

Lecture 6

Module I: Model Checking

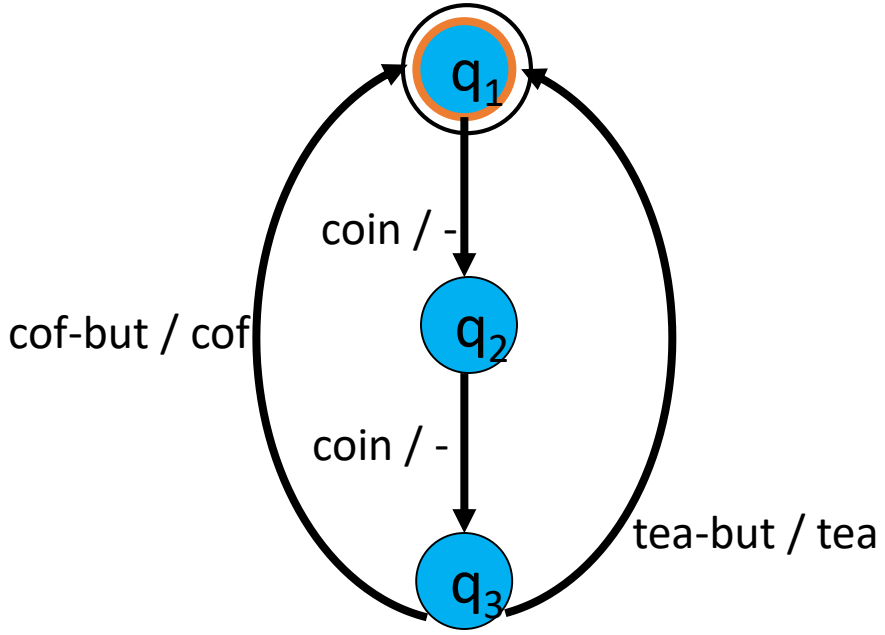
Topic: Model checking timed transition
systems: timed automata

J.Vain

10.03.2021

Slides by **Brian Nielsen** (AU)

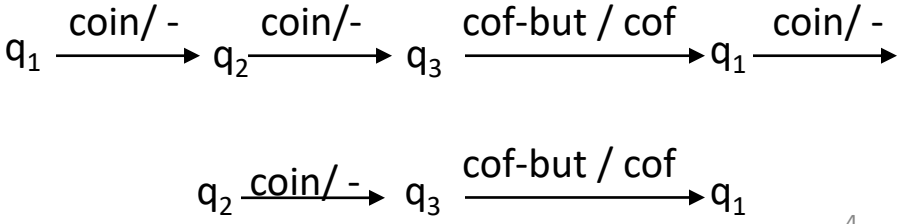
Finite State Machine (Mealy)



condition		effect	
current state	input	output	next state
q ₁	coin	-	q ₂
q ₂	coin	-	q ₃
q ₃	cof-but	cof	q ₁
q ₃	tea-but	tea	q ₁

Inputs = {cof-but, tea-but, coin}
 Outputs = {cof,tea}
 States: {q₁,q₂,q₃}
 Initial state = q₁
 Transitions= {
 (q₁, coin, -, q₂),
 (q₂, coin, -, q₃),
 (q₃, cof-but, cof, q₁),
 (q₃, tea-but, tea, q₁)
 }

Sample run:

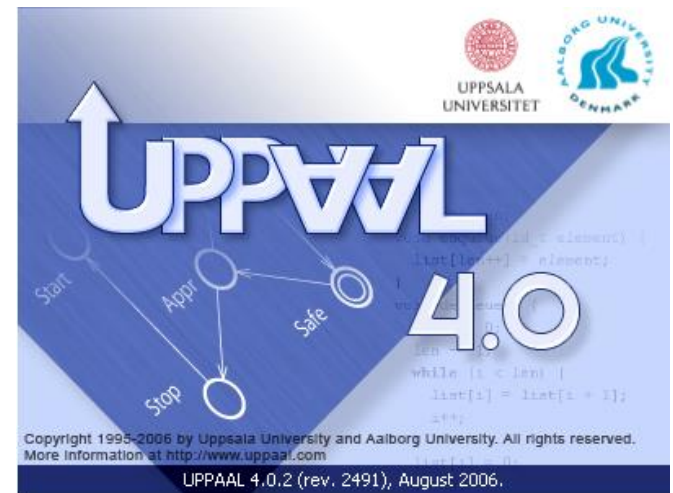


Adding Time

FSM



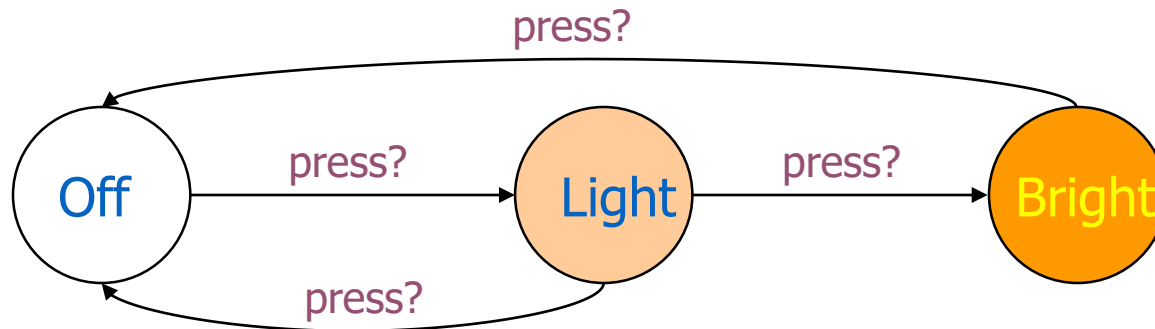
Timed Automata



Dumb Light Control

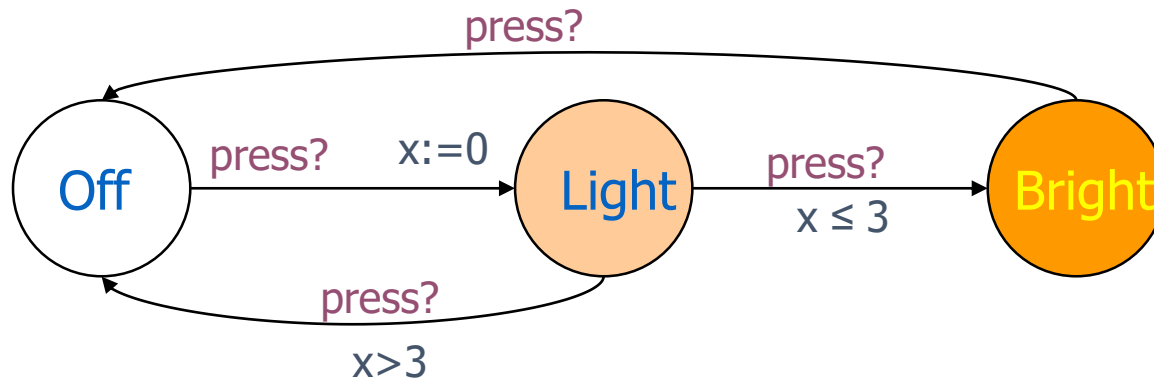
Requirement: if **press** is issued twice **quickly** then the **light** will get **brighter**; otherwise the light is turned **off**.

Solution 1:

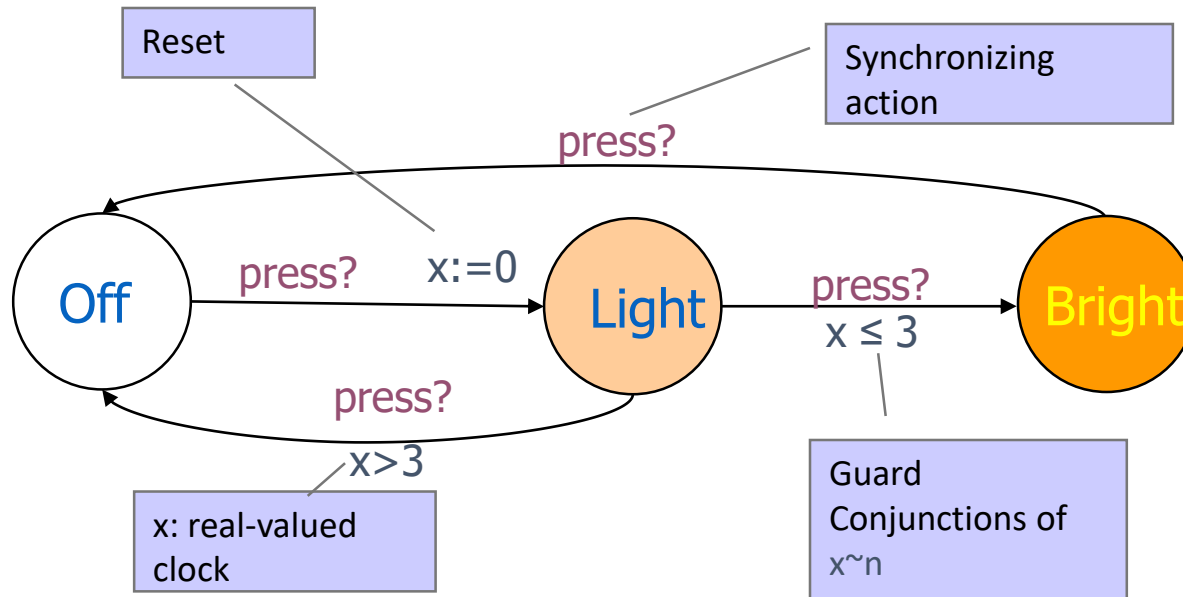


Dumb Light Control

Solution 2: Add real-valued clock x to model the timing requirements $[[quickly]] = x \leq 3$



Timed Automata



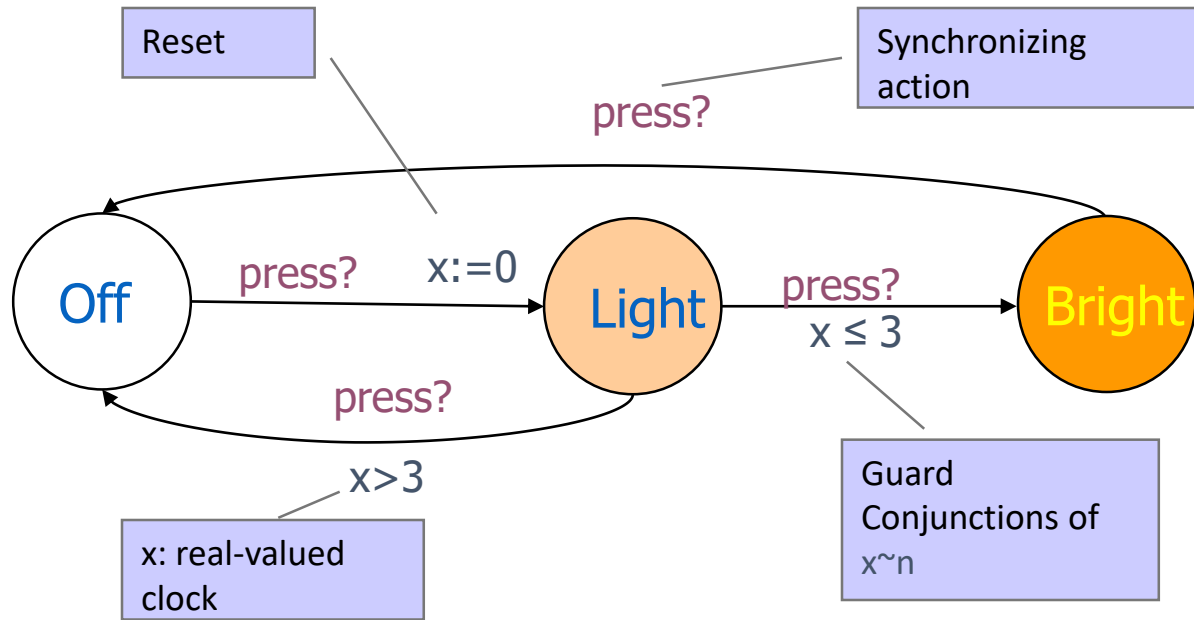
States:

(location , $x=v$) where $v \in \mathbf{R}$

Transitions:

(Off , $x=0$)

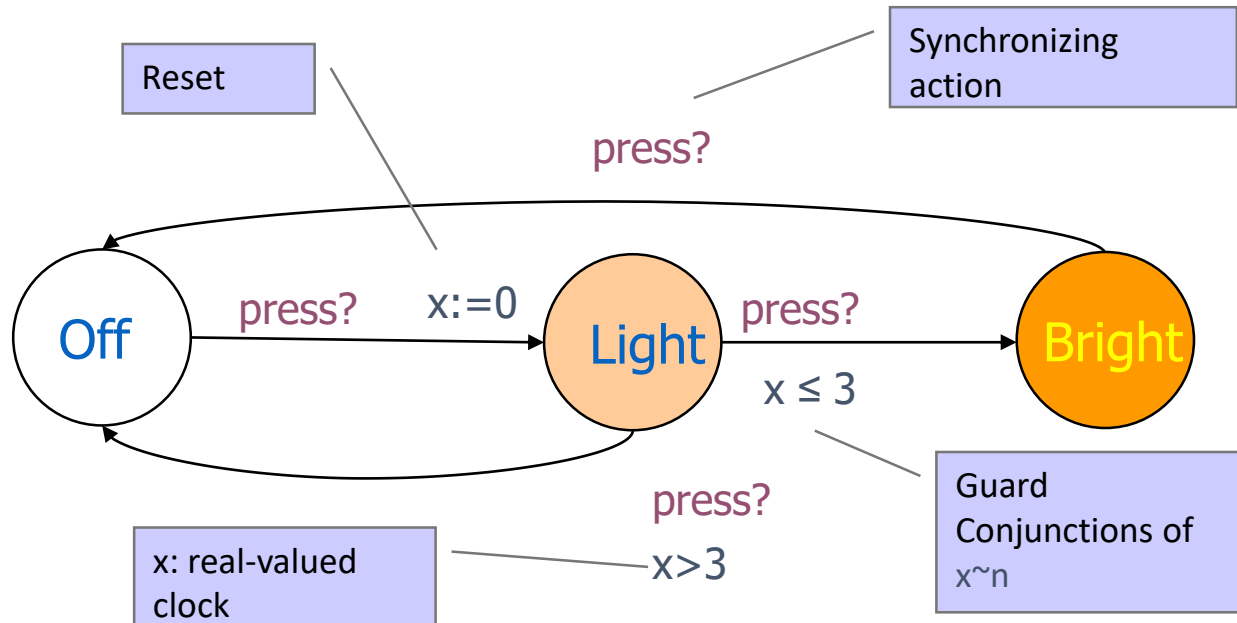
Timed Automata



States:
 (location , $x=v$) where $v \in \mathbf{R}$

Transitions:
 delay 4.32 (Off , $x=0$)
 \rightarrow (Off , $x=4.32$)

Timed Automata



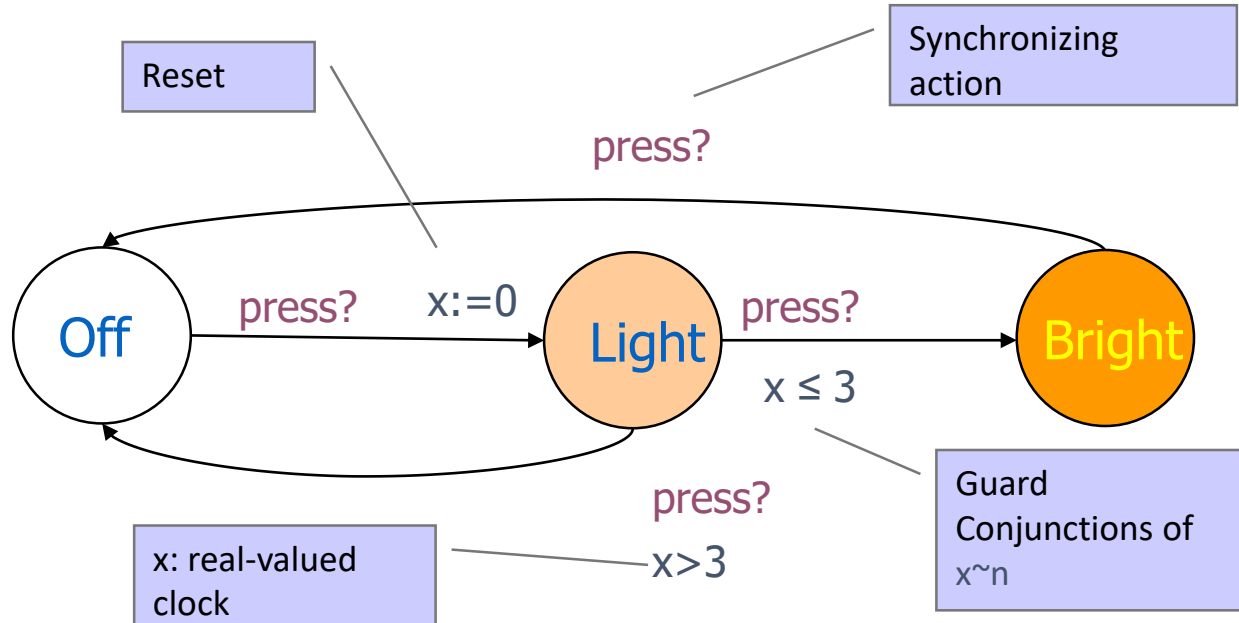
States:

(location , $x=v$) where $v \in \mathbf{R}$

Transitions:

$(\text{Off} , x=0)$
 delay 4.32 $\rightarrow (\text{Off} , x=4.32)$
 $\text{press?} \rightarrow (\text{Light} , x=0)$

Timed Automata

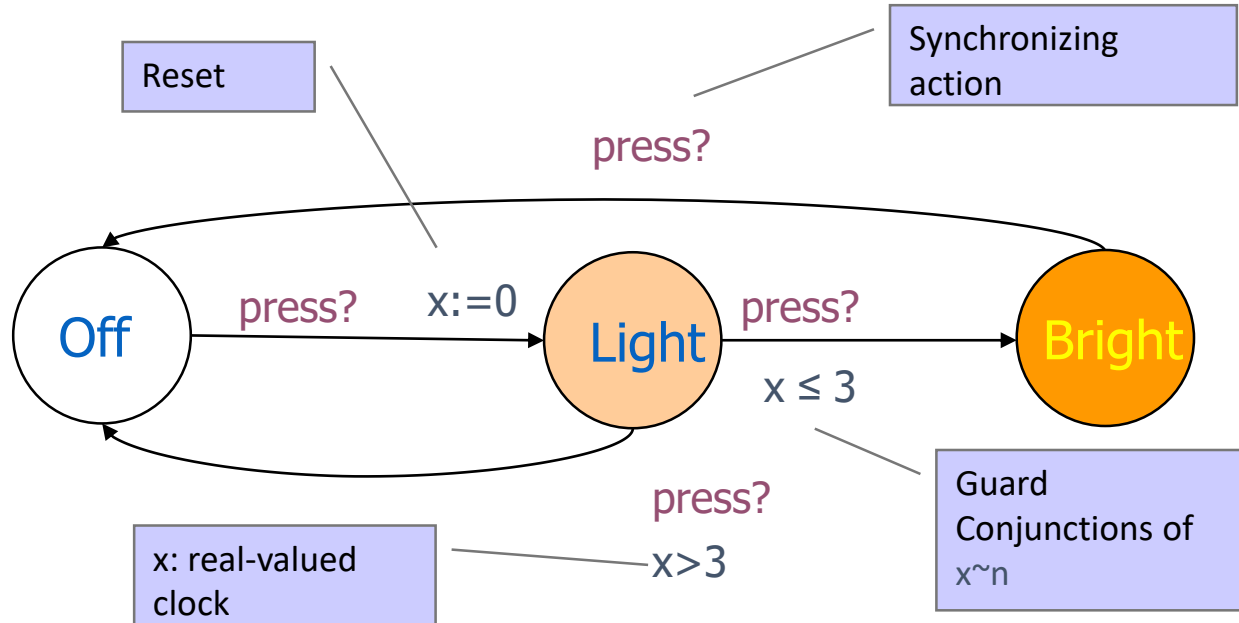


Transitions:

	(Off , x=0)
delay 4.32	→ (Off , x=4.32)
press?	→ (Light , x=0)
delay 2.51	→ (Light , x=2.51)

States:
 (location , x=v) where $v \in \mathbf{R}$

Timed Automata



States:

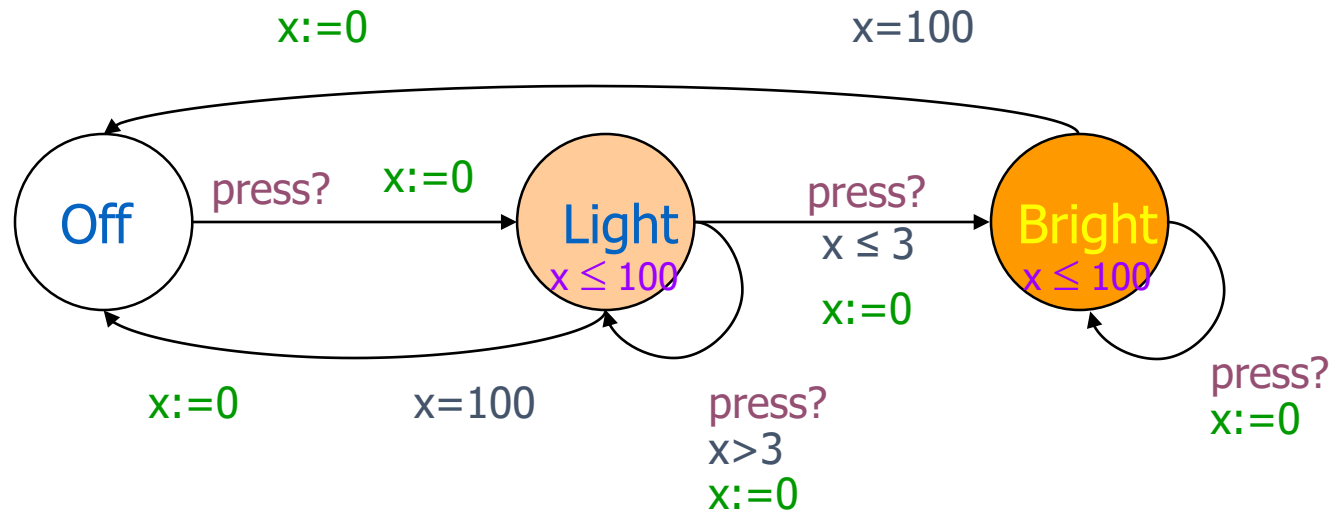
(location , $x=v$) where $v \in \mathbf{R}$

Transitions:

(Off , $x=0$)
 delay 4.32 \rightarrow (Off , $x=4.32$)
`press?` \rightarrow (Light , $x=0$)
 delay 2.51 \rightarrow (Light , $x=2.51$)
`press?` \rightarrow (Bright , $x=2.51$)

Intelligent Light Control

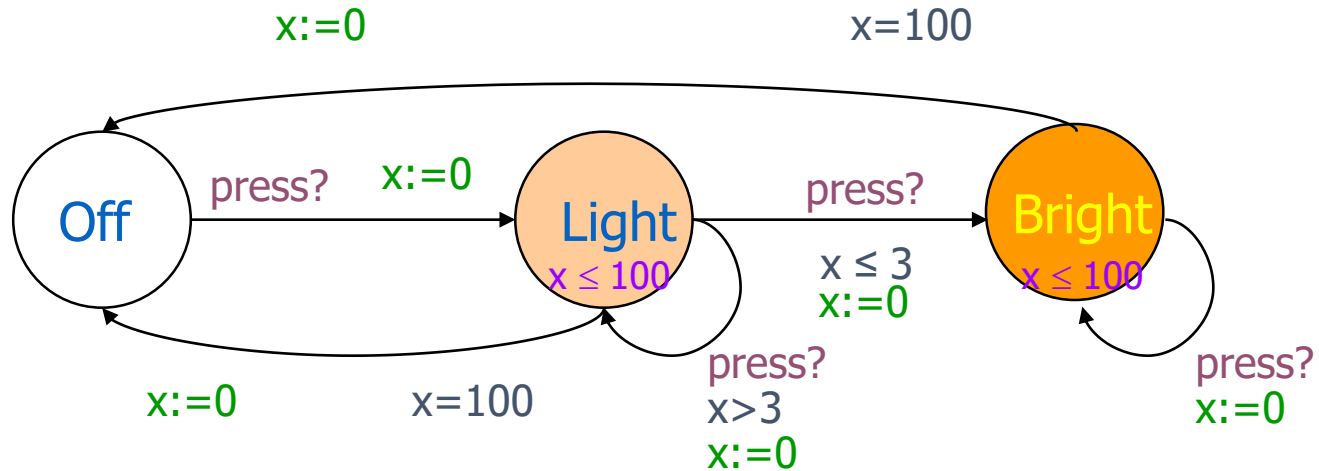
Requirement: : automatically switch light off after 100 time units



Upper time bound is specified using **invariants**

Intelligent Light Control

Using Invariants



Transitions:

$(\text{Off}, x=0)$
 delay 4.32 $\rightarrow (\text{Off}, x=4.32)$
 press? $\rightarrow (\text{Light}, x=0)$
 delay 4.51 $\rightarrow (\text{Light}, x=4.51)$
 press? $\rightarrow (\text{Light}, x=0)$
 delay 100 $\rightarrow (\text{Light}, x=100)$
 τ $\rightarrow (\text{Off}, x=0)$

Note:

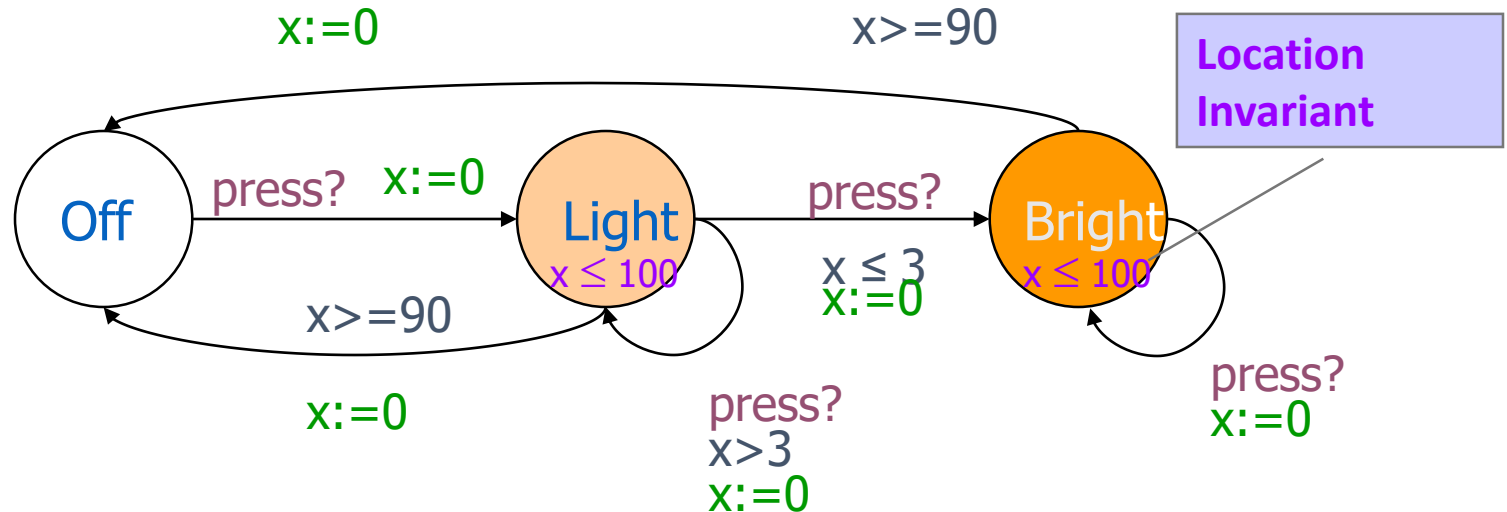
$(\text{Light}, x=0)$ delay 103 \rightarrow

Invariants
ensures
progress

Intelligent Light Control

If requirements include **uncertainty**:

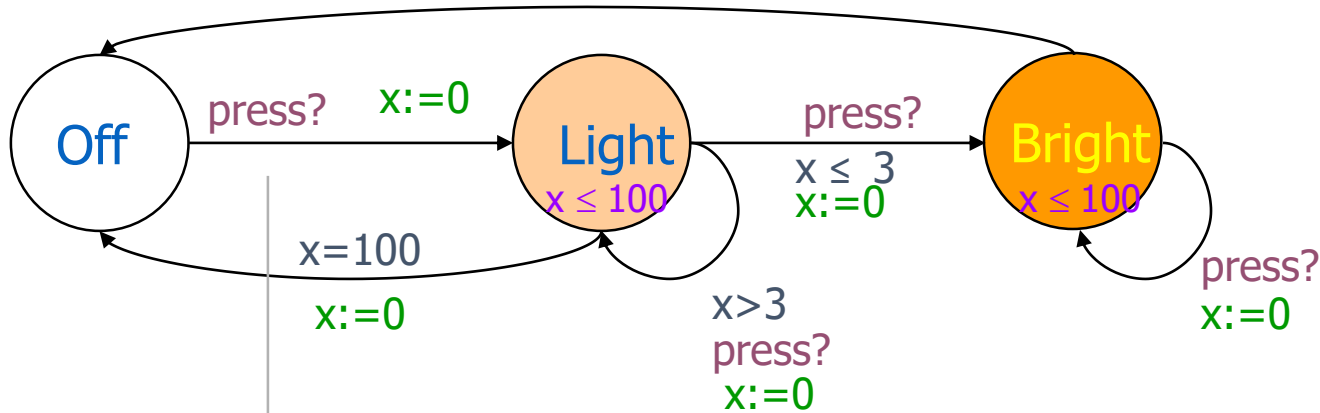
Automatically switch light off **between** 90-100 time units after switching on.



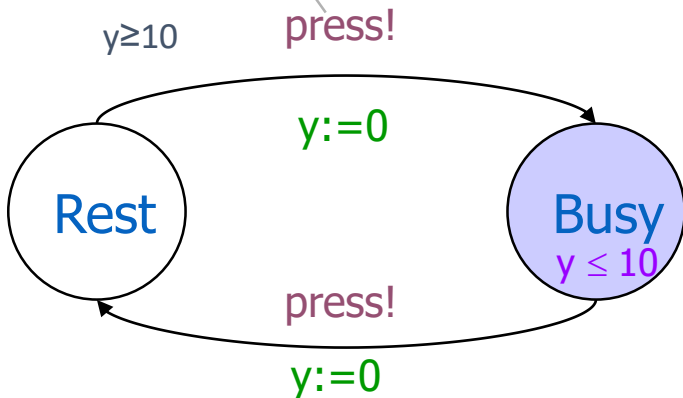
Light Controller || User

$x:=0$

$x=100$



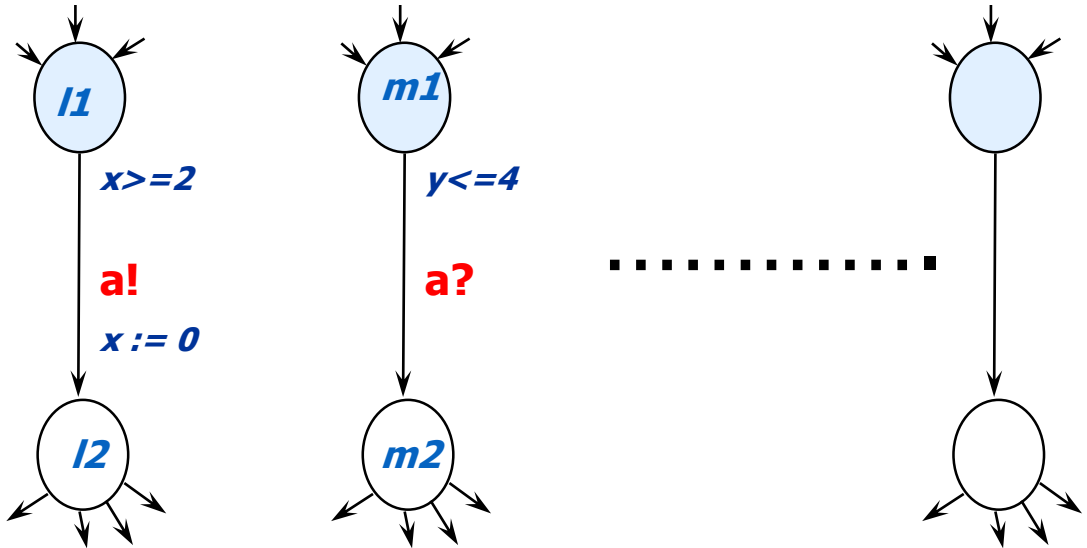
Synchronization



Transitions:

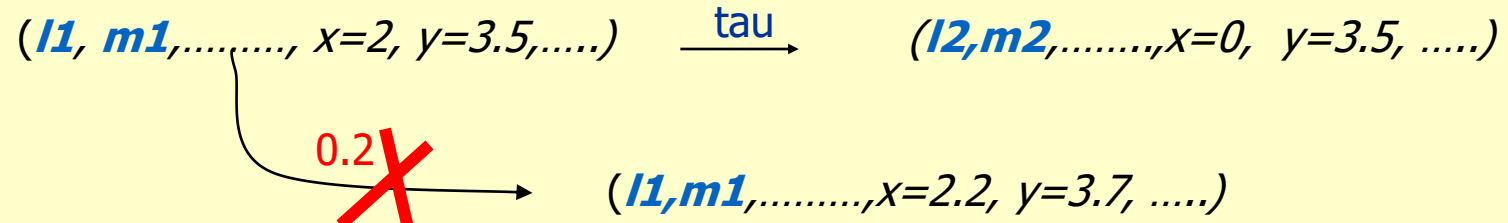
- (Off, Rest, $x=0$, $y=0$)
- delay 20 → (Off, Rest, $x=20$, $y=20$)
- press?! → (Light, Busy, $x=0$, $y=0$)
- delay 2 → (Light, Busy, $x=2$, $y=2$)
- press?! → (Bright, Rest, $x=0$, $y=0$)

Networks of Timed Automata (a'la CCS)



Two-way synchronization on *complementary* actions.
Closed Systems!

Example transitions



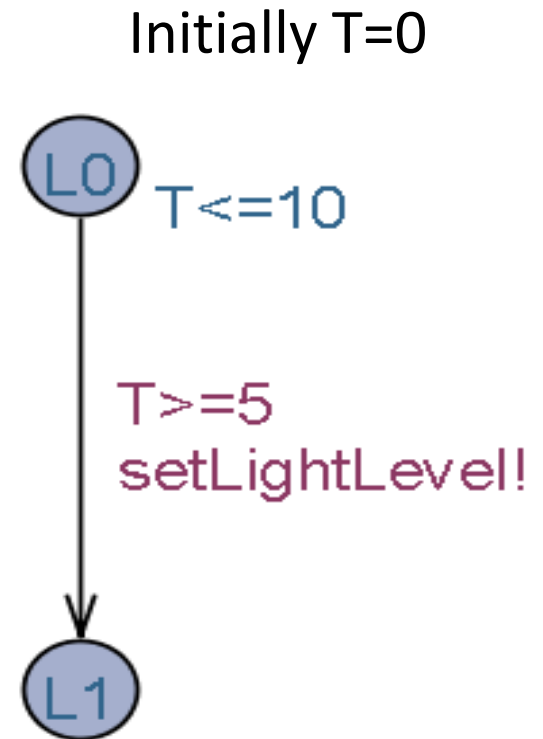
If **a** URGENT CHANNEL

Timing Uncertainty

- Unpredictable or variable timing
 - response time,
 - computation time
 - transmission time etc:

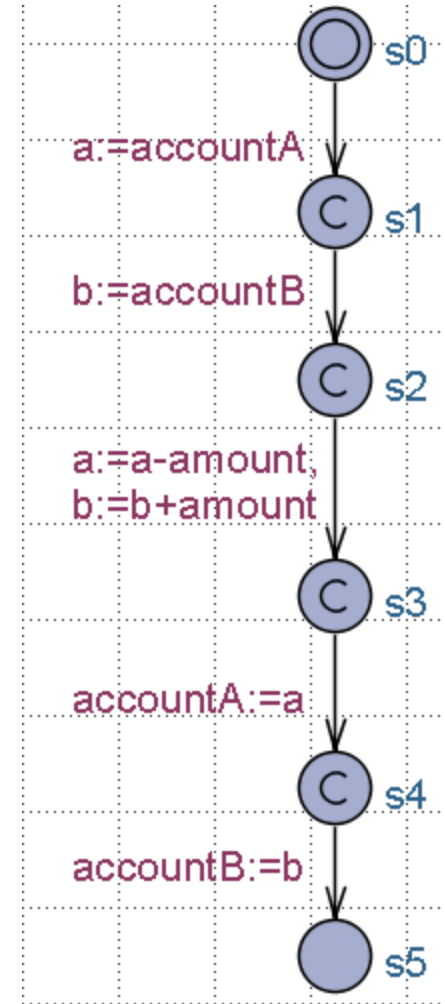
Example:

Light level must be adjusted
between 5 and 10 time units



Committed Locations

- Locations marked **C**
 - **No delay** in committed location.
 - No interleaving with parallel transitions
- Handy to model atomic sequences
- The use of committed locations reduces the number of states in a model, and allows for more space and time efficient analysis.
- S0 to s5 executed atomically



Urgent Channels and Locations

- Locations marked **U**
 - ***No delay*** like in committed location.
 - But Interleaving permitted
- Channels declared “**urgent chan**”
 - Time doesn't elapse when a synchronization is possible on a pair of urgent channels
 - Interleaving allowed

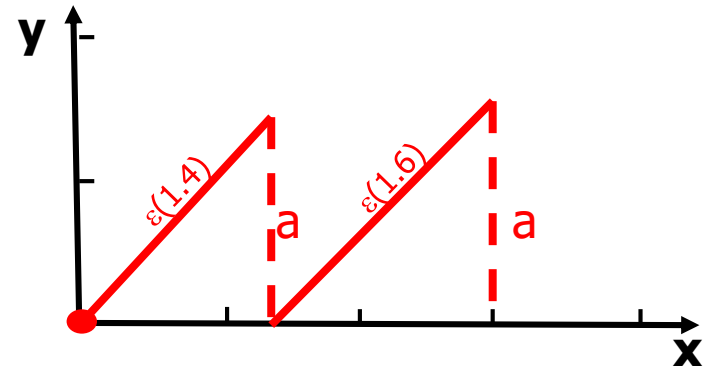
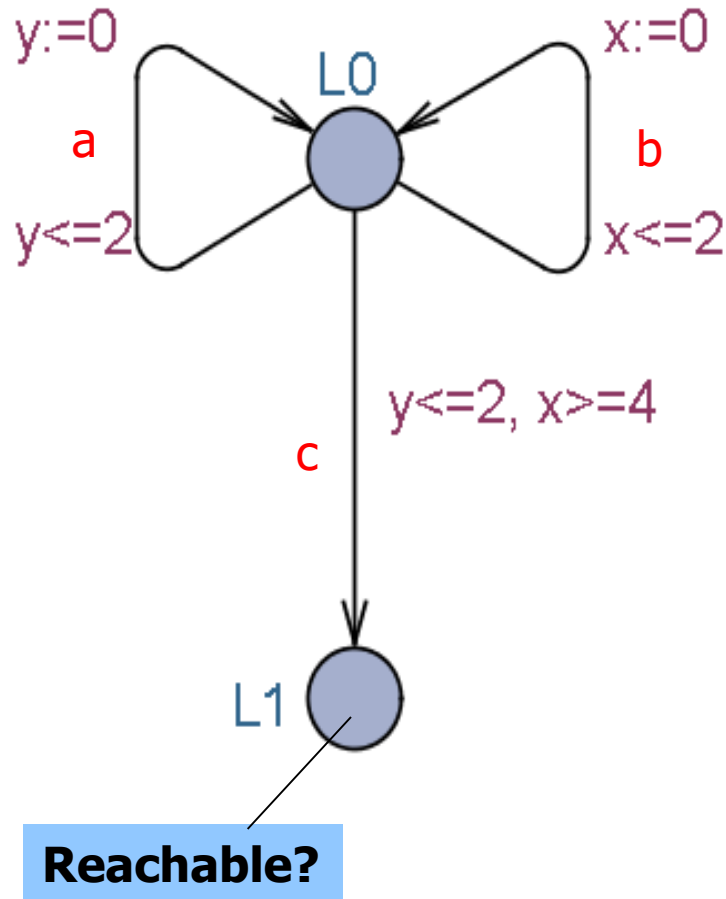
Broad-casts

- broadcast channel:
 - sending: output chan!
 - every automaton that listens to join moves on
 - ie. every automaton with enabled “chan?” transition moves in one step
 - may be zero! Listeners, sender can progress anyway

Other Uppaal features

- Bounded domains
 - `int [1..4] a;`
- C-like data-structures and user defined functions in declaration section
 - structs, arrays, and typedef
- non-deterministic assignment:
 - `select a:T // select a random value from T`
- `forall`, `exists` in expressions
- Scalar sets (for giving unique ID's)
- Process and channel **priorities**
- Value passing (emulation)

Semantics: Timed traces



$(L0, x=0, y=0)$

$\rightarrow_{\epsilon(1.4)}$

$(L0, x=1.4, y=1.4)$

\rightarrow_a

$(L0, x=1.4, y=0)$

$\rightarrow_{\epsilon(1.6)}$

$(L0, x=3.0, y=1.6)$

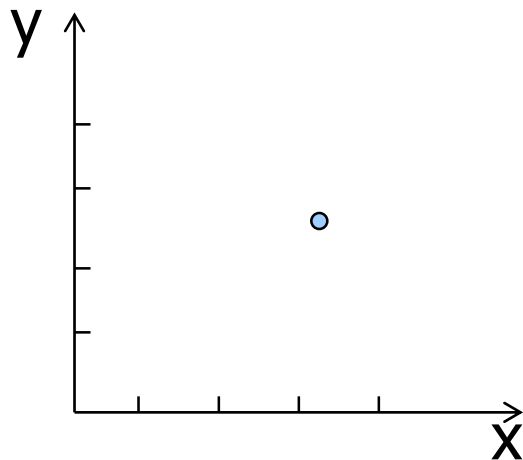
\rightarrow_a

$(L0, x=3.0, y=0)$

From explicit clock values to zones *(from infinite to finite)*

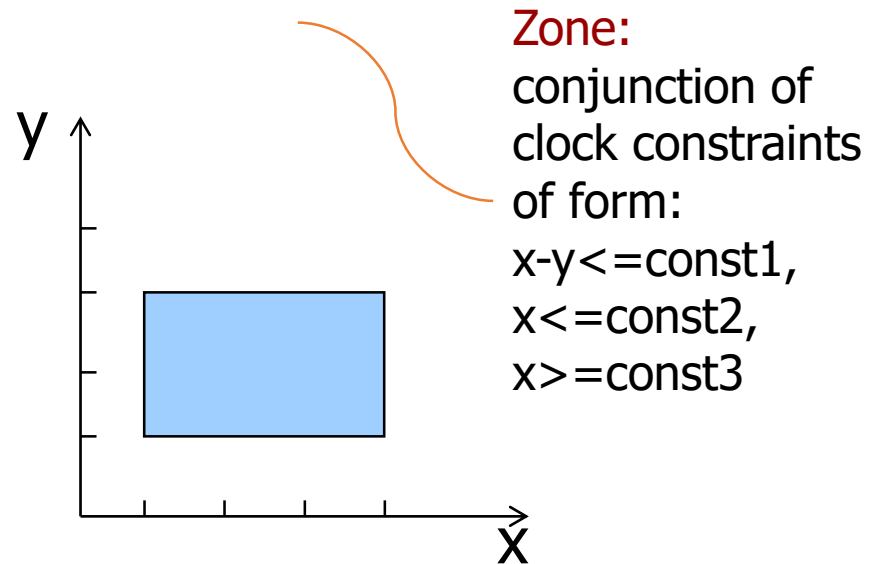
Explicit state

$(n, x=3.2, y=2.5)$

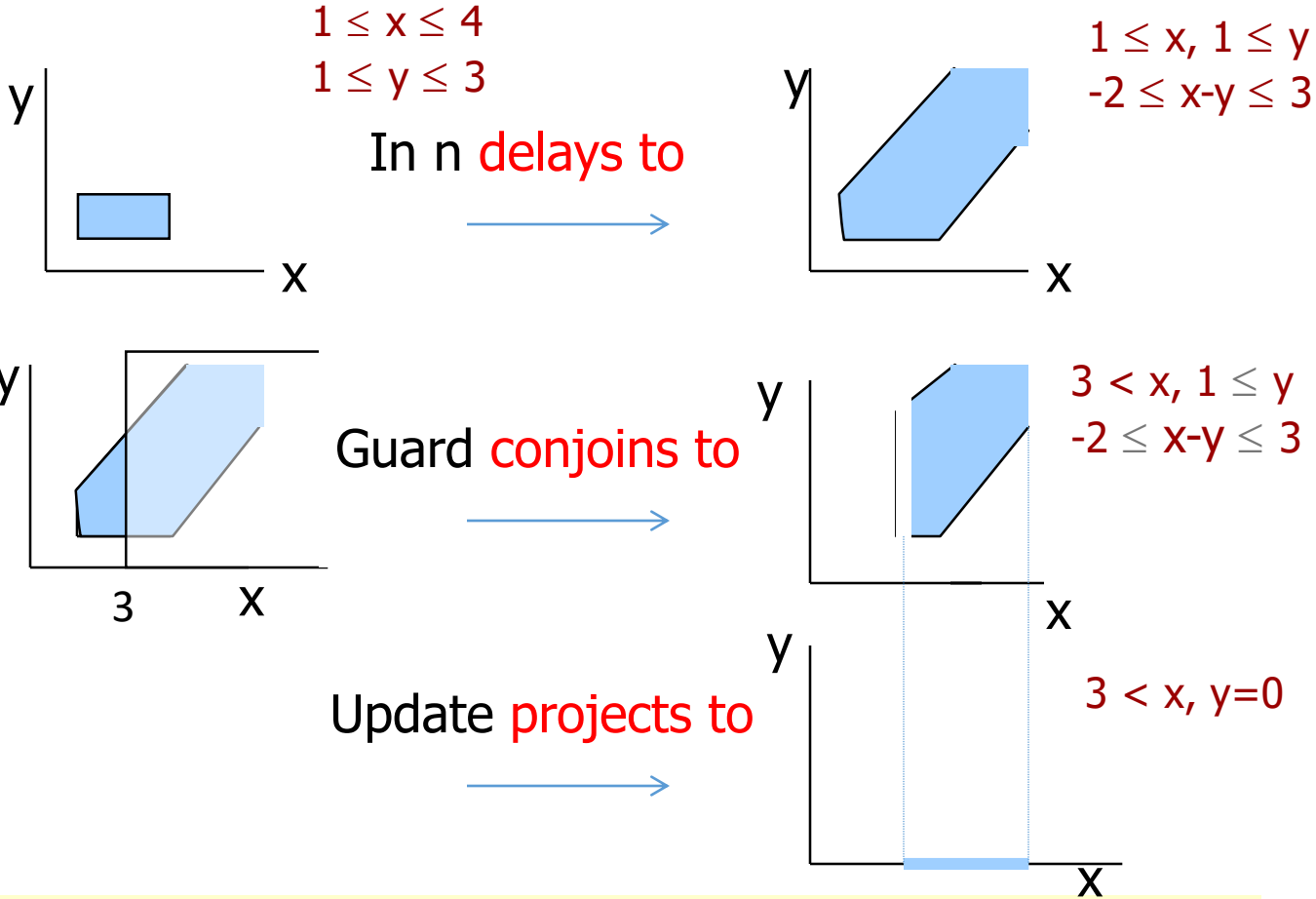
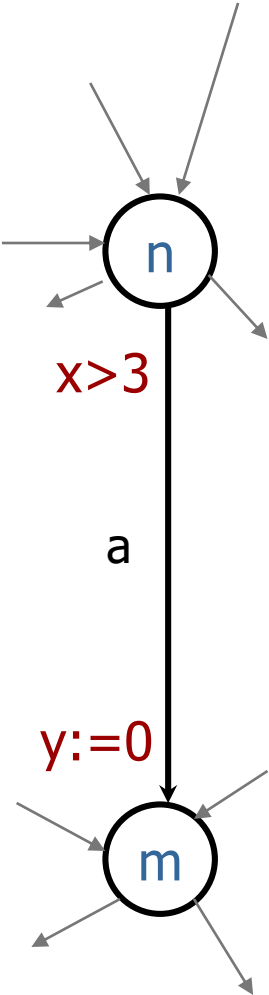


Symbolic state (set)

$(n, 1 \leq x \leq 4, 1 \leq y \leq 3)$

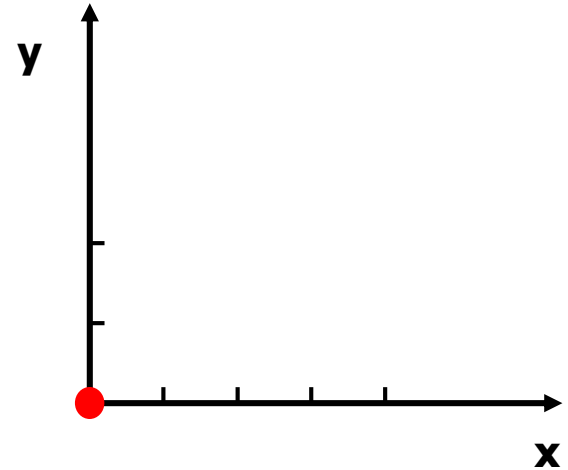
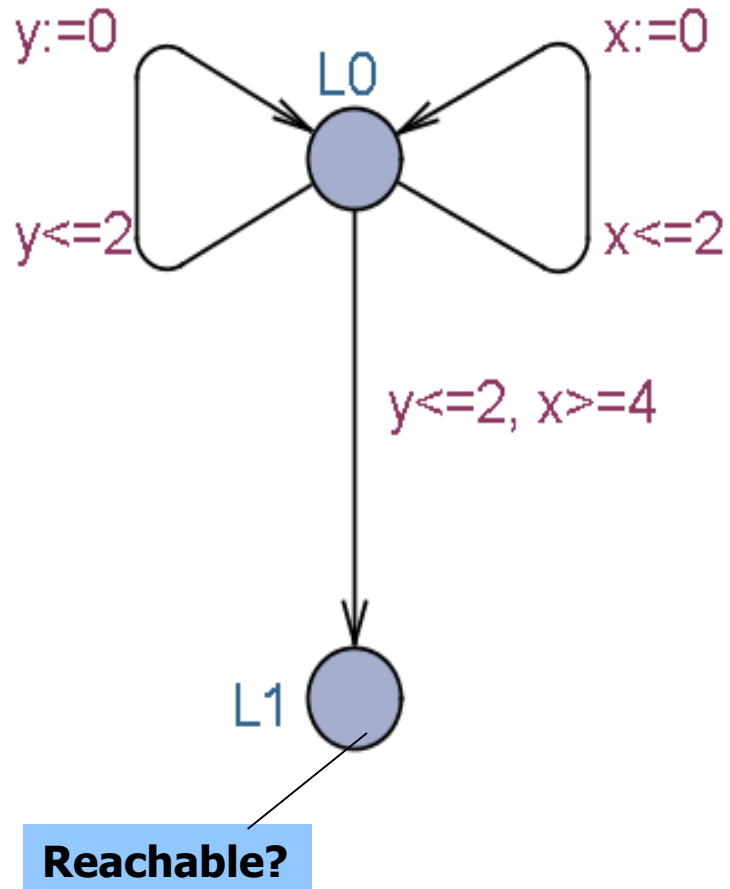


Symbolic Transitions

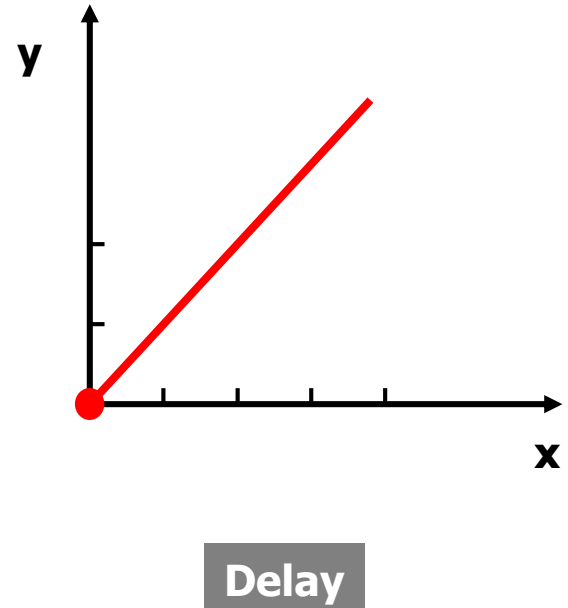
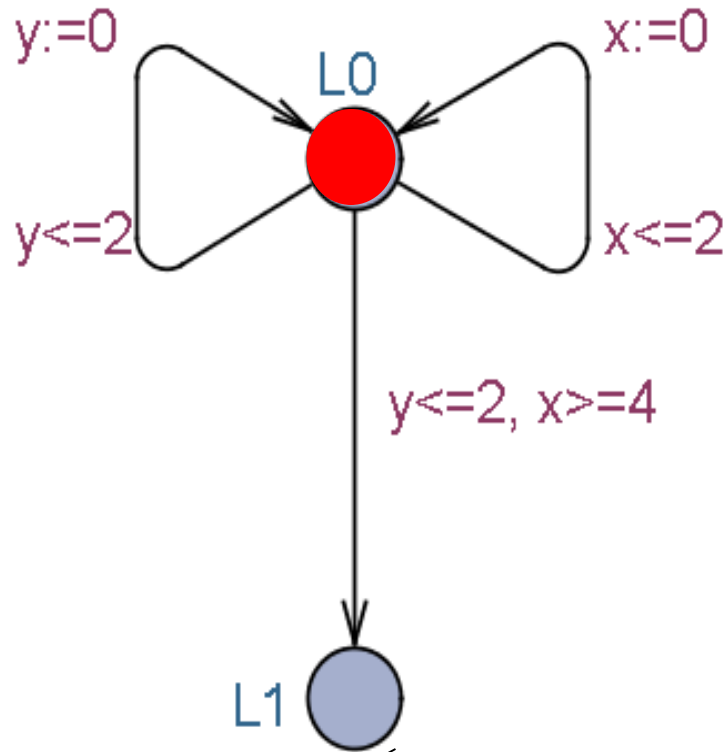


Thus $(n, 1 \leq x \leq 4, 1 \leq y \leq 3) \xrightarrow{a} (m, 3 < x, y = 0)$

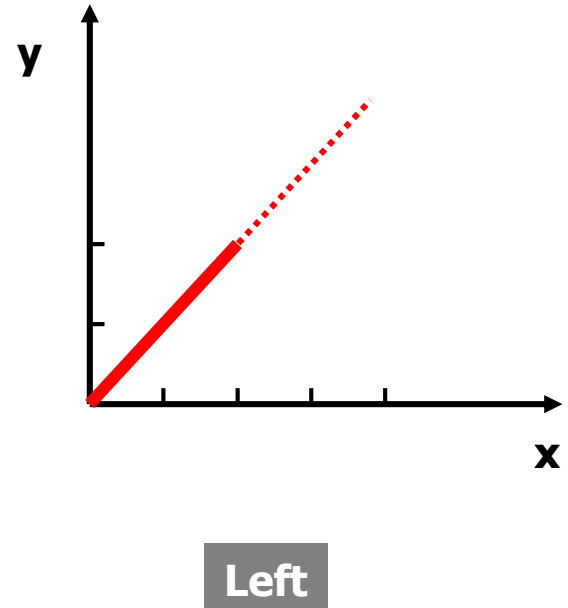
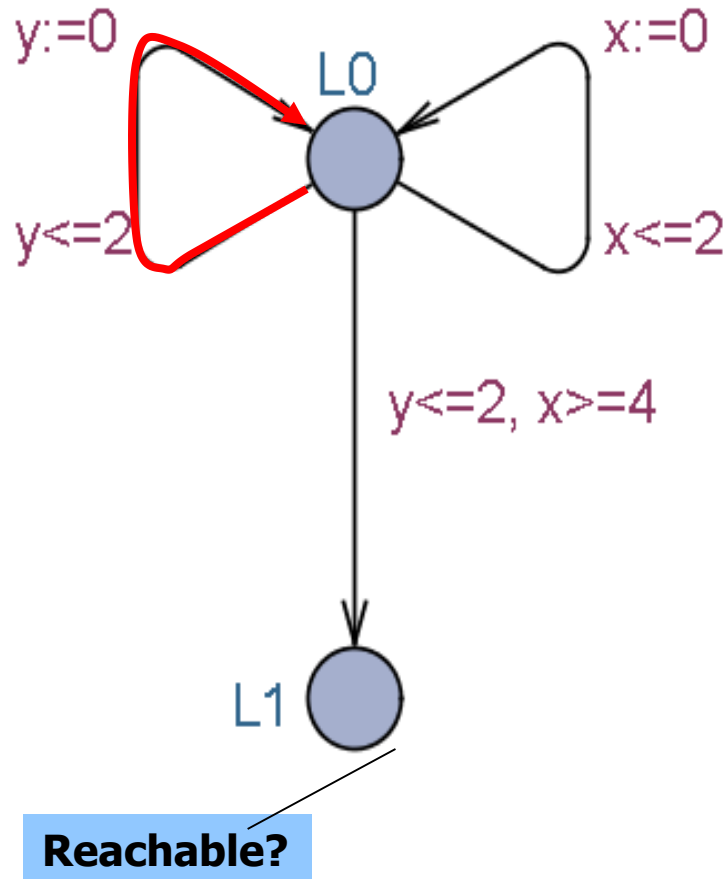
Symbolic state exploration



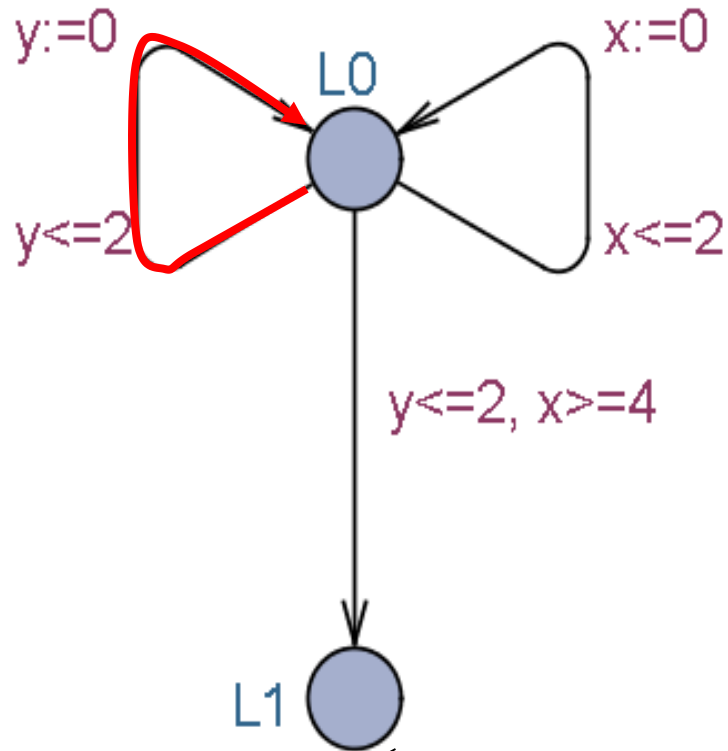
Symbolic Exploration



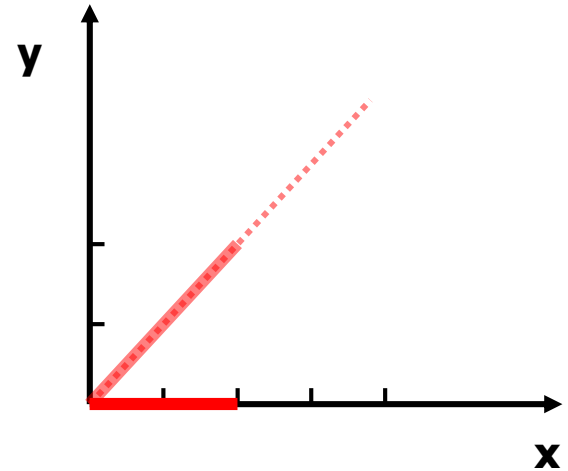
Symbolic Exploration



Symbolic Exploration

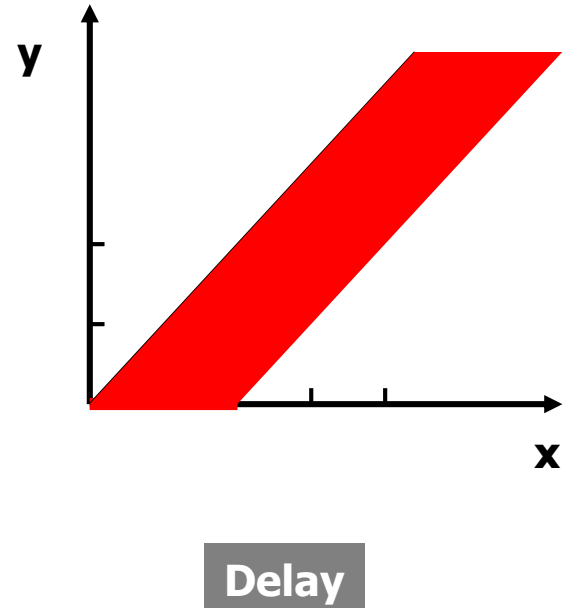
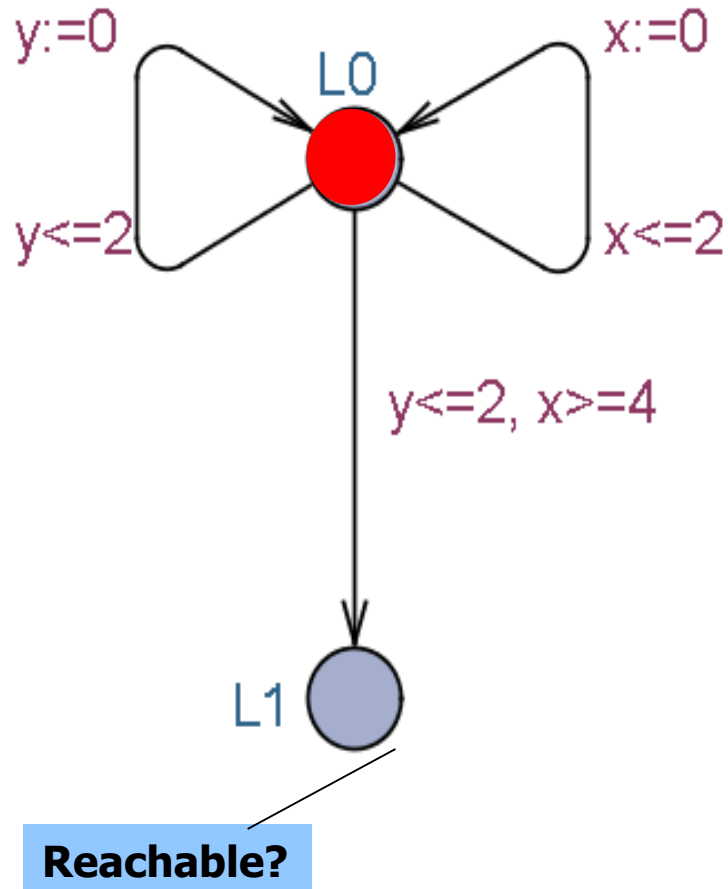


Reachable?

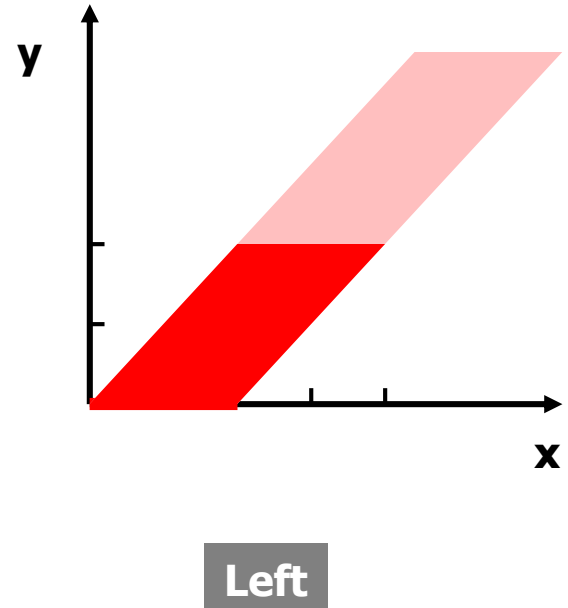
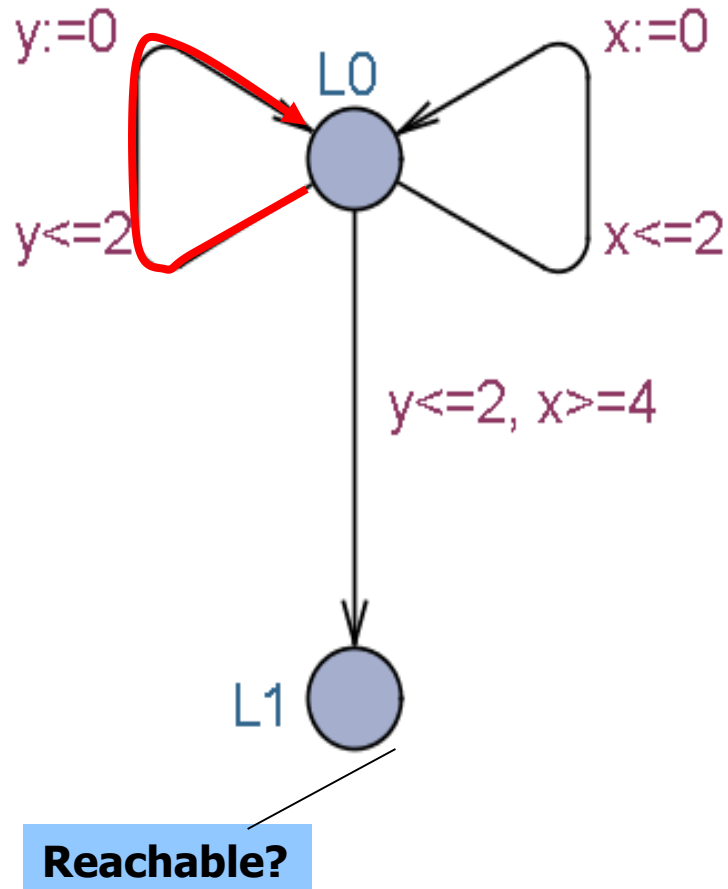


Left

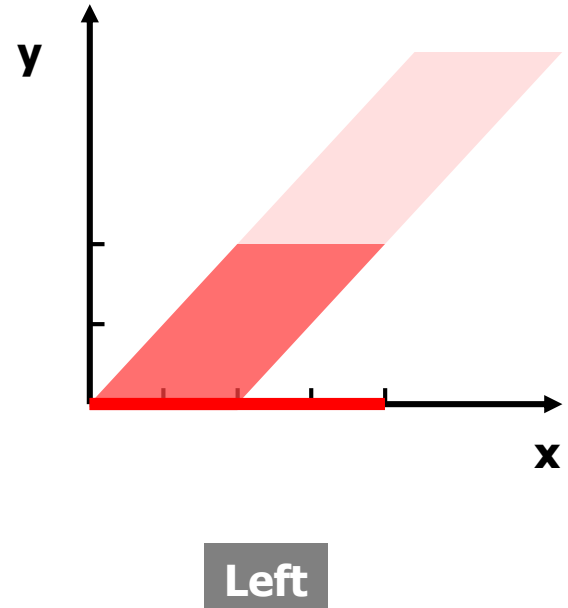
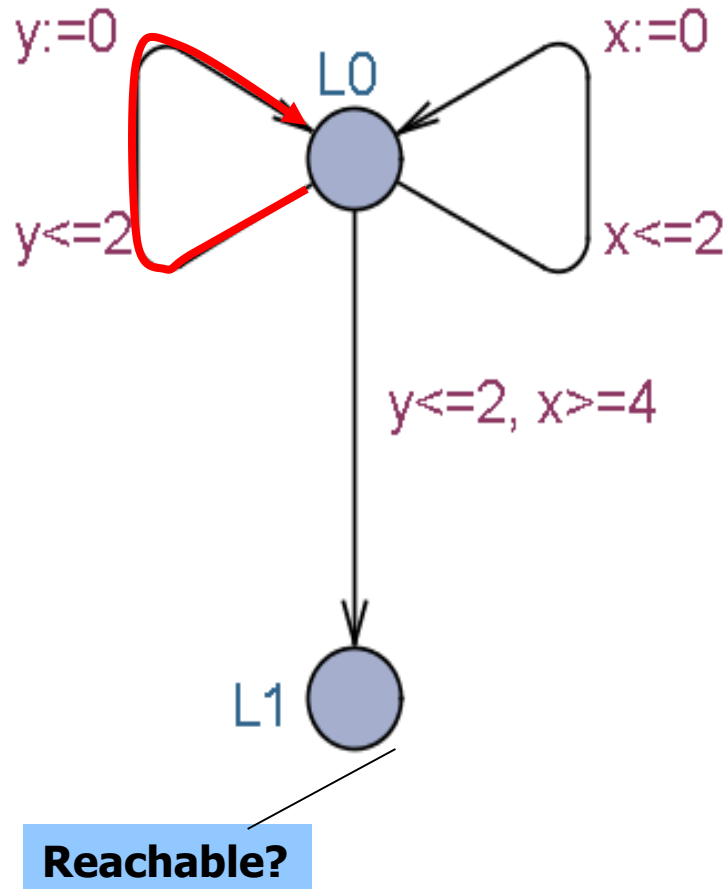
Symbolic Exploration



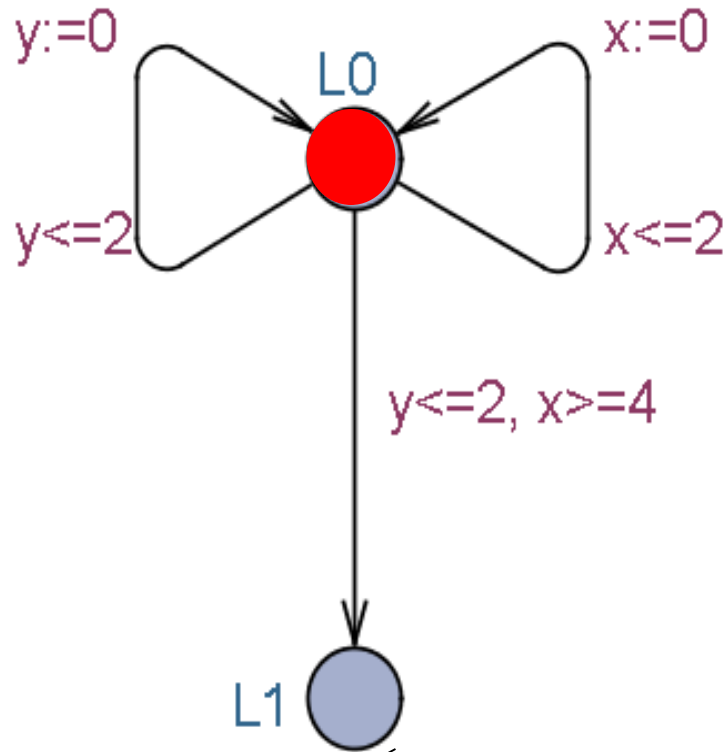
Symbolic Exploration



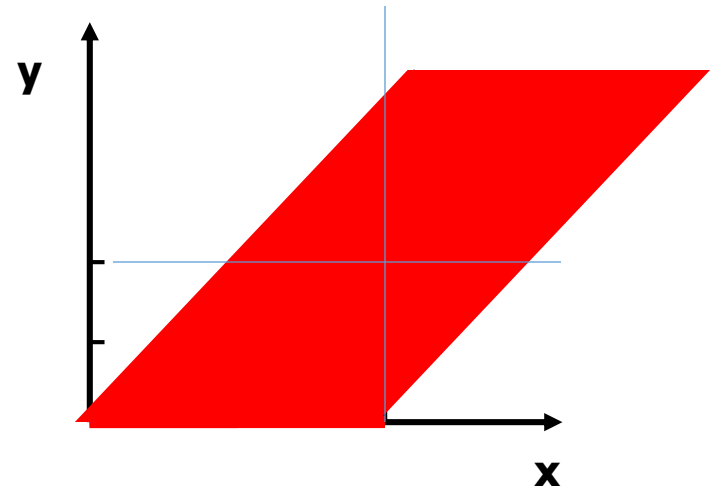
Symbolic Exploration



Symbolic Exploration

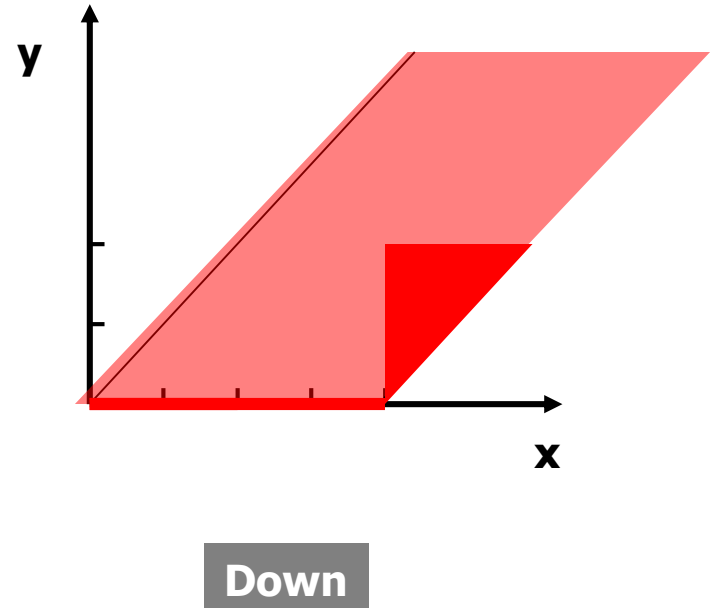
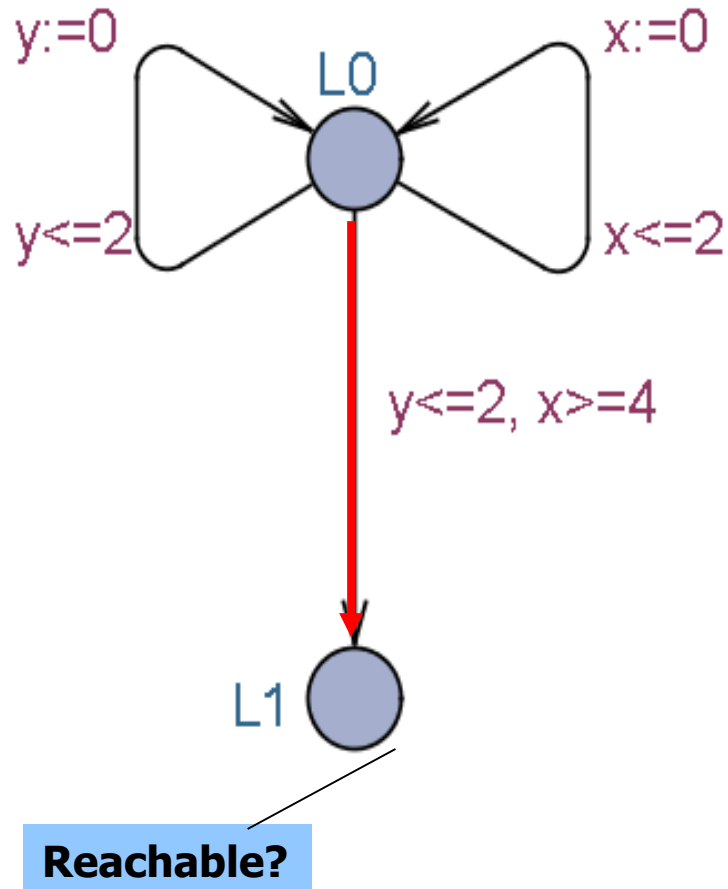


Reachable?



Delay

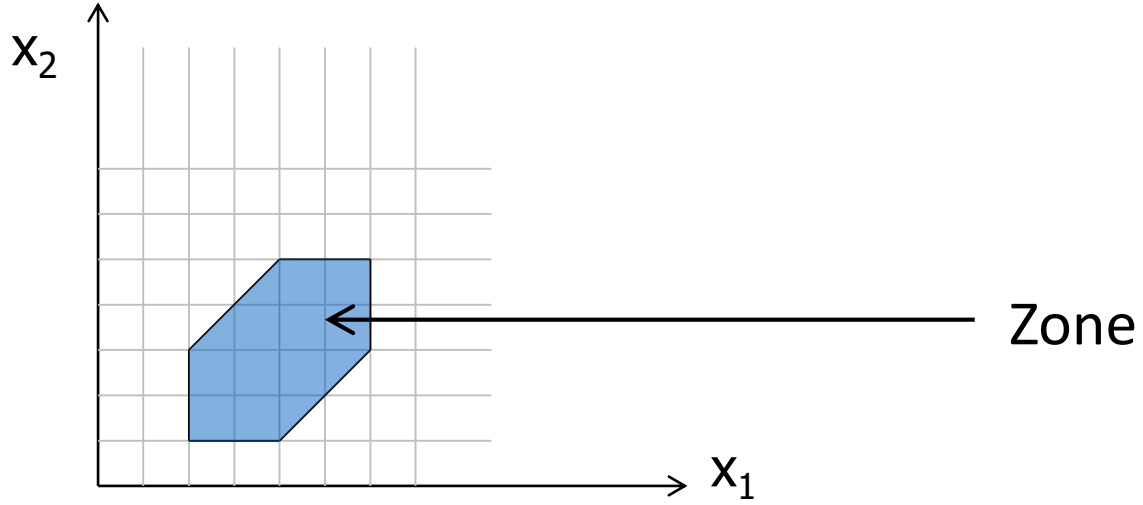
Symbolic Exploration



Difference Bound Matrices

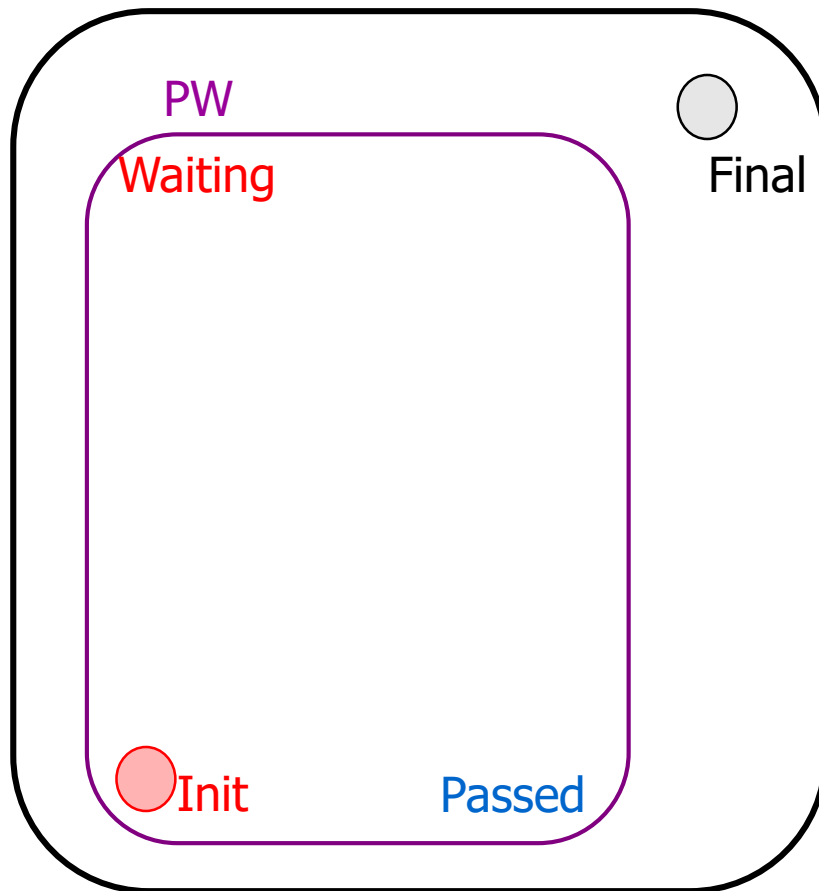
$x_0 - x_0 \leq 0$	$x_0 - x_1 \leq -2$	$x_0 - x_2 \leq -1$
$x_1 - x_0 \leq 6$	$x_1 - x_1 \leq 0$	$x_1 - x_2 \leq 3$
$x_2 - x_0 \leq 5$	$x_2 - x_1 \leq 1$	$x_2 - x_2 \leq 0$

$$x_i - x_j \leq C_{ij}$$



Forward Reachability Algorithm

Init \rightarrow Final ?



INITIAL $\text{Passed} := \emptyset;$
 $\text{Waiting} := \{(n_0, Z_0)\}$

REPEAT

pick (n, Z) in Waiting

if $(n, Z) = \text{Final}$ return true

for all $(n, Z) \rightarrow (n', Z')$:

if for some (n', Z'') $Z' \subseteq Z''$ continue

else add (n', Z') to Waiting

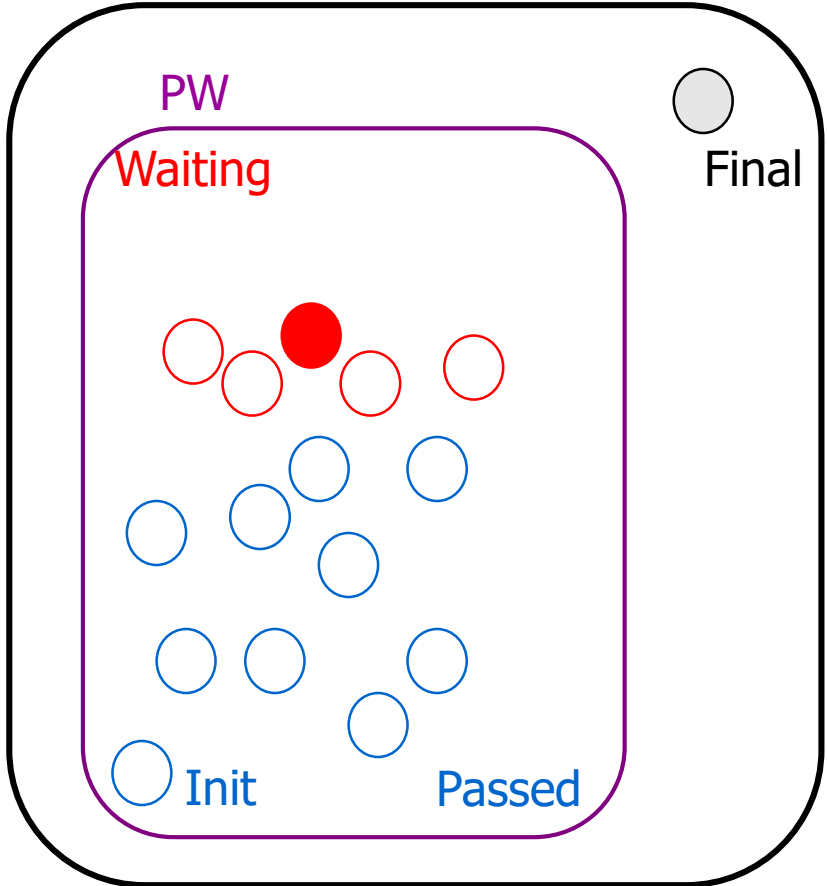
move (n, Z) to Passed

UNTIL $\text{Waiting} = \emptyset$

return false

Forward Reachability Algorithm

Init -> Final ?



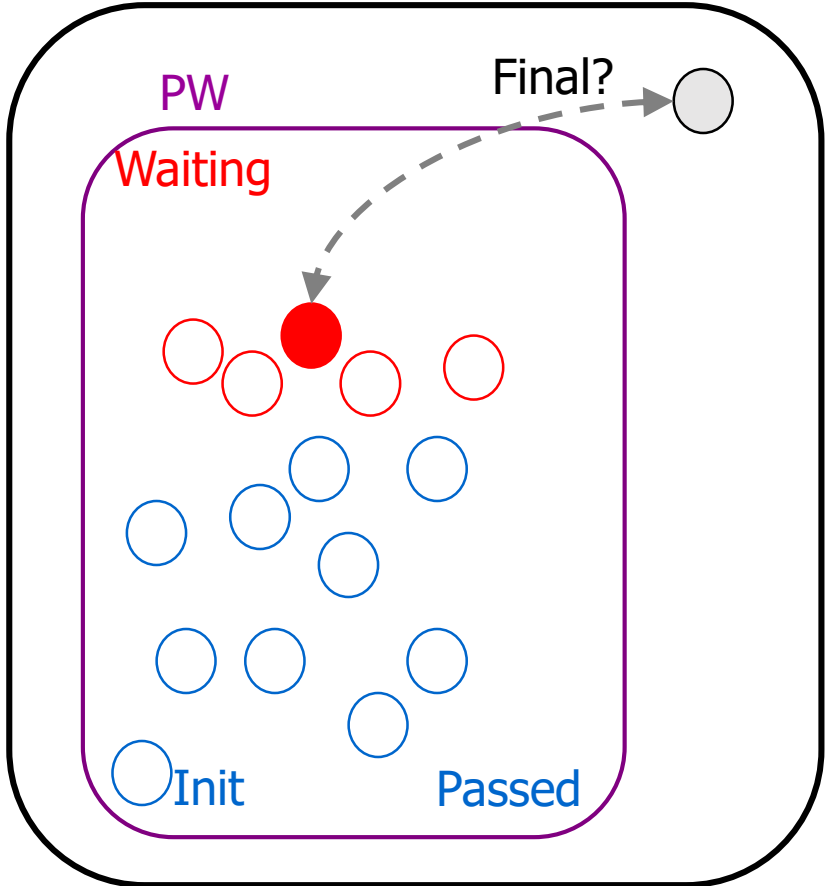
INITIAL **Passed** := \emptyset ;
Waiting := $\{(n_0, Z_0)\}$

REPEAT
 pick (n, Z) in **Waiting**
 if $(n, Z) = \text{Final}$ return true
 for all $(n, Z) \rightarrow (n', Z')$:
 if for some (n', Z'') $Z' \subseteq Z''$ continue
 else add (n', Z') to **Waiting**
 move (n, Z) to **Passed**

UNTIL **Waiting** = \emptyset
 return false

Forward Reachability Algorithm

Init -> Final ?



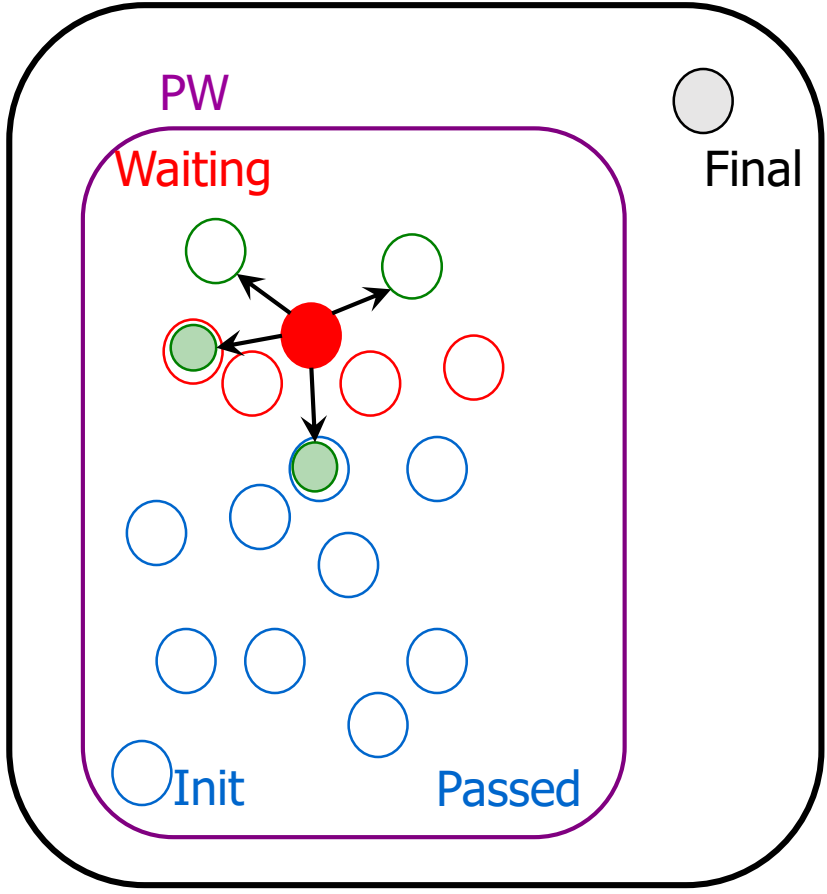
INITIAL **Passed** := \emptyset ;
Waiting := $\{(n_0, Z_0)\}$

REPEAT
 pick (n, Z) in **Waiting**
 if $(n, Z) = \text{Final}$ return true
 for all $(n, Z) \rightarrow (n', Z')$:
 if for some (n', Z'') $Z' \subseteq Z''$ continue
 else add (n', Z') to **Waiting**
 move (n, Z) to **Passed**

UNTIL **Waiting** = \emptyset
 return false

Forward Reachability Algorithm

Init -> Final ?



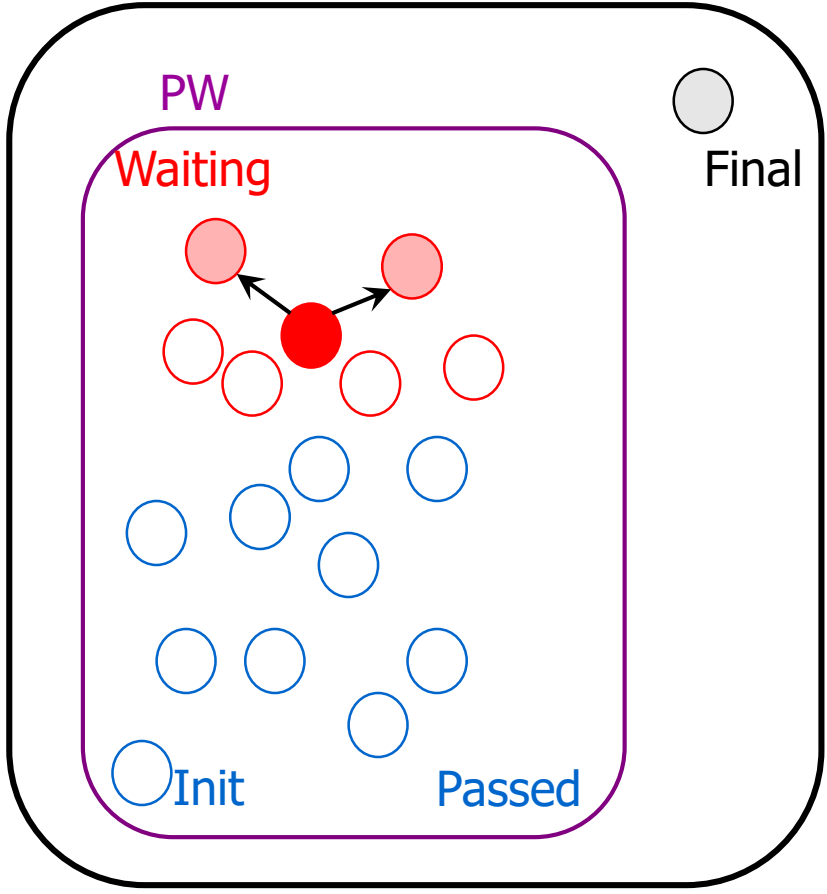
INITIAL **Passed** := \emptyset ;
Waiting := $\{(n_0, Z_0)\}$

REPEAT
 pick (n, Z) in **Waiting**
 if $(n, Z) = \text{Final}$ return true
 for all $(n, Z) \rightarrow (n', Z')$:
 if for some (n', Z'') $Z' \subseteq Z''$ continue
 else add (n', Z') to **Waiting**
 move (n, Z) to **Passed**

UNTIL **Waiting** = \emptyset
 return false

Forward Reachability Algorithm

Init -> Final ?



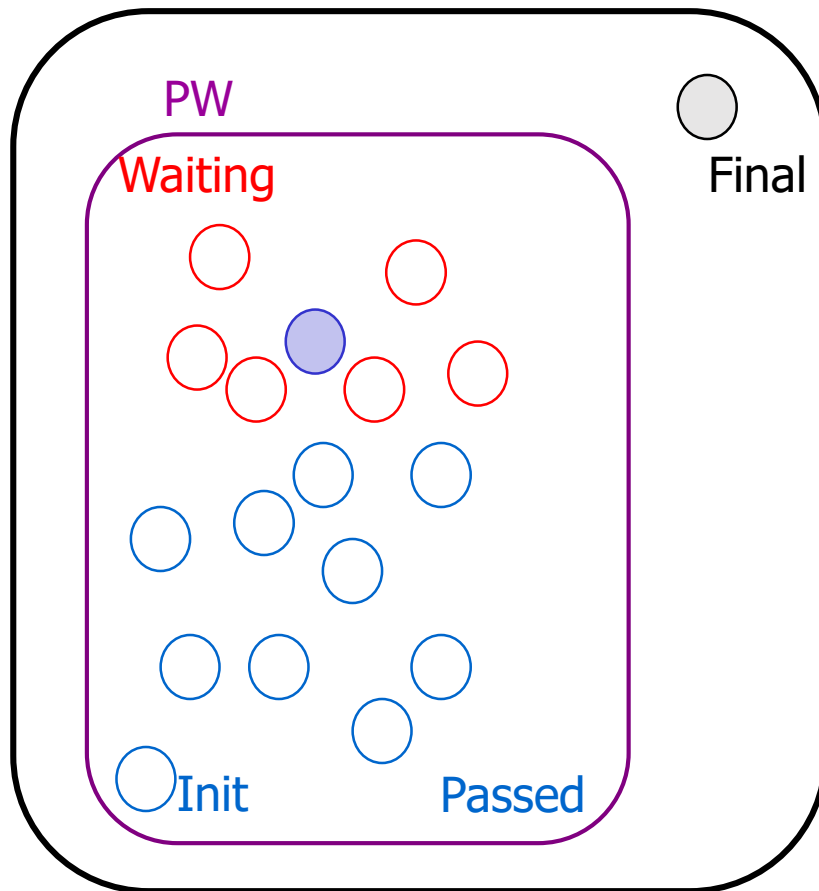
INITIAL **Passed** := \emptyset ;
Waiting := $\{(n_0, Z_0)\}$

REPEAT
 pick (n, Z) in **Waiting**
 if $(n, Z) = \text{Final}$ return true
 for all $(n, Z) \rightarrow (n', Z')$:
 if for some (n', Z'') $Z' \subseteq Z''$ continue
 else add (n', Z') to **Waiting**
 move (n, Z) to **Passed**

UNTIL **Waiting** = \emptyset
 return false

Forward Reachability Algorithm

Init \rightarrow Final ?



INITIAL $\text{Passed} := \emptyset;$
 $\text{Waiting} := \{(n_0, Z_0)\}$

REPEAT

pick (n, Z) in Waiting

if $(n, Z) = \text{Final}$ return true

for all $(n, Z) \rightarrow (n', Z')$:

if for some (n', Z') $Z' \subseteq Z''$ continue

else add (n', Z') to Waiting

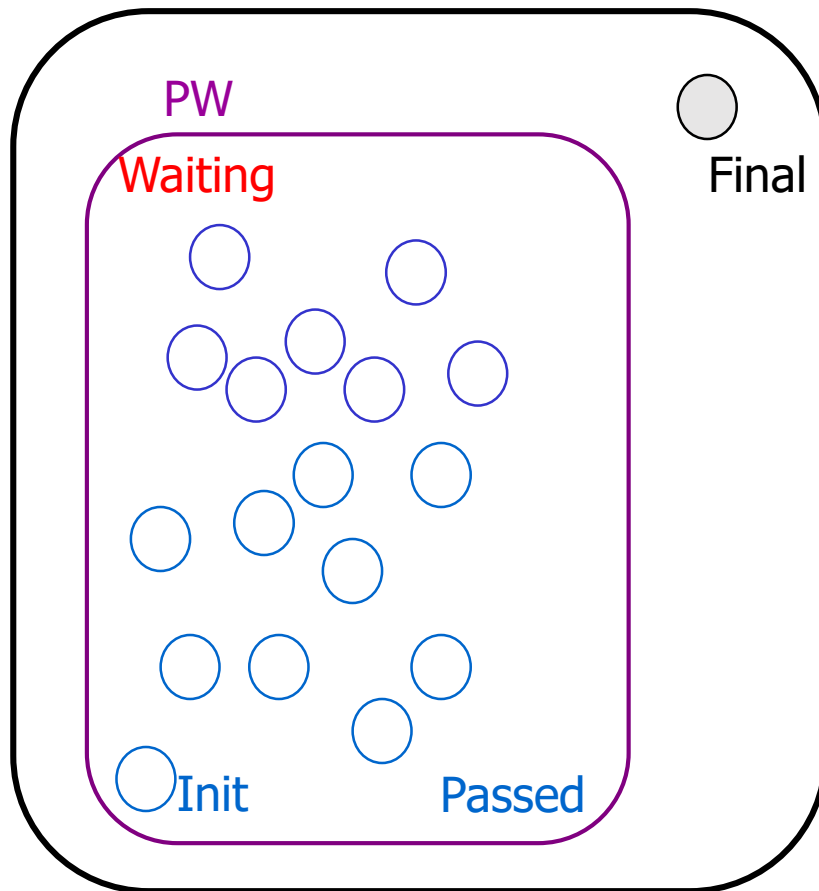
move (n, Z) to Passed

UNTIL $\text{Waiting} = \emptyset$

return false

Forward Reachability Algorithm

Init \rightarrow Final ?



INITIAL $\text{Passed} := \emptyset;$
 $\text{Waiting} := \{(n_0, Z_0)\}$

REPEAT

pick (n, Z) in Waiting

if $(n, Z) = \text{Final}$ return true

for all $(n, Z) \rightarrow (n', Z')$:

if for some (n', Z'') $Z' \subseteq Z''$ continue

else add (n', Z') to Waiting

move (n, Z) to Passed

UNTIL $\text{Waiting} = \emptyset$

return false

Specification (Query) Language

UPPAAL Property Specification Language

- **A[] p**
- **A<> p**

always
inevitable

- **E<> p**
- **E[] p**
- **P --> q**

Possible
potentially always
leads-to

process location data guards clock guards

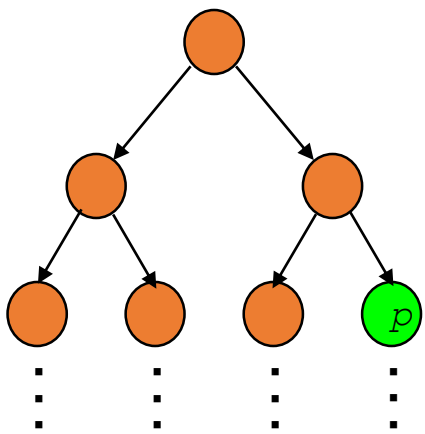
```

p ::= a.l | gd | gc | p and p |
      p or p | not p | p imply p |
      ( p ) | deadlock (only for A[], E<>)
  
```

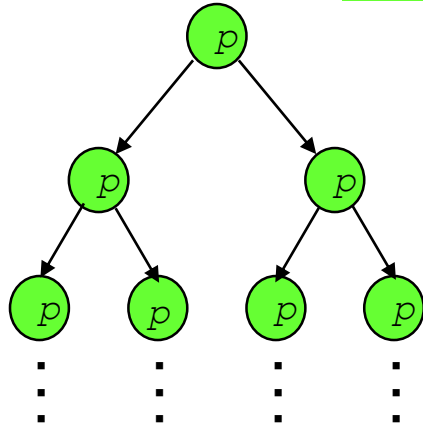
```
A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)
```

Uppaal “Computation Tree Logic”

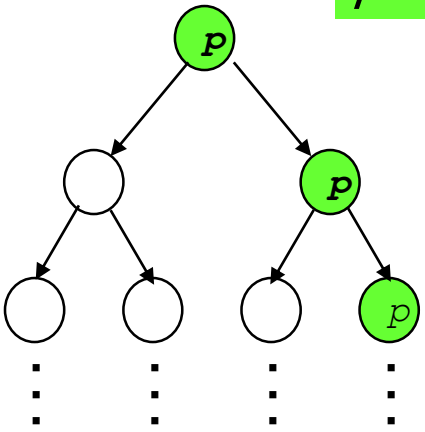
$E \langle \rangle p$ **Possible**



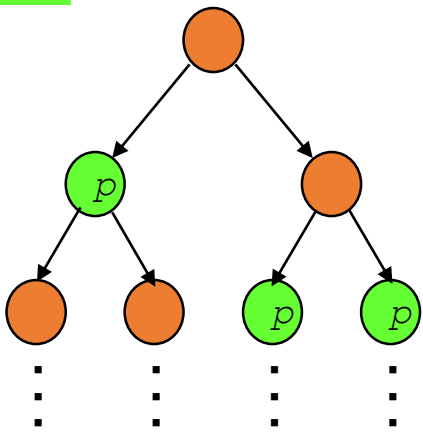
$A [] p$ **always**



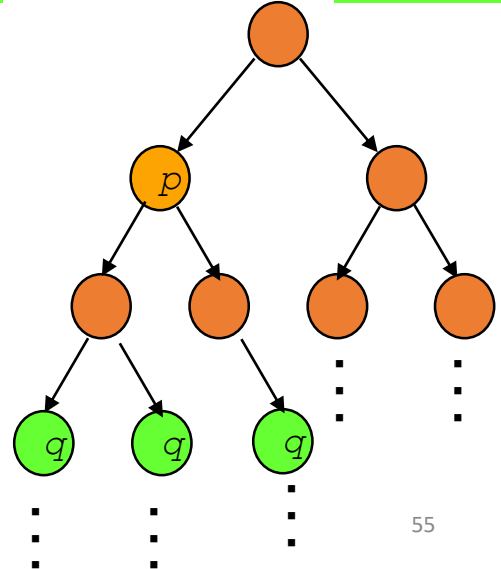
$E [] p$ **potentially always**



$A \langle \rangle p$ **inevitable**



$p \dashrightarrow q$ **leads-to**



Logical Specifications

- Validation Properties

- Possibly: $E \leftrightarrow p$

- Safety Properties

- Invariant: $A[] p$
- Possibly Inv.: $E[] P$

- Liveness Properties

- Eventually: $A \langle \rangle p$
- Leads_to: $p \dashrightarrow p$

- Bounded Liveness

- Leads to within: $p \dashrightarrow_{\leq t} q$

The expressions p and q

- must be type safe, side effect free, and evaluate to a boolean.

- only references to integer variables, constants, clocks, and locations are allowed (and arrays of these).

Logical Specifications

- Validation Properties

- Possibly: $E \langle \rangle \varphi$

- Safety Properties

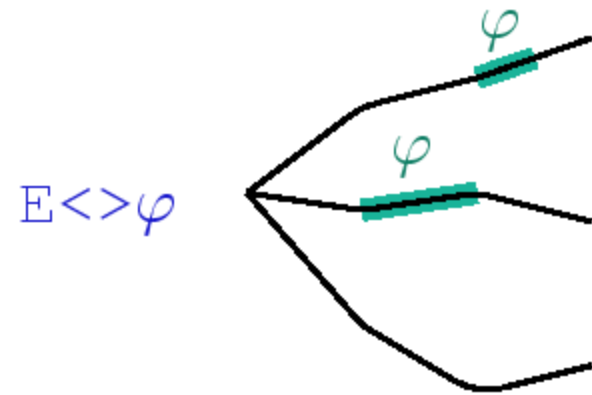
- Invariant: $A[] \varphi$
- Pos. Inv.: $E[] \varphi$

- Liveness Properties

- Eventually: $A \langle \rangle \varphi$
- Leadsto: $\varphi \dashrightarrow \psi$

- Bounded Liveness

- Leads to within: $\varphi \dashrightarrow_{\leq t} \psi$



Logical Specifications

- Validation Properties

- Possibly: $E \langle \rangle p$

- Safety Properties

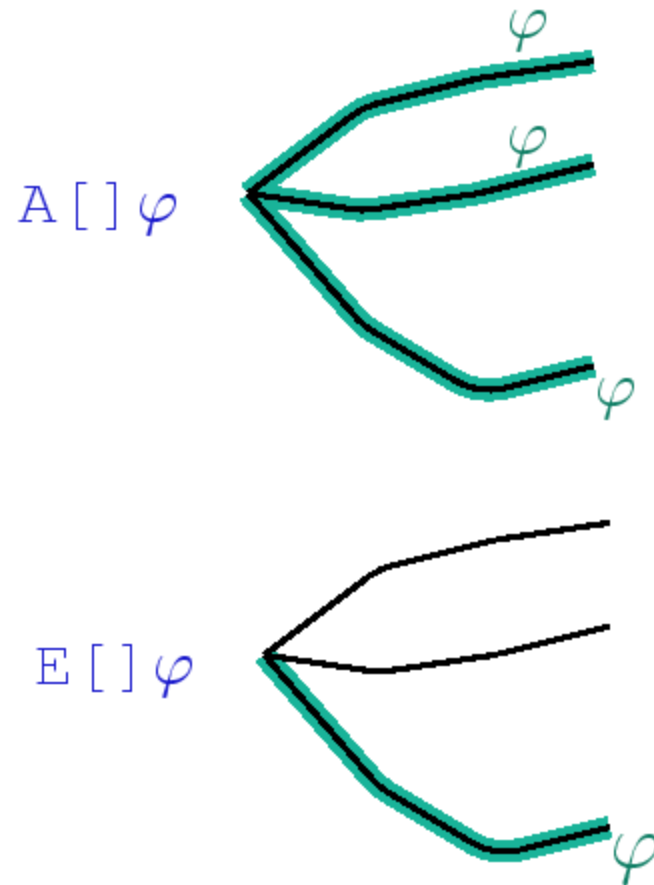
- Invariant: $A [] \varphi$
- Pos. Inv.: $E [] \varphi$

- Liveness Properties

- Eventually: $A \langle \rangle \varphi$
- Leadsto: $\varphi \dashrightarrow \psi$

- Bounded Liveness

- Leads to within: $\varphi \dashrightarrow_{\leq t} \psi$



Logical Specifications

- Validation Properties

- Possibly: $E \langle \rangle \varphi$

- Safety Properties

- Invariant: $A [] \varphi$

- Pos. Inv.: $E [] \varphi$

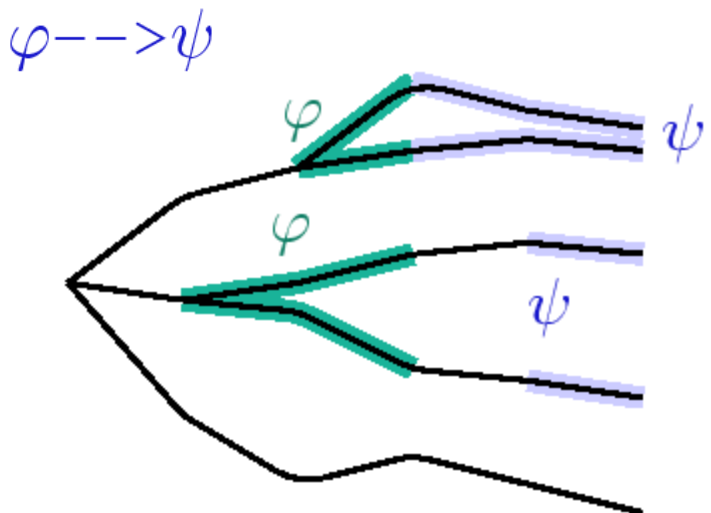
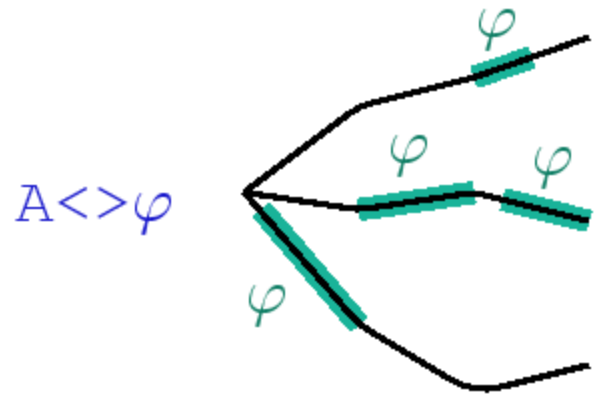
- Liveness Properties

- Eventually: $A \langle \rangle \varphi$

- Leadsto: $\varphi \dashrightarrow \psi$

- Bounded Liveness

- Leads to within: $\varphi \dashrightarrow_{\leq t} \psi$



Logical Specifications

- Validation Properties

- Possibly: $E \langle \rangle \varphi$

- Safety Properties

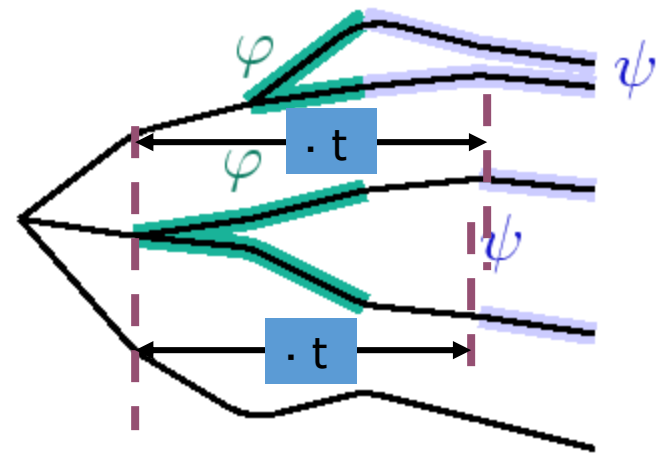
- Invariant: $A [] \varphi$
- Pos. Inv.: $E [] \varphi$

- Liveness Properties

- Eventually: $A \langle \rangle \varphi$
- Leadsto: $\varphi \dashrightarrow \psi$

- Bounded Liveness

- Leads to within: $\varphi \dashrightarrow_{\leq t} \psi$



Jug Example

- Safety: Never overflow.
 - $A[] \text{ forall}(i:id_t) \text{ level}[i] \leq \text{capa}[i]$
- Validation/Reachability: How to get 1 unit.
 - $E \langle \rangle \text{ exists}(i:id_t) \text{ level}[i] == 1$

Train-Gate Crossing

- Safety: One train on crossing at a time.
 - $A[]$ forall (i : id_t) forall (j : id_t)
Train(i).Cross && Train(j).Cross imply i == j
- Liveness: Approaching trains eventually cross.
 - Train(0).Appr --> Train(0).Cross
 - Train(1).Appr --> Train(1).Cross
 - ...
- No deadlock.
 - $A[]$ not deadlock