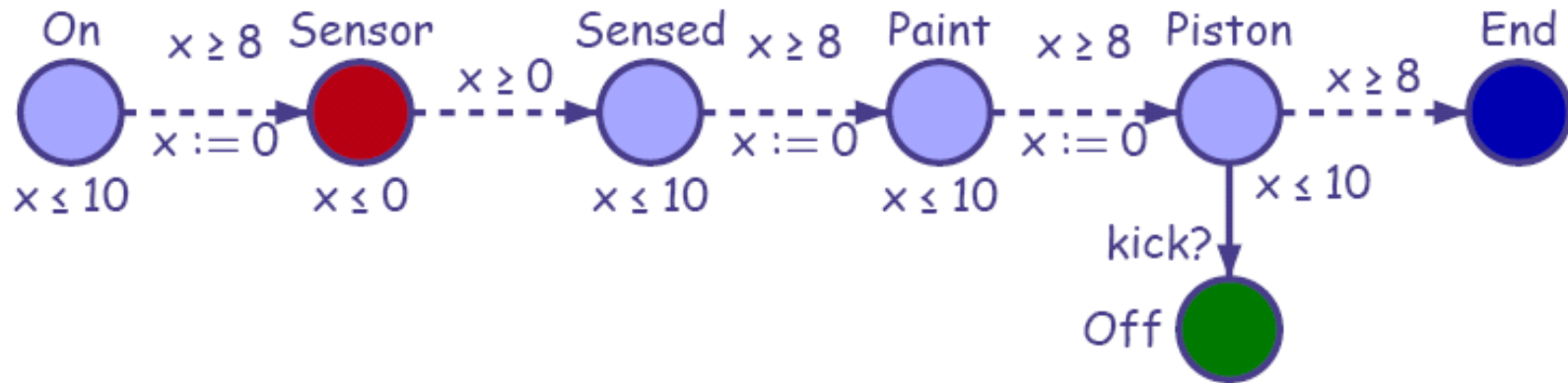
# Game with Partial Observability

---

- State-based observations
  - locations, variables
  - clock intervals of the form [lower,upper[
- What is observable:
  - change of observation,
  - *not* every state change.
  - → stuttering free observations.



# Example



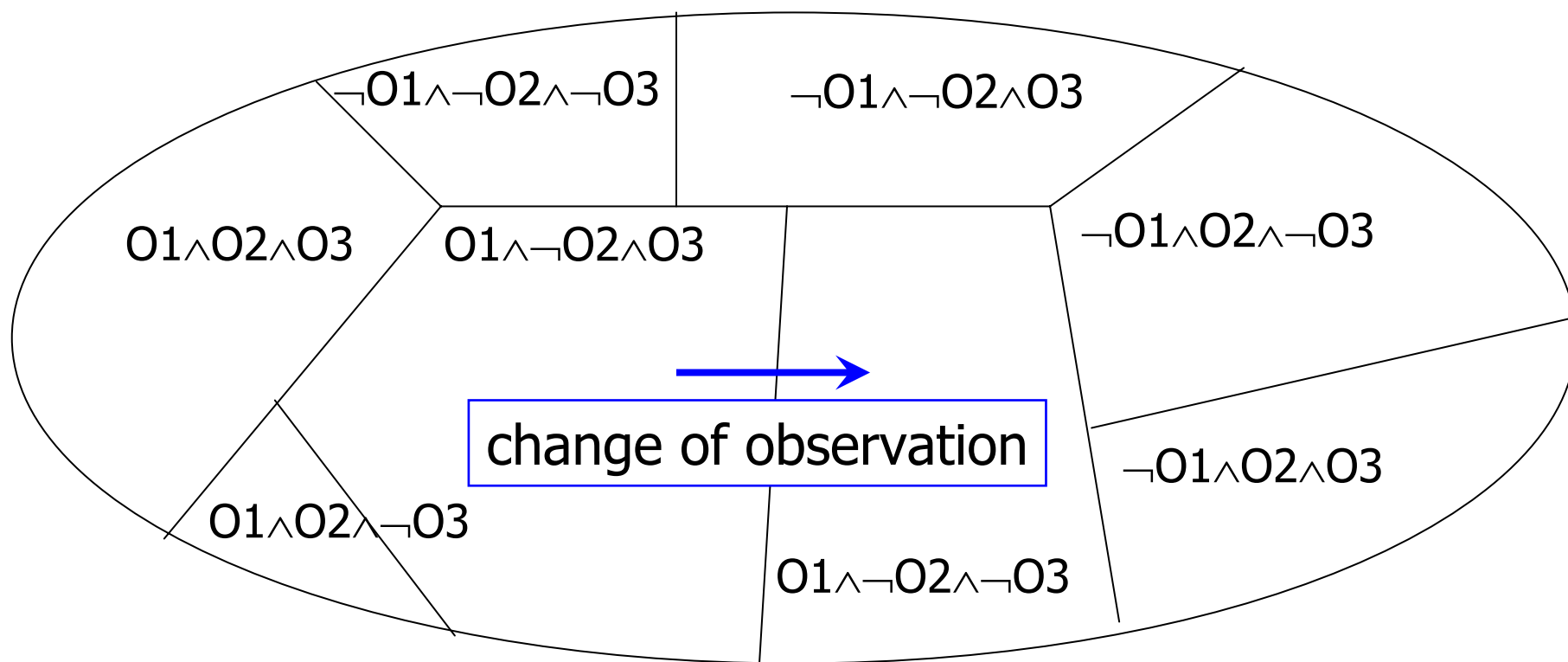
Assumption: the controller can only see **changes** of observations

Stuttering-free observation: ● ● ● ●

Must play based on **stuttering-free** observations

# Algorithm

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



# Algorithm with Reachability Objective

## Initialization:

1

```

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*[{s<sub>0</sub>}] ≠ 1 ∧ *Losing*[{s<sub>0</sub>}] ≠ 1)) **do**

*e* = (*W*, α, *W'*) ← *pop*(*Waiting*);

**if** *W'* ∉ *Passed* **then**

2

```

        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** \*)

3

```

        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**



# Initialization

$Passed \leftarrow \{\{s_0\}\};$

$Waiting \leftarrow \{(\{s_0\}, \alpha, W') \mid \alpha \in \Sigma_1, o \in \mathcal{O}, W' = \text{Next}_\alpha(\{s_0\}) \cap o \wedge W' \neq \emptyset\};$

$Win[\{s_0\}] \leftarrow (\{s_0\} \subseteq \gamma(\text{Goal}) ? 1 : 0);$

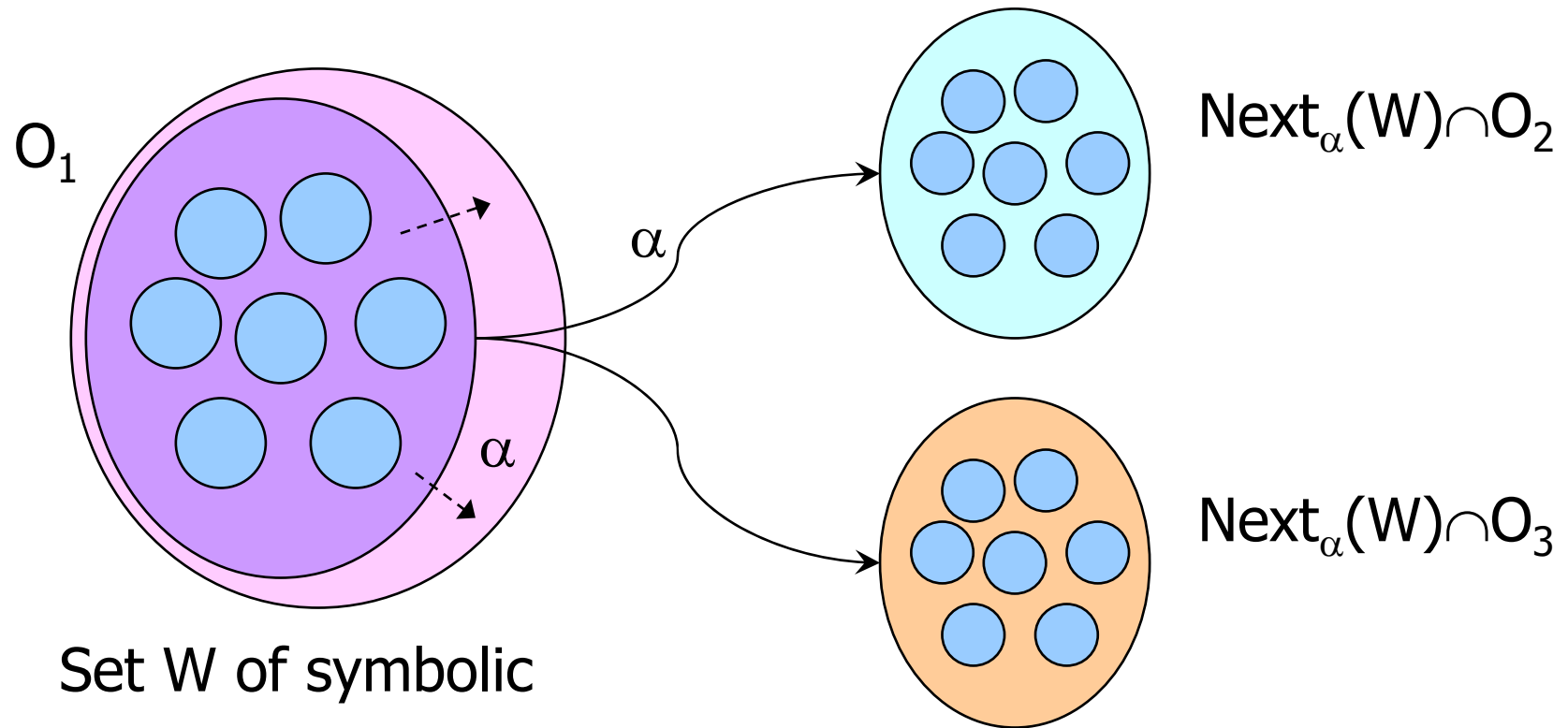
$Losing[\{s_0\}] \leftarrow (\{s_0\} \not\subseteq \gamma(\text{Goal}) \wedge (Waiting = \emptyset \vee \forall \alpha \in \Sigma_1, \text{Sink}_\alpha(s_0) \neq \emptyset) ? 1 : 0);$

$Depend[\{s_0\}] \leftarrow \emptyset;$

- 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.
- $\text{Sink}_\alpha$ : sink states by doing  $\alpha$ .

# Sets of Symbolic States & Successors

$\text{Next}_\alpha(W)$ : sets of symbolic successors for some action  $\alpha$ .



Set  $W$  of symbolic states within one observation.

$\text{Next}_\alpha$ : do  $\{\alpha$  asap and wait for  $\alpha\}$  until change of observation.

# Forward Phase

```
Passed ← Passed ∪ {W'};  
Depend[W'] ← {(W,  $\alpha$ , W')};  
Win[W'] ← (W' ⊆  $\gamma$ (Goal) ? 1 : 0);  
Losing[W'] ← (W' ⊈  $\gamma$ (Goal) ∧ Sink $_{\alpha}$ (W') ≠ ∅ ? 1 : 0);
```

Update Win,  
Losing, graph.

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

```
NewTrans ← {(W',  $\alpha$ , W'') |  $\alpha \in \Sigma$ ,  $o \in \mathcal{O}$ , W' = Next $_{\alpha}$ (W) ∩  $o \wedge W' \neq \emptyset$ };
```

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

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

```
Waiting ← Waiting ∪ NewTrans;
```

Continue forward.

Back-propagate if necessary.

Detect deadlocks.

Partition successors.



# Backward Phase

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

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

if  $Win^*$  then

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

then  $W$  is winning and back-propagate.

If every  $c$  has a losing successor

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

if  $Losing^*$  then

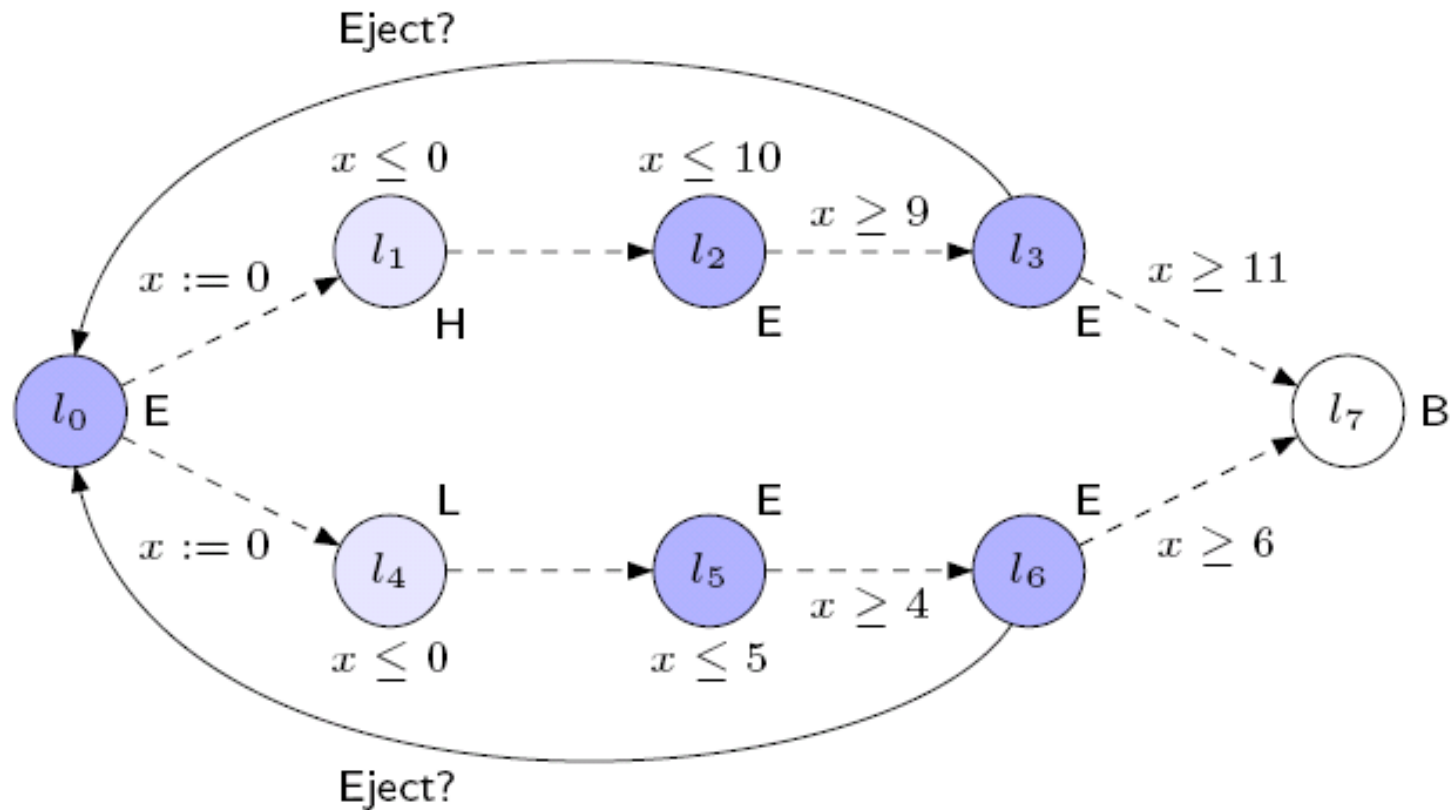
$$Waiting \leftarrow Waiting \cup Depend[W]; Losing[W] \leftarrow 1;$$

then  $W$  is losing and back-propagate.

Update graph in case of new paths to passed states.

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

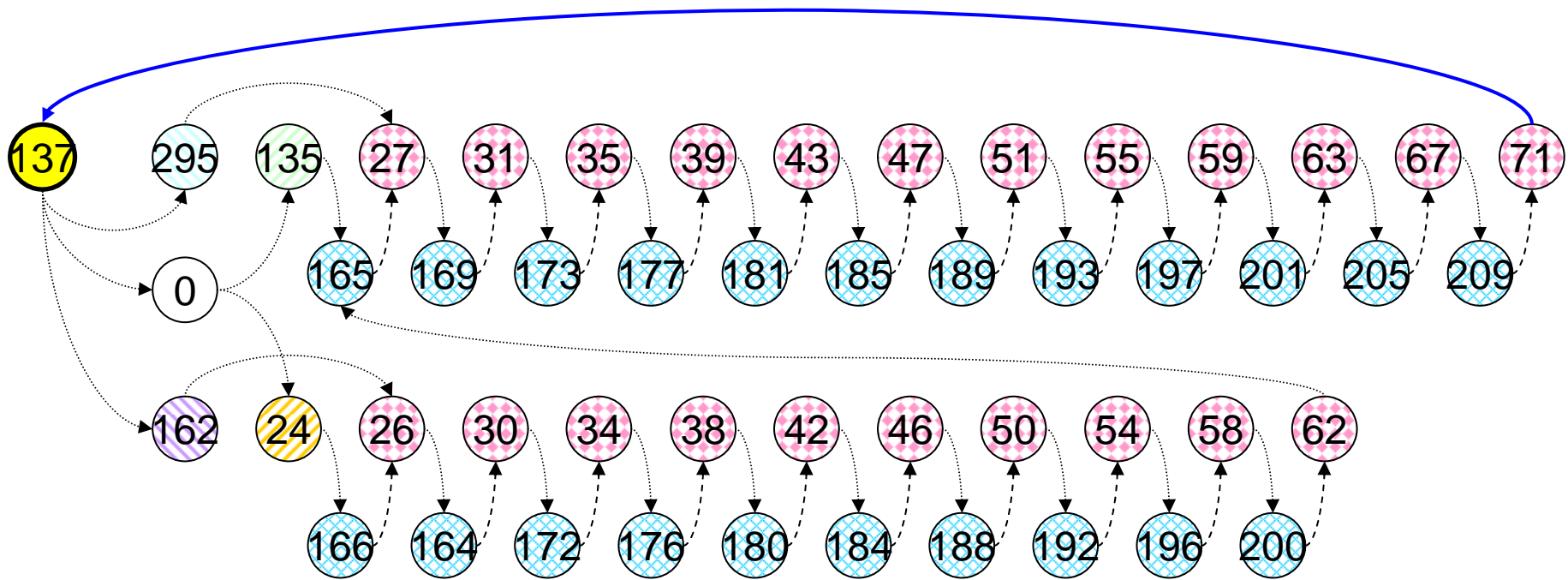
# Example - Model



Observations: L, H, E, B,  $y$  in  $[0, 1/2[$



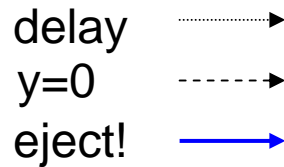
# Example – Memoryful Strategy



Partition:

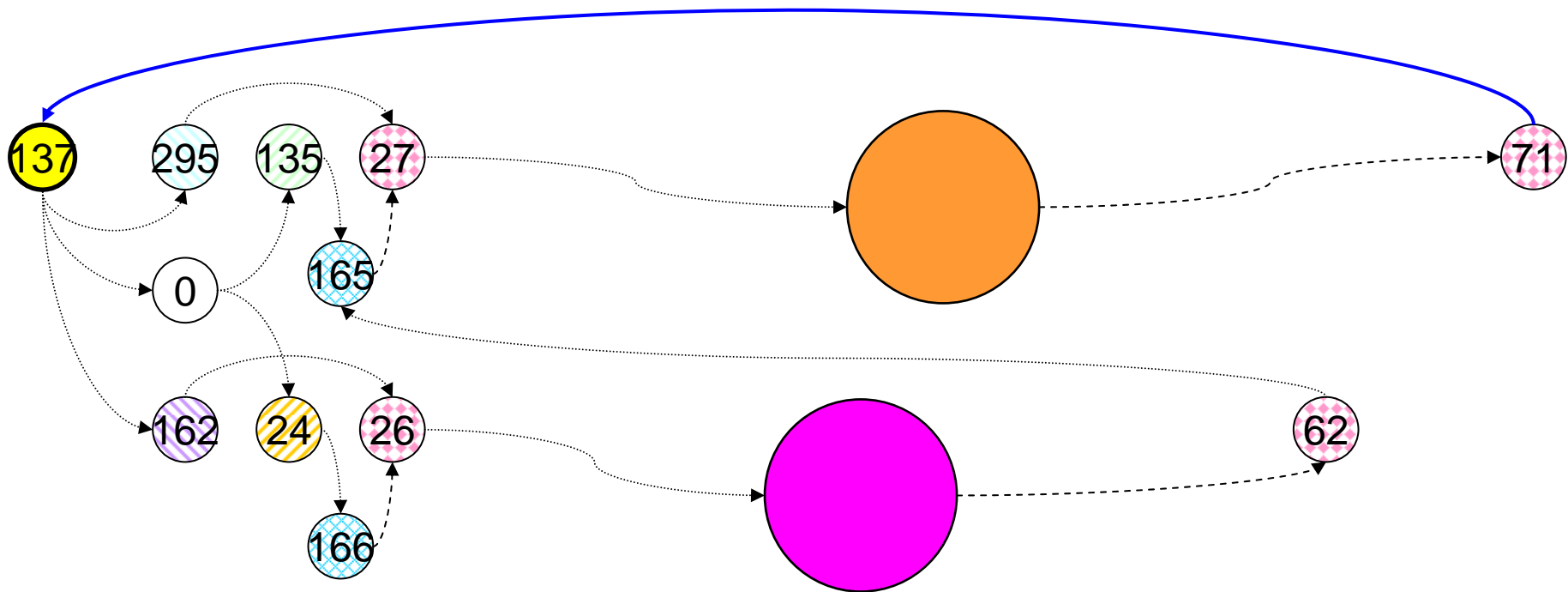


Actions:



Controllable with  $y \in [0, 1/2[$ ,  
not with  $y \in [0, 1[$ .

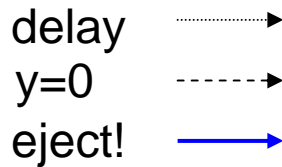
# Motivations



Partition:



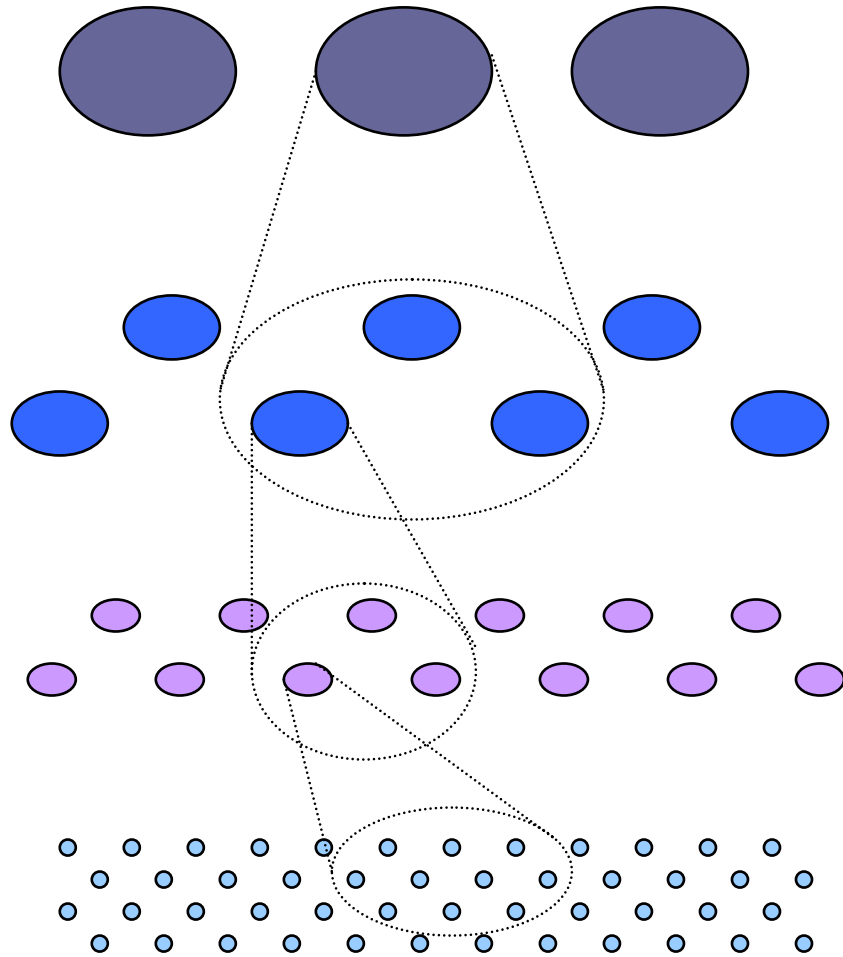
Actions:



Use coarser observations  
where possible

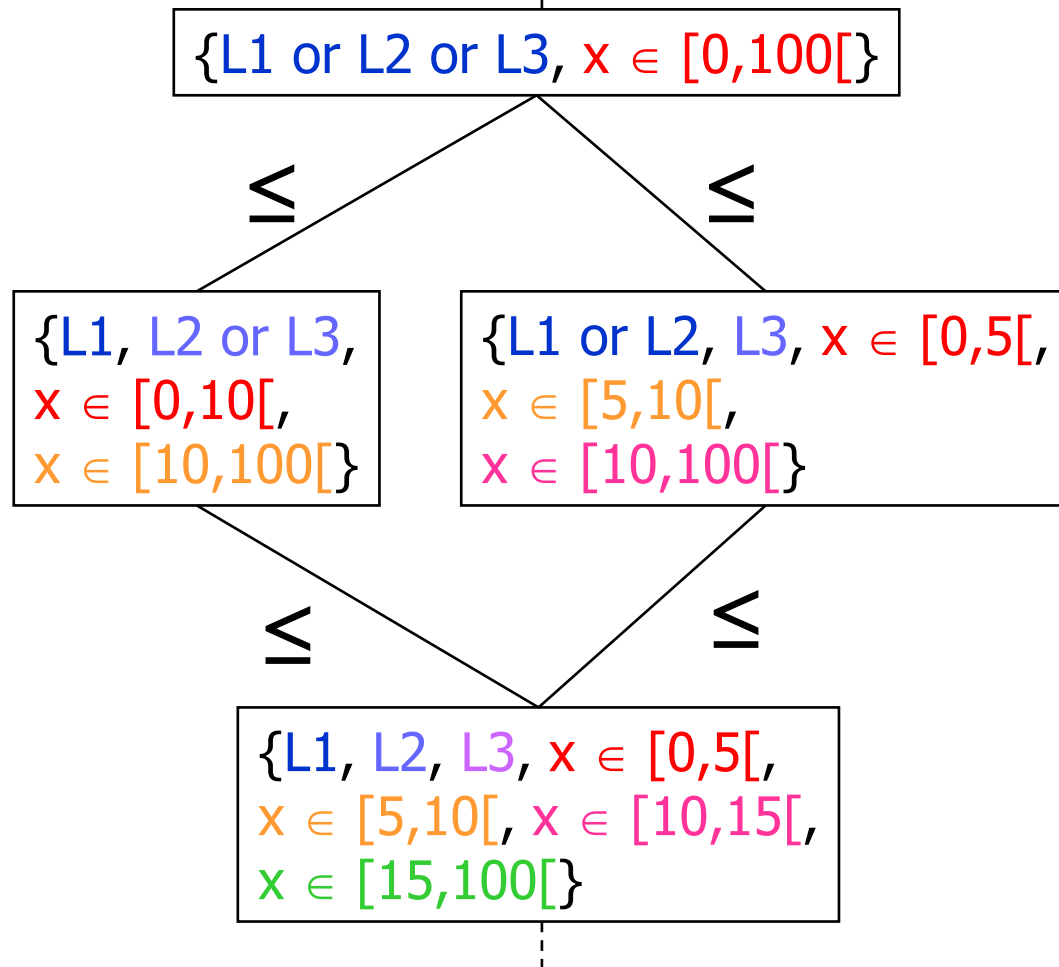
- shorter strategies
- cheaper sensors

# Algorithm: Intuition



- Use the algorithm for partial observability at a coarse level.
- Refine the observation and run the algorithm at the finer level.
- When to refine?
- Relationship between observations?

# Lattice of Observations



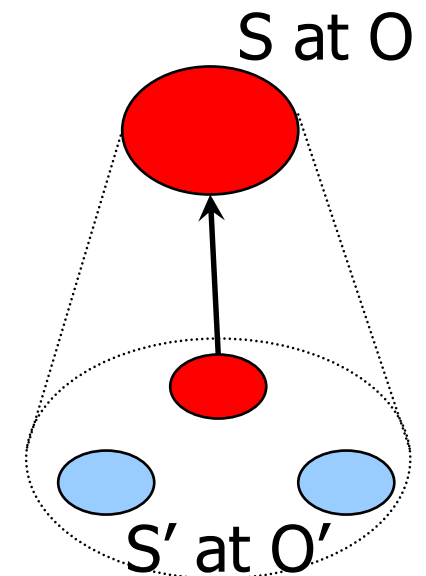
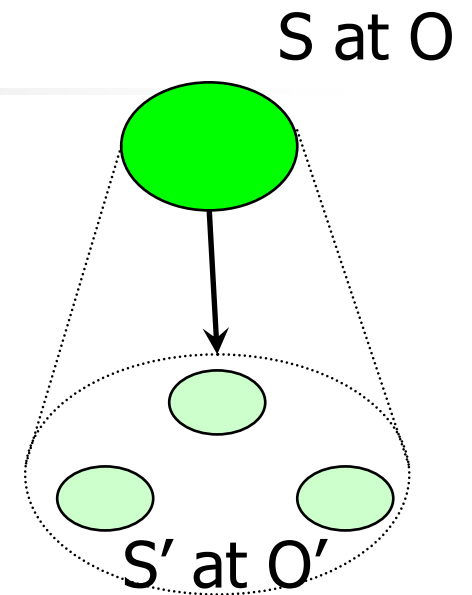
- Ordering relation: (Obs1 refines Obs2)  $\text{Obs1} \leq \text{Obs2}$  iff states observable by Obs2 are observable by Obs1.
  - Finer clock constraints.
  - Finer sets of locations
- Top: coarse observations. Supremum observes nothing (blind).
- Bottom: fine observation. Infimum: full observability.

Locations L1,L2,L3,L4, clock x.

# Lattice of Observations

## ■ Why?

- Notion of refinement of observations consistent with inclusion of sets of states.
- If  $S$  at  $O$  is winning then  $S'$  at  $O'$  is also winning for  $O'$  finer than  $O$  and  $S'$  included in  $S$ .
- If  $S'$  at  $O'$  is losing then  $S$  at  $O$  is also losing for  $O$  coarser than  $O'$  and  $S$  including  $S'$ .





# Algorithm for Safety Games

---

- Duplicate structures for each set of observations  $O$ .
  - $Waiting_O$ ,  $Passed_O$ ,  $Depend_O$
  - $Win_O$  is different
    - $0 \rightarrow$  lose
    - $? \rightarrow$  unknown
    - $1 \rightarrow$  win(put together previous Win and Losing)
  - No  $Losing_O$ .
  - $Refine_O$  maps  $S$  to its refined states.



# Algorithm

---

- Several actions that can be taken non-deterministically.
  - Initialize  $Win_0$  to ?.
  - ACT1: main exploration.
  - ACT2: propagate losing “up” (more abstract).
  - ACT3: propagate winning “down” (finer).
  - ACT4: conclude winning/losing at some  $O$ .
  - ACT5: change to winning because of refinement.

# Algorithm (1)

## ACT1

Pick  $(S, a, S')$  in some  $\text{Waiting}_O$

if  $S' \notin \text{Passed}_O$

add  $S'$  to  $\text{Passed}_O$

add  $(S, a, S')$  to  $\text{Depend}_O$

add successors of  $S'$  to  $\text{Waiting}_O$

compute  $\text{Win}_O[S']$

else

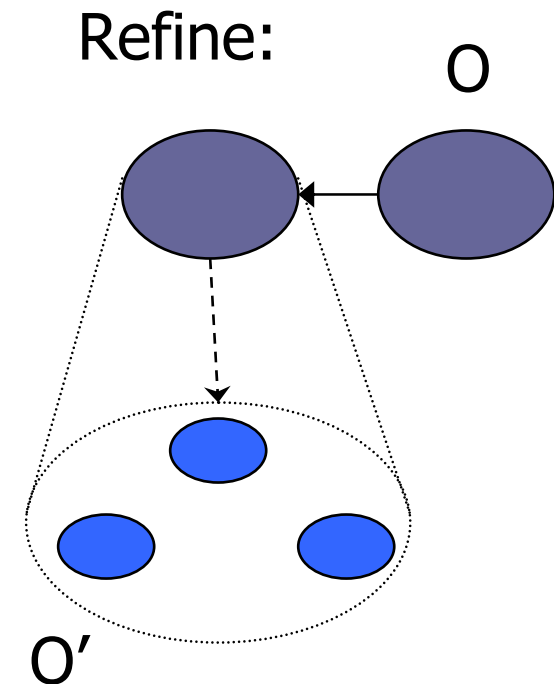
choose

:: Re-evaluate some  $\text{Win}_O[S]$

:: Refine  $S'$  for some finer  $O'$

and add the successors to  $\text{Waiting}_{O'}$

endif







# Algorithm (2,3)

## ACT2

if  $\text{Win}_O[S'] = 0$  then  
     $\text{Win}_O[S] = 0$  for any coarser  $O$  and larger  $S$   
endif

## ACT3

if  $\text{Win}_O[S'] = 1$  then  
     $\text{Win}_O[S] = 1$  for any finer  $O$  and smaller  $S$   
endif



# Algorithm (4)

Safety only

ACT4

if  $\text{Waiting}_O = \emptyset$  then

$\text{Win}_O[S] = 1$  for any  $S$  in  $\text{Passed}_O$  where  $\text{Win}_O[S] = ?$   
endif

Reachability only

ACT4

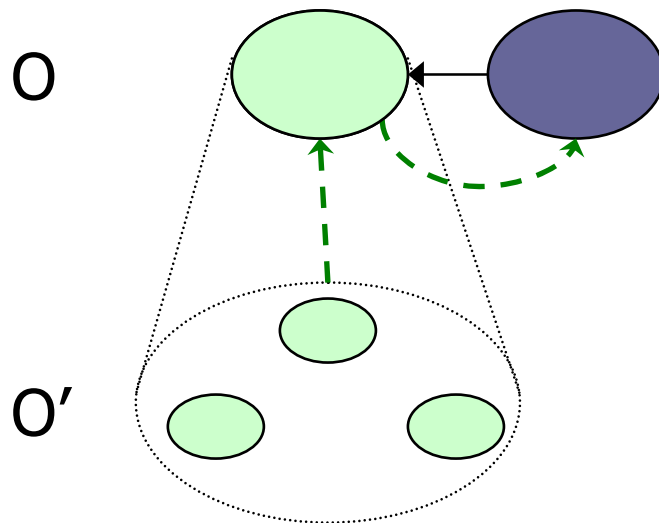
if  $\text{Waiting}_O = \emptyset$  then

$\text{Win}_O[S] = 0$  for any  $S$  in  $\text{Passed}_O$  where  $\text{Win}_O[S] = ?$   
endif

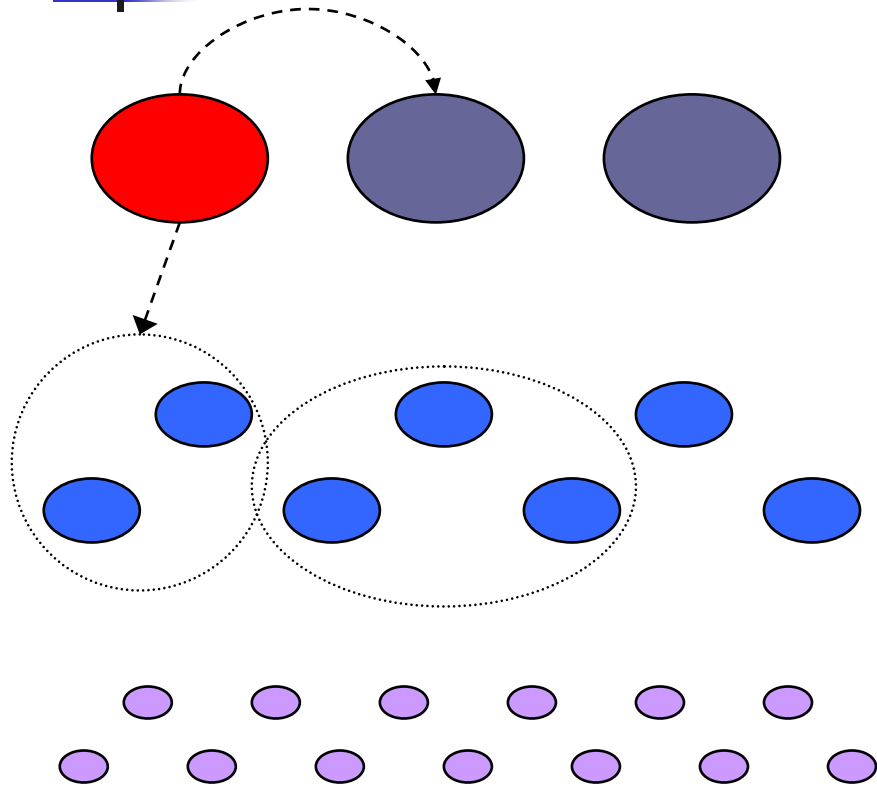
# Algorithm (5)

## ACT5

if  $\text{Win}_O[S] \neq 1$  and  
     $\forall S' \in \text{Refine}_O[S]. \text{Win}_O[S'] = 1$  then  
     $\text{Win}_O[S] = 1$   
    add  $\text{Depend}_O[S]$  to  $\text{Waiting}_O$   
endif



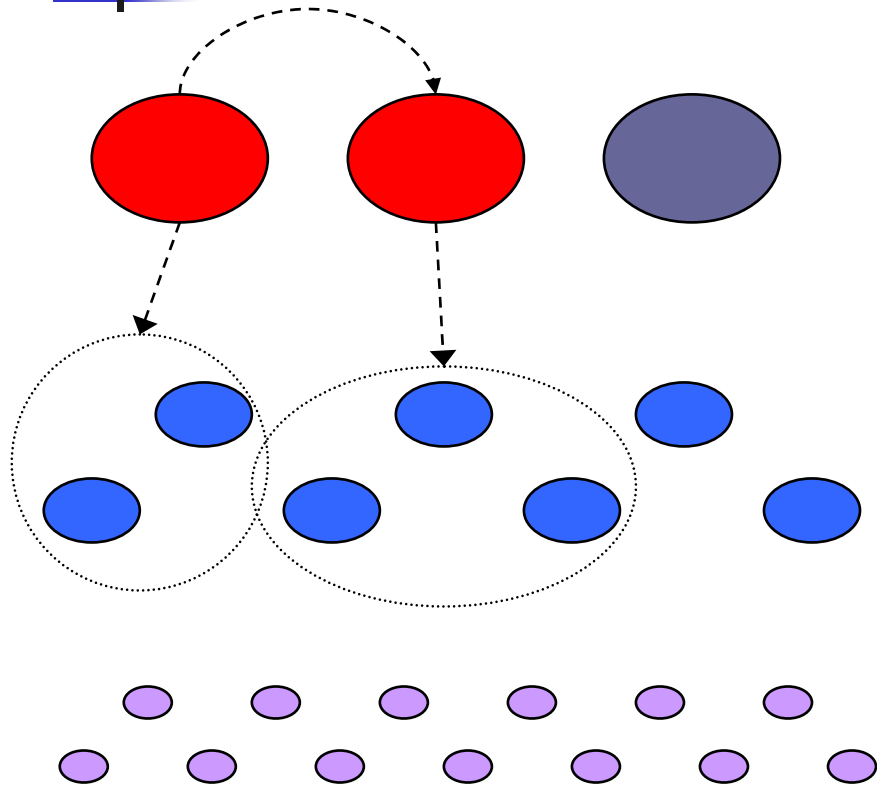
# Issue with Strategies



1. Explore forward.
2. Find losing.
3. Back-propagate.

Suppose we refine when detecting losing.

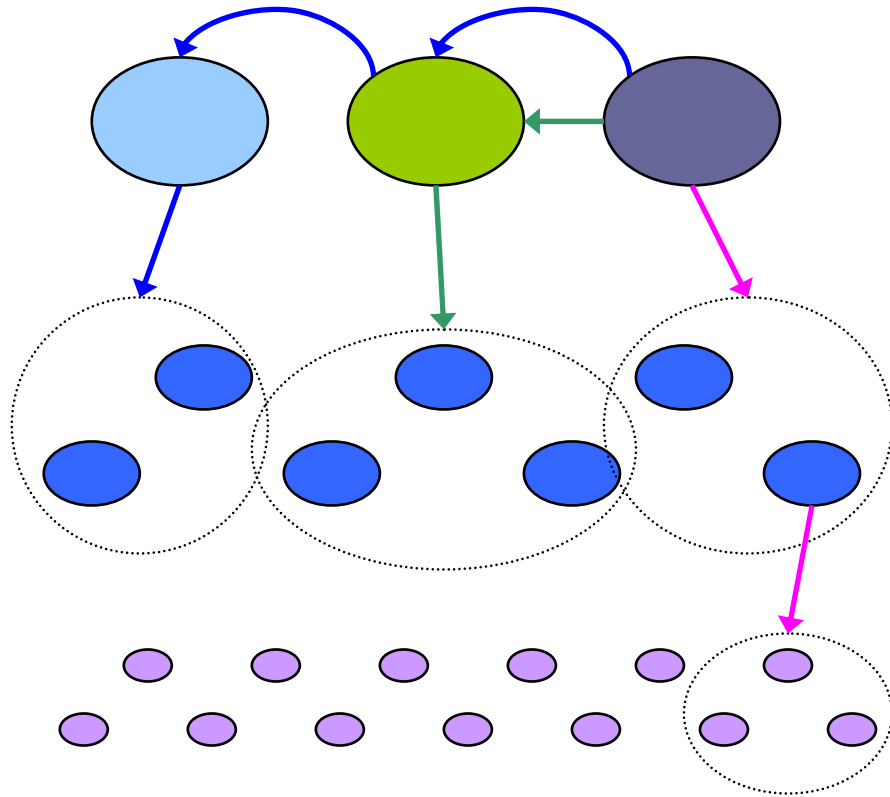
# Issue with Strategies



1. Explore forward.
2. Find losing.
3. Back-propagate.

Information is not there at the right place to extract the strategy.

# Issue with Strategies



- Different possible solutions.
- “Good” path is not known.
- 2<sup>nd</sup> issue to ensure progress → no loop.
  - Fix-point of winning  $\neq$  how to win.
  - Solved for full observability, different problem here.



# Work in Progress

---

- Experiments done with a Ruby prototype implementation – no strategy.
- Post-processing of refinement graph can solve the problem.
- On-the-fly problem: How to choose good actions.