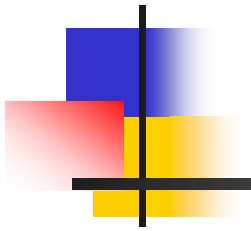
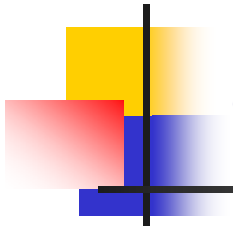
Controller Synthesis Based on On-Demand Partial Observations

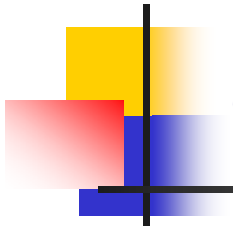


Franck Cassez
Alexandre David
Kim G. Larsen
Jean-François Raskin
Pierre-Alain Reynier



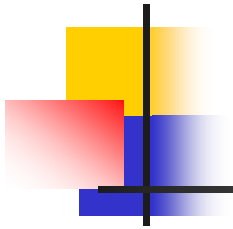
Overview

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



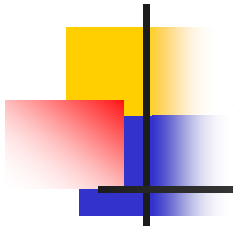
Controller Synthesis/TGA

- Given
  - System moves S,
  - Controller moves C,
  - and a property φ,
- find
  - a strategy S_c s.t. $S_c || S \models \varphi$,
  - 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).

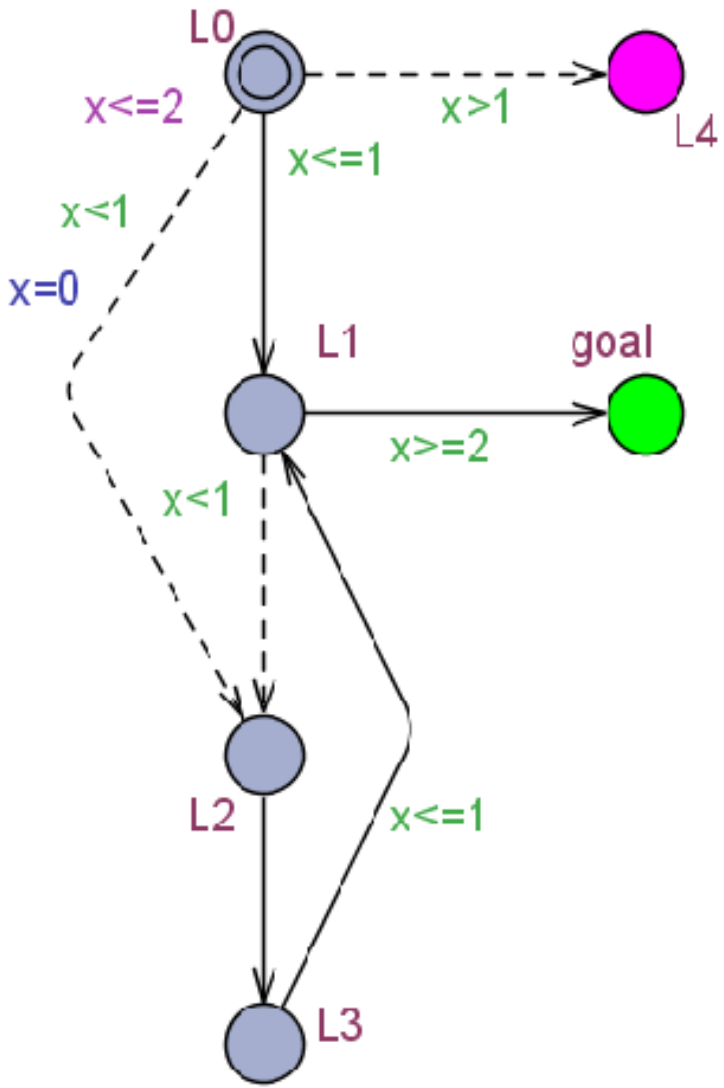


Case-Studies

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



One Strategy

- control: A<> TGA.goal

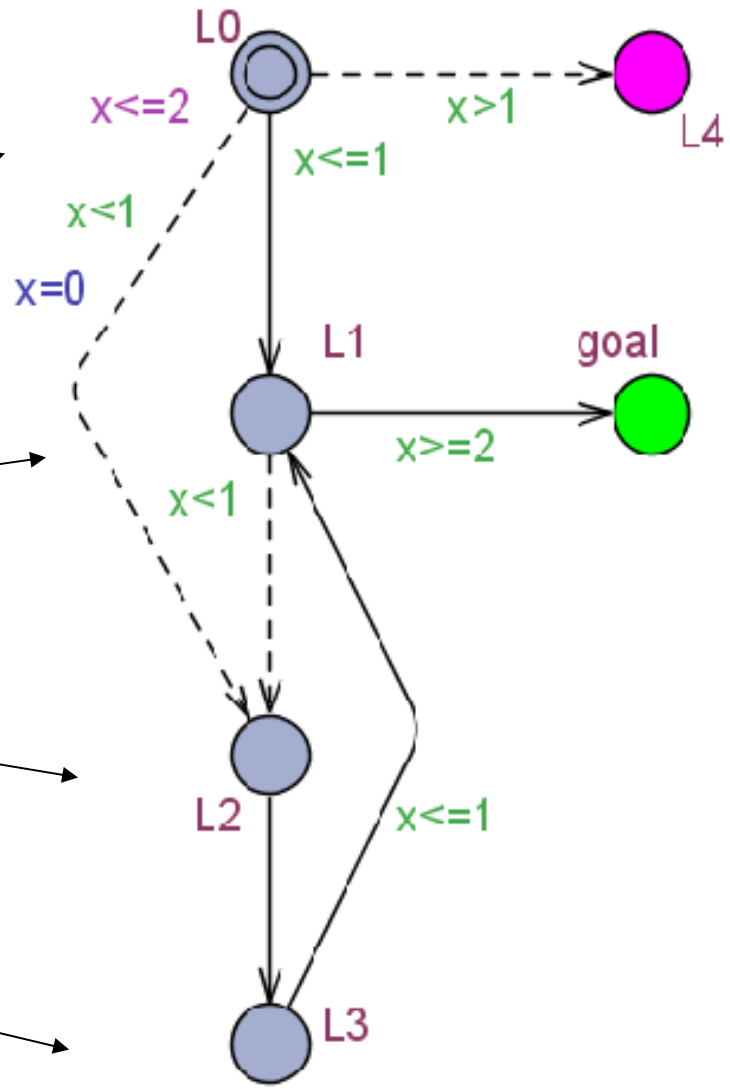
Strategy

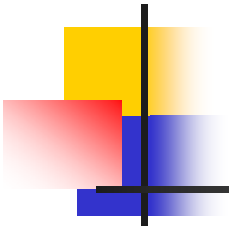
x<1	: λ
x==1	: c

x<2	: λ
x≥2	: c

x≤1	: c

x<1	: λ
x==1	: c

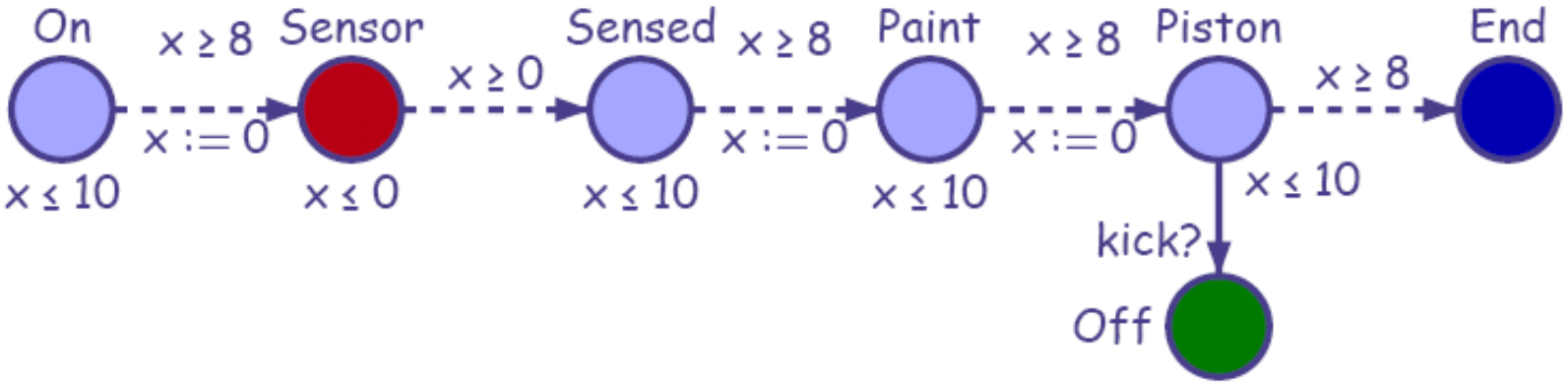




Game with Partial Observability

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

Example



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

Assumption: the controller can only see changes of observations

Stuttering-free observation:

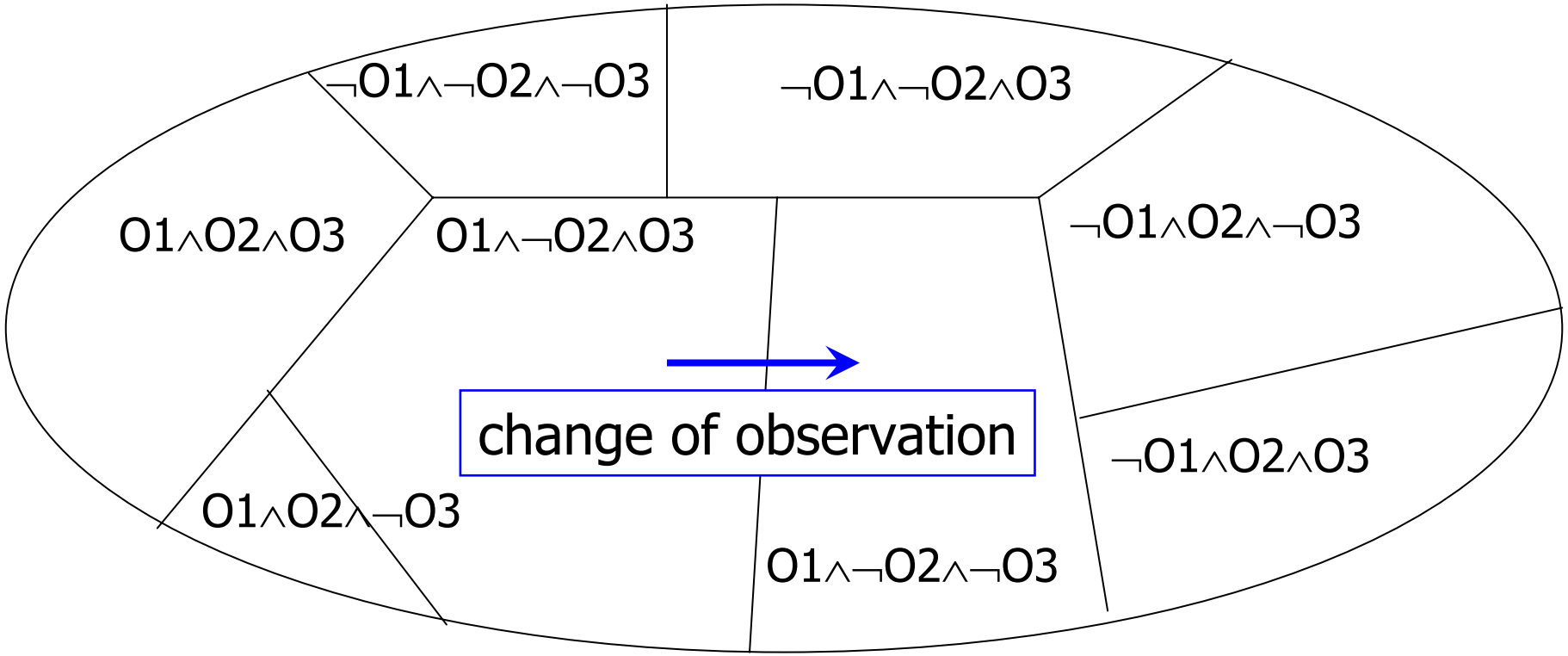
Must play based on stuttering-free observations

Algorithm

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

Observations O1 O2 O3.

Winning/losing is observable.



Algorithm with Reachability Objective

1

2

3

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 $W' \notin Passed$ **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

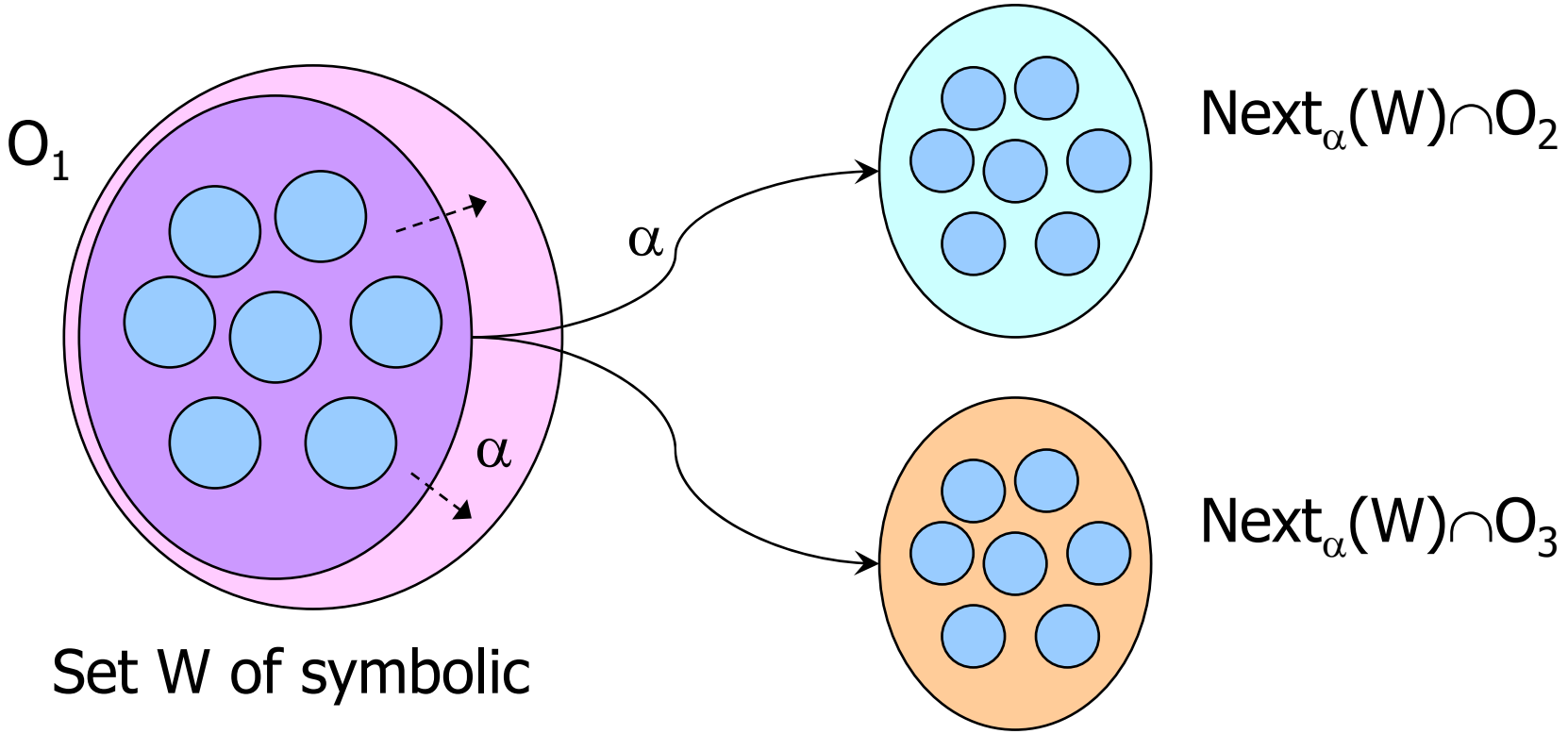
Initialization

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

- Passed list { sets of W }.
- Waiting list { tuples (W,α,W') }.
- Win[W] maps to 1 (winning) or 0 (unknown).
- Similar Losing[W].
- Depend[W] records the graph.
- Sink_α: sink states by doing α.

Sets of Symbolic States & Successors

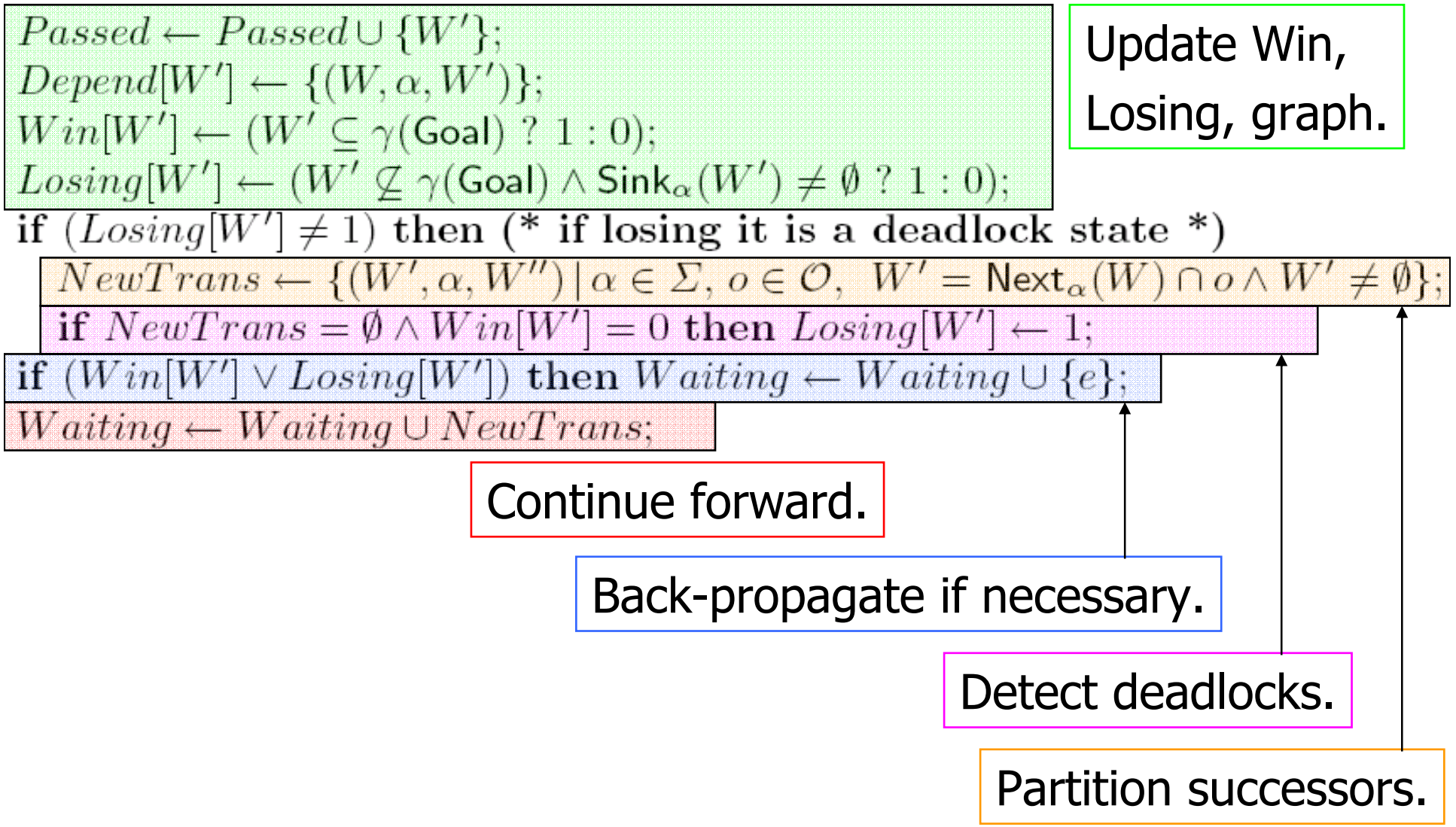
$Next_\alpha(W)$: sets of symbolic successors for some action α.



Set W of symbolic states within one observation.

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

Forward Phase



Backward Phase

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

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

if Win^* **then**

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

then W is winning and back-propagate.

If every *c* has a losing successor

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

if $Losing^*$ **then**

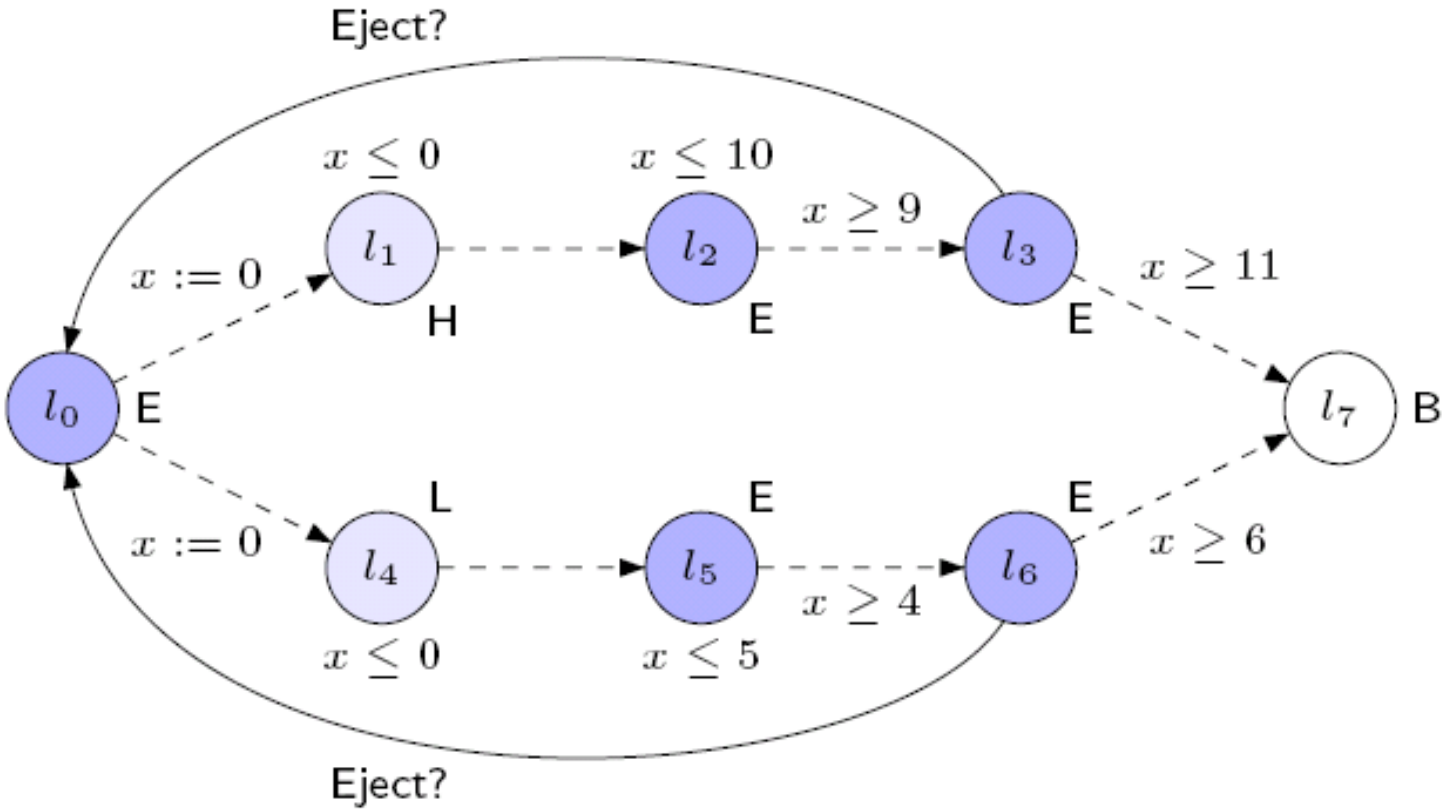
$$Waiting \leftarrow Waiting \cup Depend[W]; Losing[W] \leftarrow 1;$$

then W is losing and back-propagate.

Update graph in case of new paths to passed states.

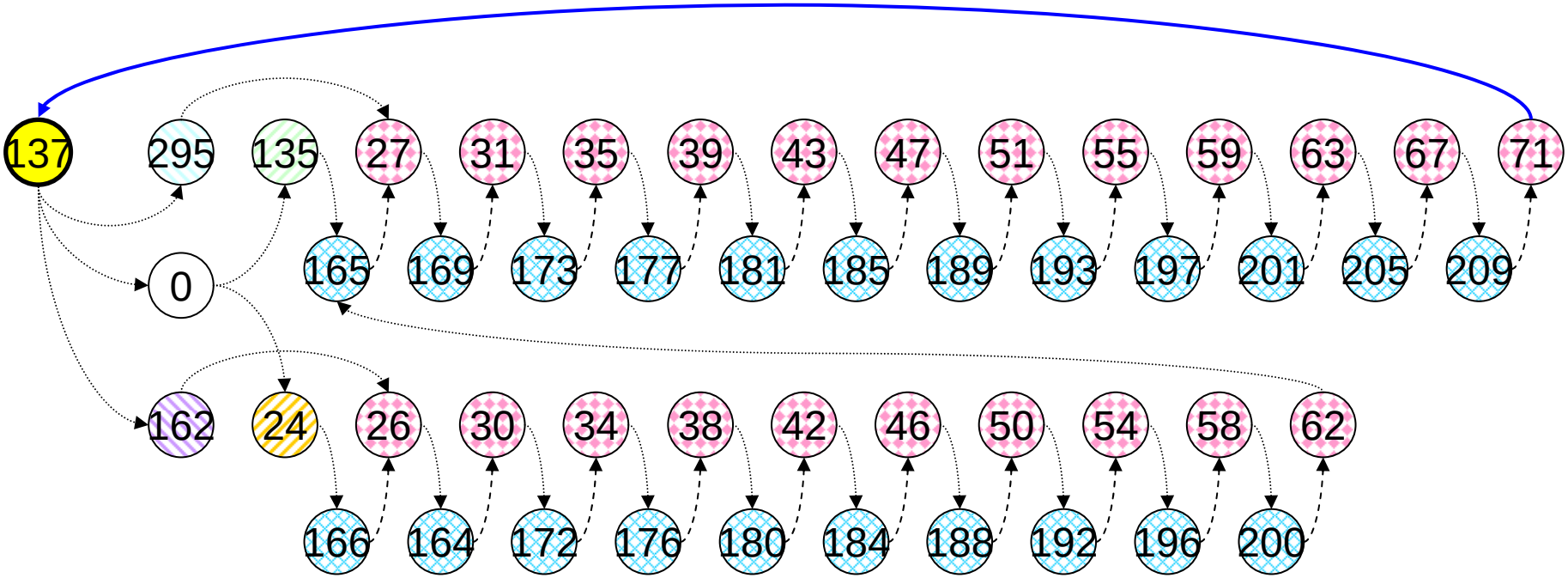
if $(Win[W'] = 0 \wedge Losing[W'] = 0)$ **then** $Depend[W'] \leftarrow Depend[W'] \cup \{e\};$

Example - Model



Observations: L, H, E, B, y in [0,½[

Example – Memoryful Strategy



Partition:

Actions:

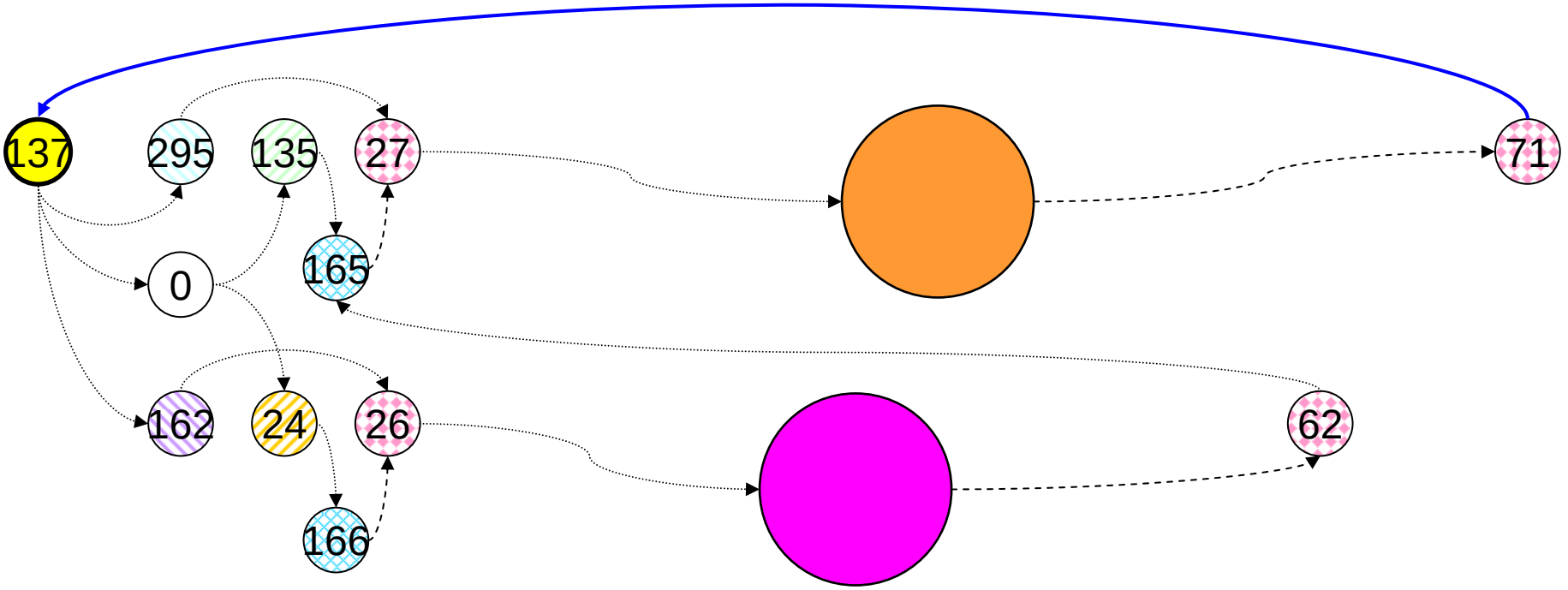
delay

y=0

eject!

Controllable with $y \in [0, ½[$, not with $y \in [0,1[$.

Motivations



Partition:

Actions:

delay

y=0

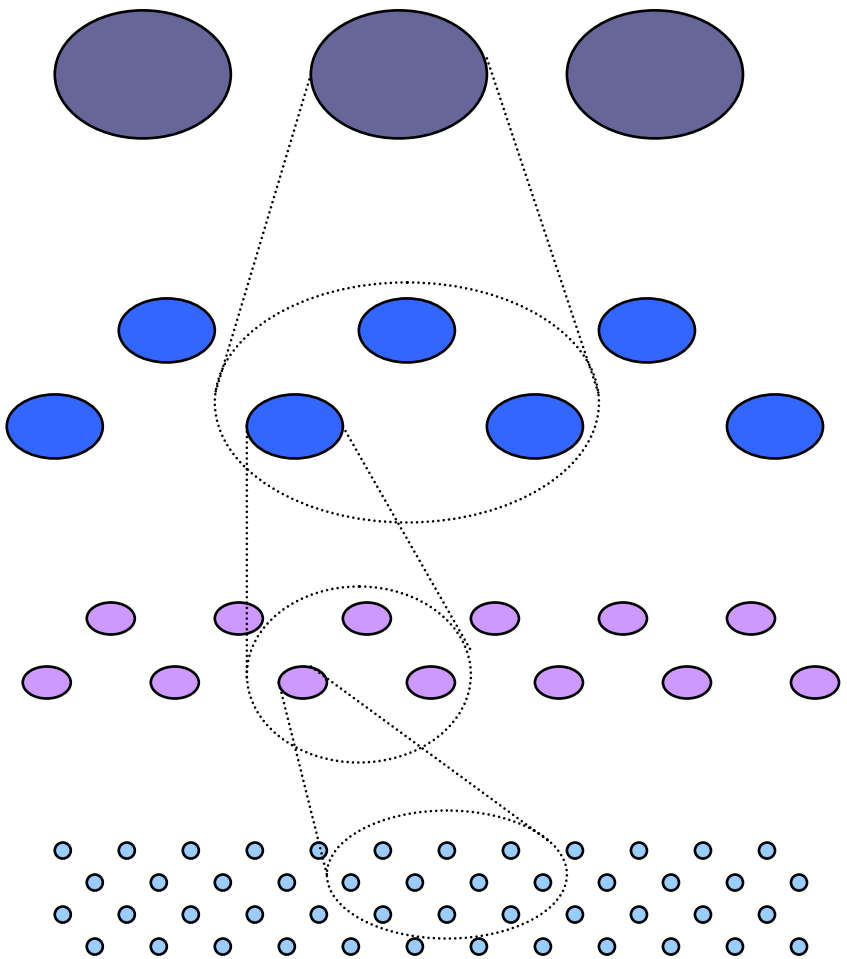
eject!

Use coarser observations where possible

→ shorter strategies

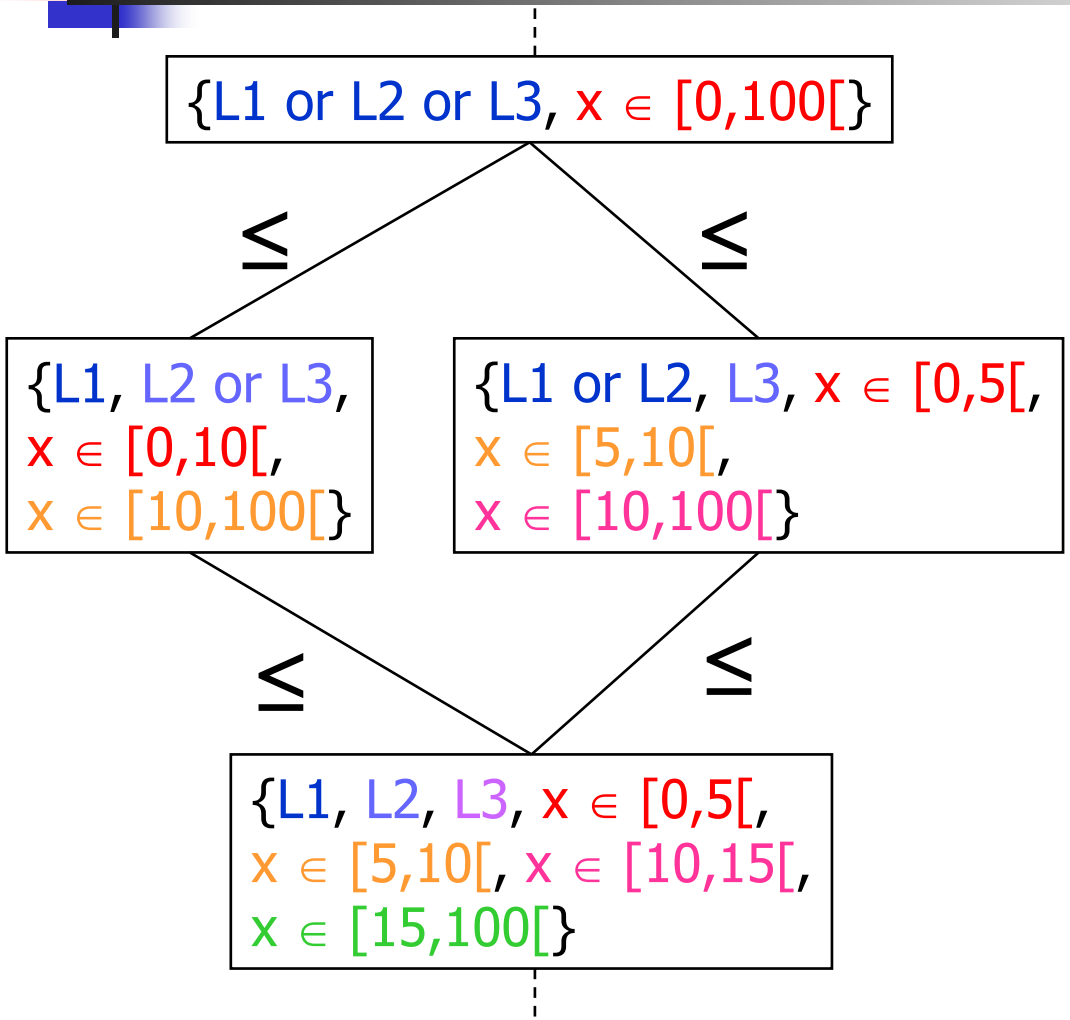
→ cheaper sensors

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

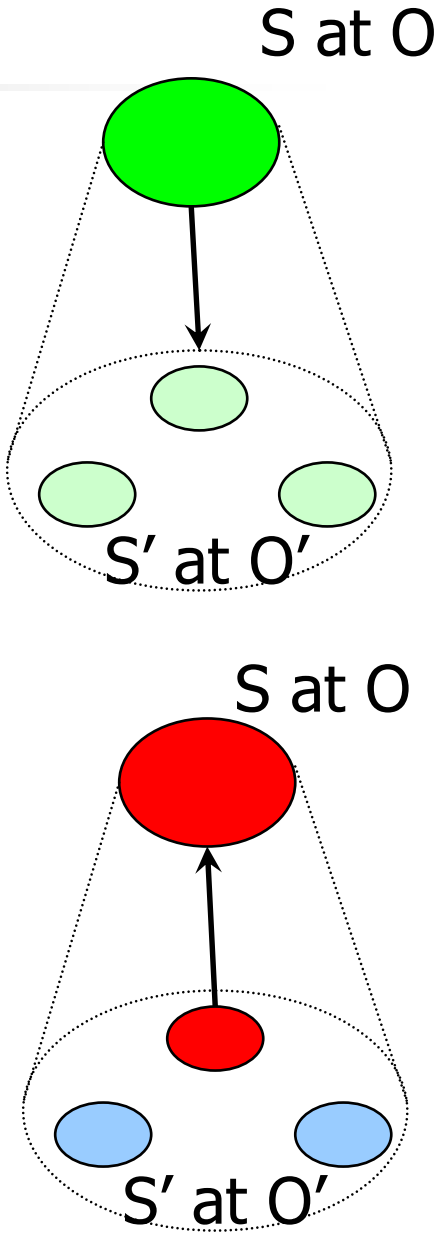


Locations L1,L2,L3,L4, clock x.

- Ordering relation: (Obs1 refines Obs2) Obs1 ≤ 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.

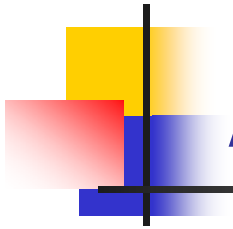
Lattice of Observations

- 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.



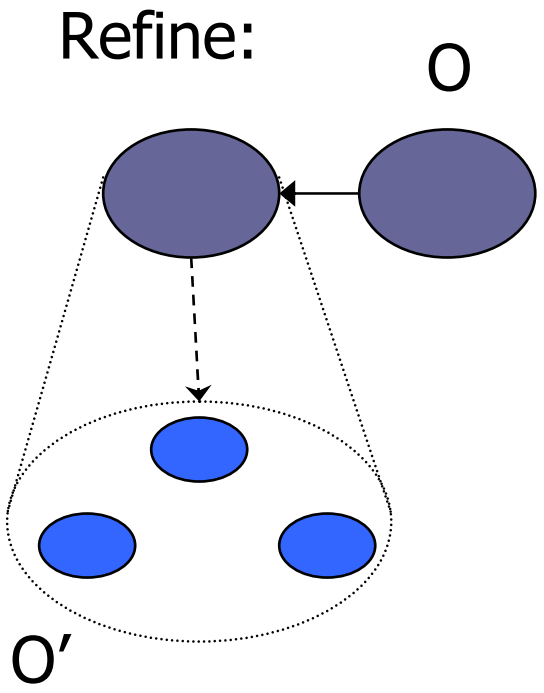
Algorithm

- Several actions that can be taken non-deterministically.
  - Initialize Win_O 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 $Waiting_O$
if $S' \notin Passed_O$
  add S' to $Passed_O$
  add (S,a,S') to $Depend_O$
  add successors of S' to $Waiting_O$
  compute $Win_O[S']$
else
  choose
  :: Re-evaluate some $Win_O[S]$
  :: Refine S' for some finer O'
    and add the successors to $Waiting_{O'}$
endif



Algorithm (2,3)

ACT2

if $Win_{O'}[S'] = 0$ then

$Win_O[S] = 0$ for any coarser O and larger S

endif

ACT3

if $Win_{O'}[S'] = 1$ then

$Win_O[S] = 1$ for any finer O and smaller S

endif

Algorithm (4)

Safety only

ACT4
  if $Waiting_O = \emptyset$ then
    $Win_O[S] = 1$ for any S in $Passed_O$ where $Win_O[S] = ?$
  endif

Reachability only

ACT4
  if $Waiting_O = \emptyset$ then
    $Win_O[S] = 0$ for any S in $Passed_O$ where $Win_O[S] = ?$
  endif

Algorithm (5)

ACT5

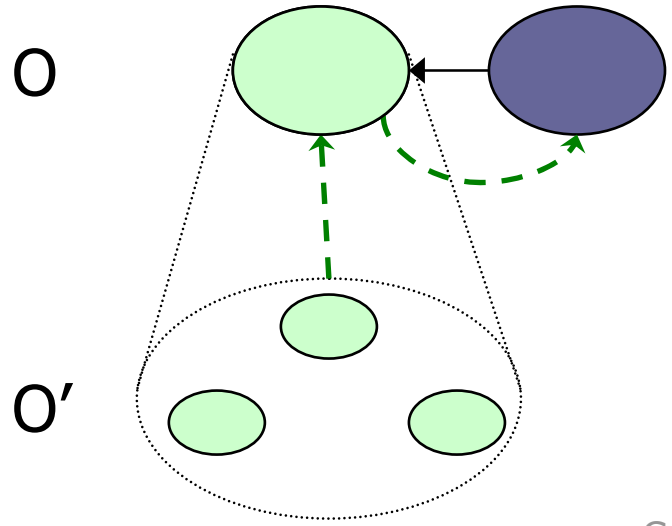
if $Win_O[S] \neq 1$ and

$\forall S' \in Refine_O[S]. Win_{O'}[S'] = 1$ then

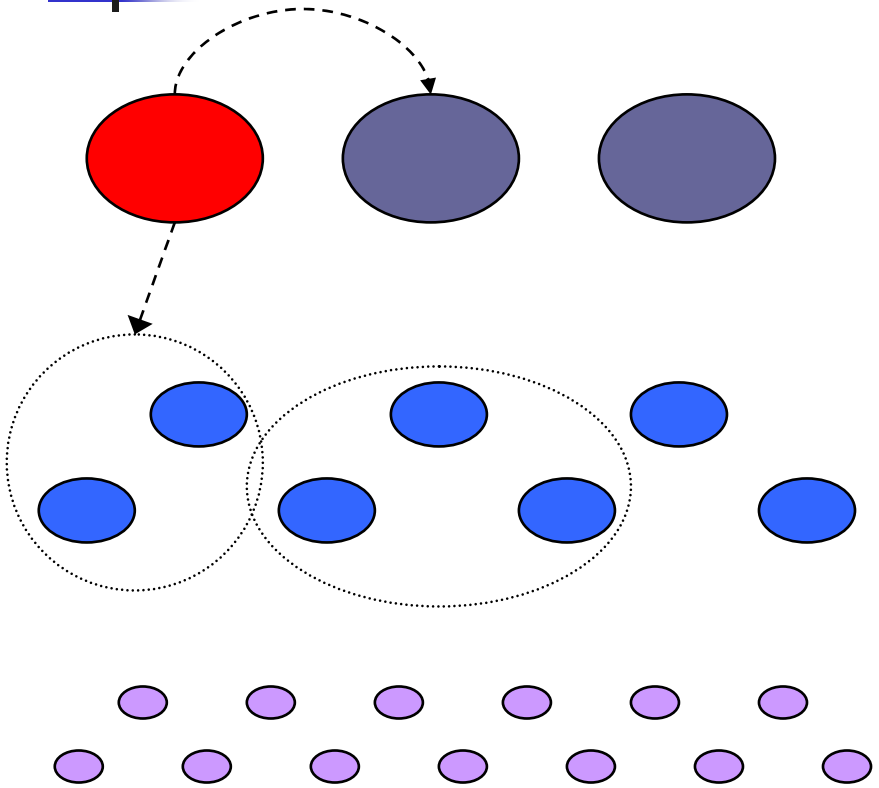
$Win_O[S] = 1$

add $Depend_O[S]$ to $Waiting_O$

endif



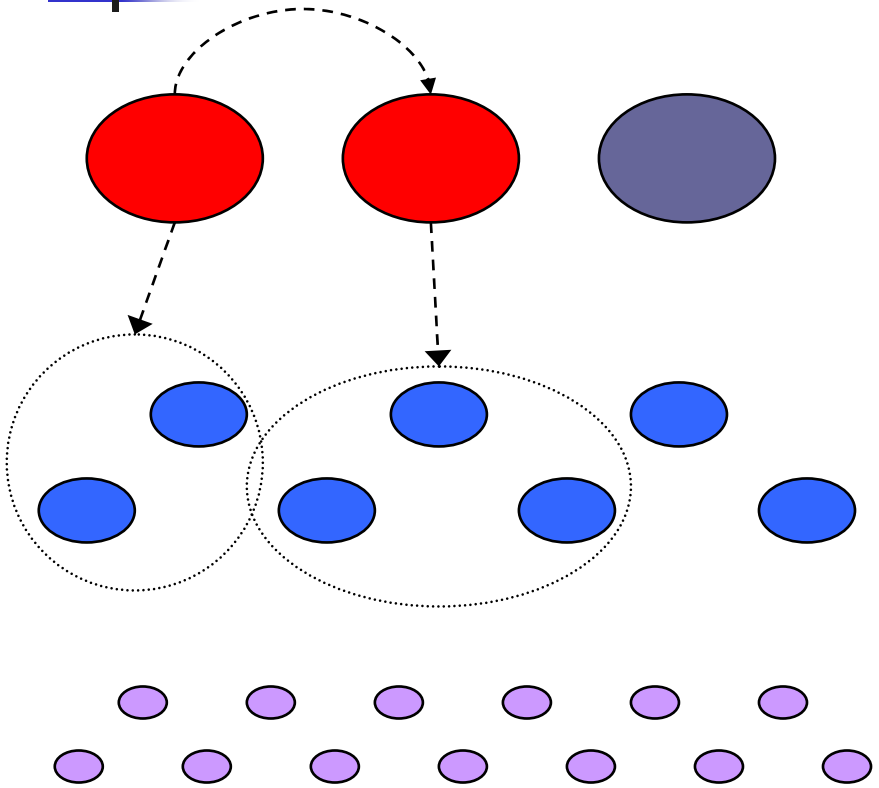
Issue with Strategies



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

Suppose we refine when detecting losing.

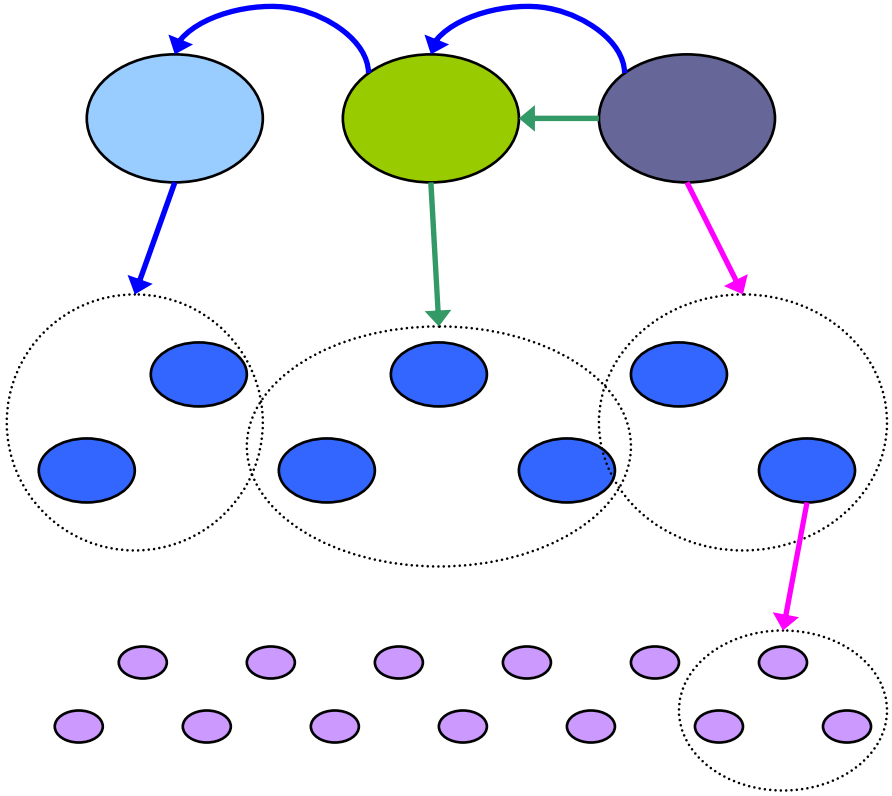
Issue with Strategies



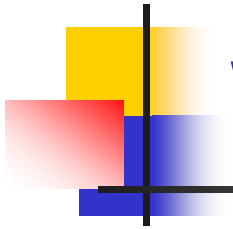
1. Explore forward.
2. Find losing.
3. Back-propagate.

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

Issue with Strategies



- Different possible solutions.
- "Good" path is not known.
- 2nd issue to ensure progress → no loop.
  - Fix-point of winning ≠ 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.