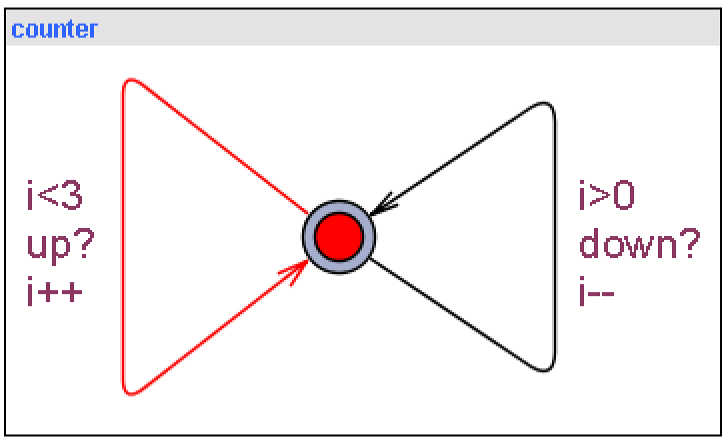
An introduction to Uppaal and Timed Automata

What is Uppaal?
(http://www.uppaal.com/)

- A simple graphical interface for drawing extended finite state machines (automatons + shared variables
- A graphical simulator – including MSC's
- An analyser (model checker)
- In addition, Uppaal supports the notion of *timed automatons*

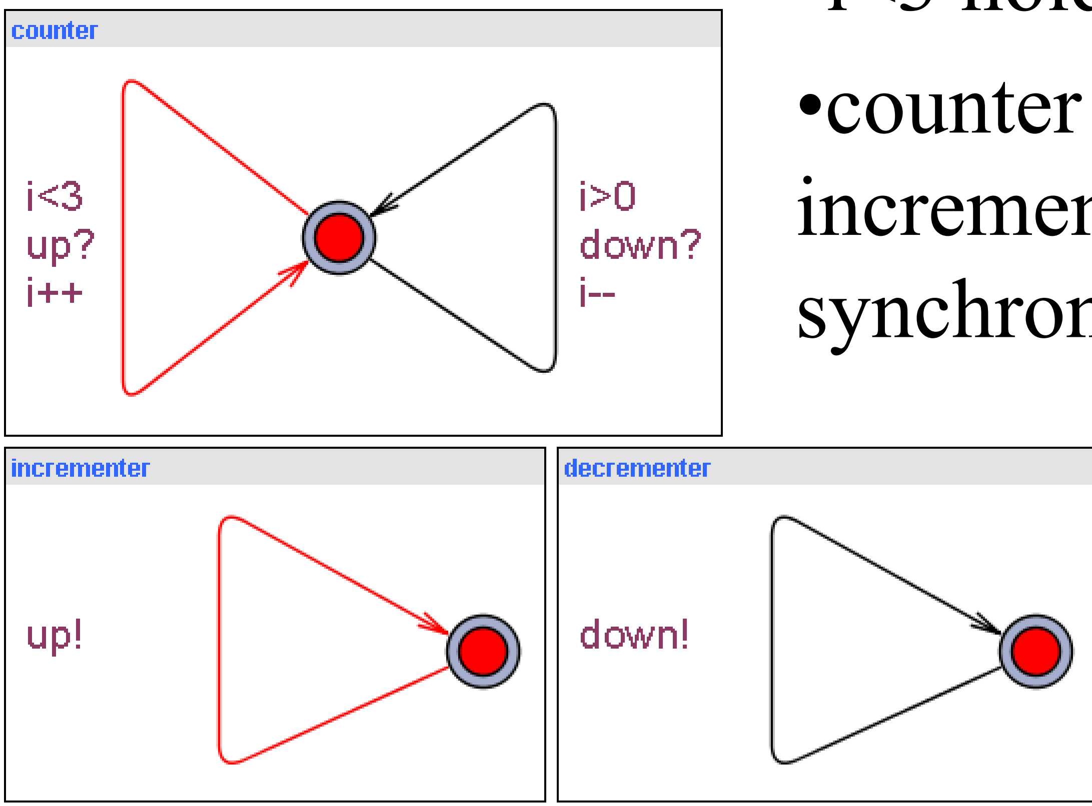
Transitions in Uppaal

- Automata transitions are labelled with the following (optional) parts:
  - A set of guards on variables
  - A label (input? or output!)
  - A set of variable assignments



- A transition can be taken when:
  - All guards are true
  - A synchronization is possible with another process

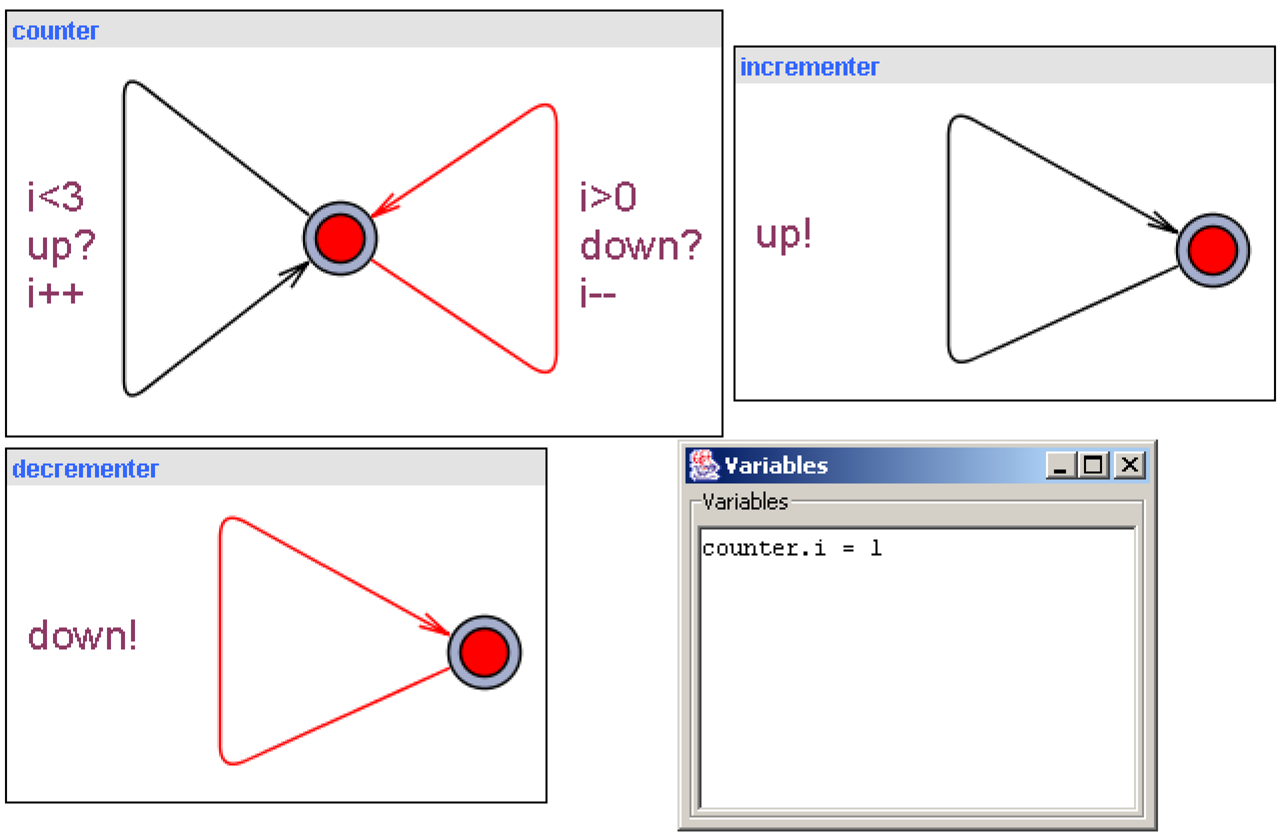
Transitions in Uppaal



- i<3 holds
- counter and incrementer may synchronize

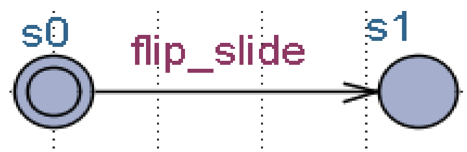
A Uppaal system

- Consists of at set (network) of automata
- System state = snapshot of each machines control location + local variables + global variables



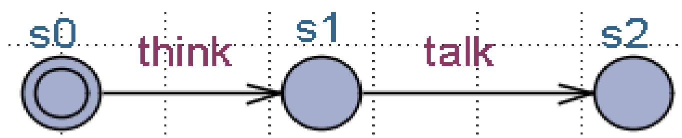
Parallel Composition: interleaving

Flipper



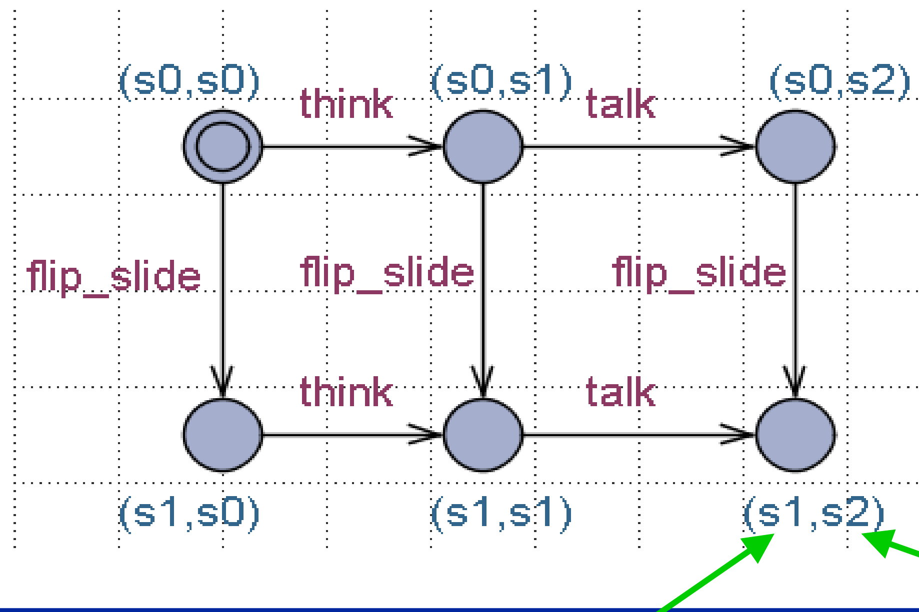
2 states

Speaker



3 states

Lecturer =
Speaker || Flipper



2*3 states

from Flipper

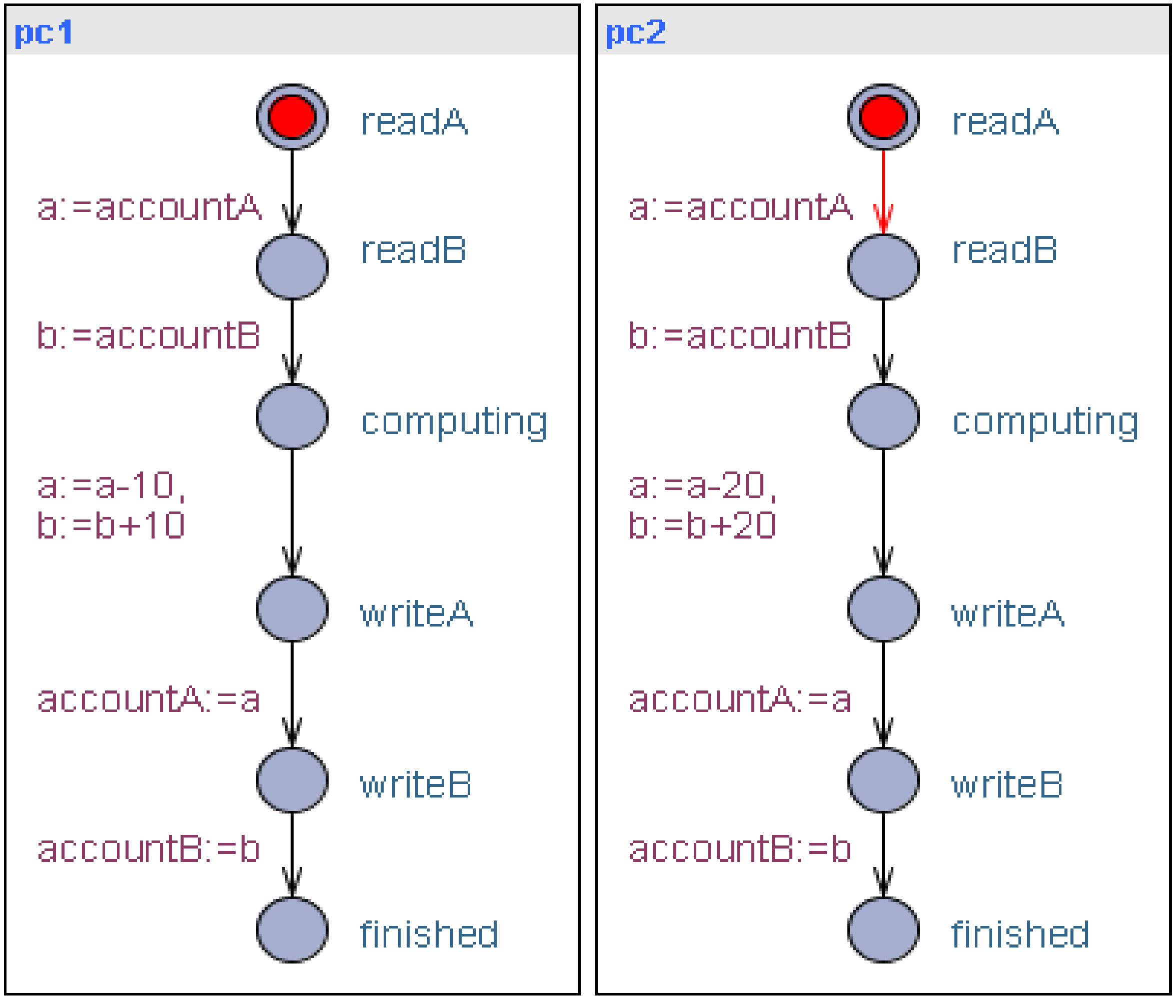
from Speaker

Home-Banking?

```
int accountA, accountB; //Shared global variables
//Two concurrent bank costumers

Thread costumer1 () {
  int a,b; //local tmp copy

  a=accountA;
  b=accountB;
  a=a-10;b=b+10;
  accountA=a;
  accountB=b;
}

Thread costumer2 () {
  int a,b;

  a=accountA;
  b=accountB;
  a=a-20; b=b+20;
  accountA=a;
  accountB=b;
}
```

- Are the accounts in balance after the transactions?

Home Banking



`A[] (pc1.finished and pc2.finished) imply (accountA+accountB==200)?`

Home Banking

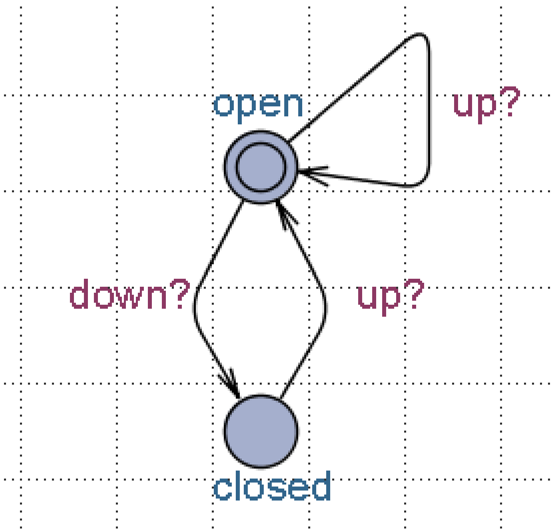
```
int accountA, accountB; //Shared global variables
Semaphore A,B;          //Protected by sem A,B
//Two concurrent bank costumers
```

```
Thread costumer1 () {
  int a,b; //local tmp copy

  down(A);
  down(B);
  a=accountA;
  b=accountB;
  a=a-10;b=b+10;
  accountA=a;
  accountB=b;
  up(A);
  up(B);
}
```

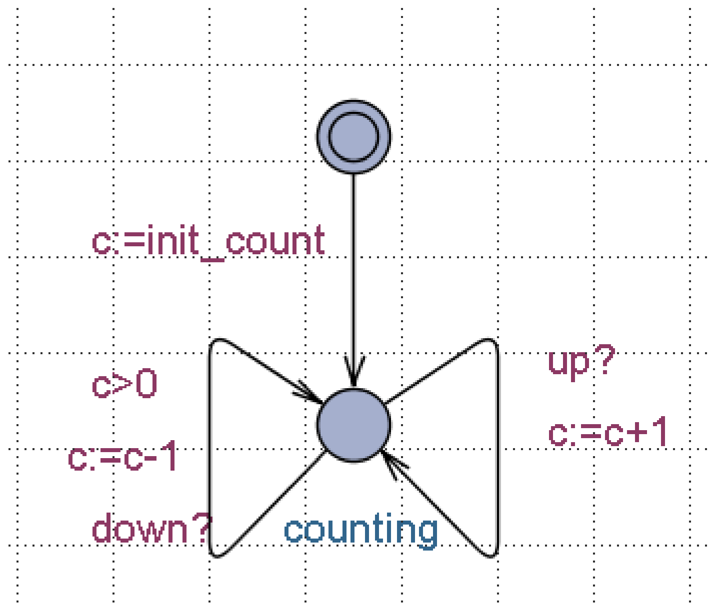
```
Thread costumer2 () {
  int a,b;

  down(B);
  down(A);
  a=accountA;
  b=accountB;
  a=a-20; b=b+20;
  accountA=a;
  accountB=b;
  up(B);
  up(A);
}
```

Semaphore FSM Model

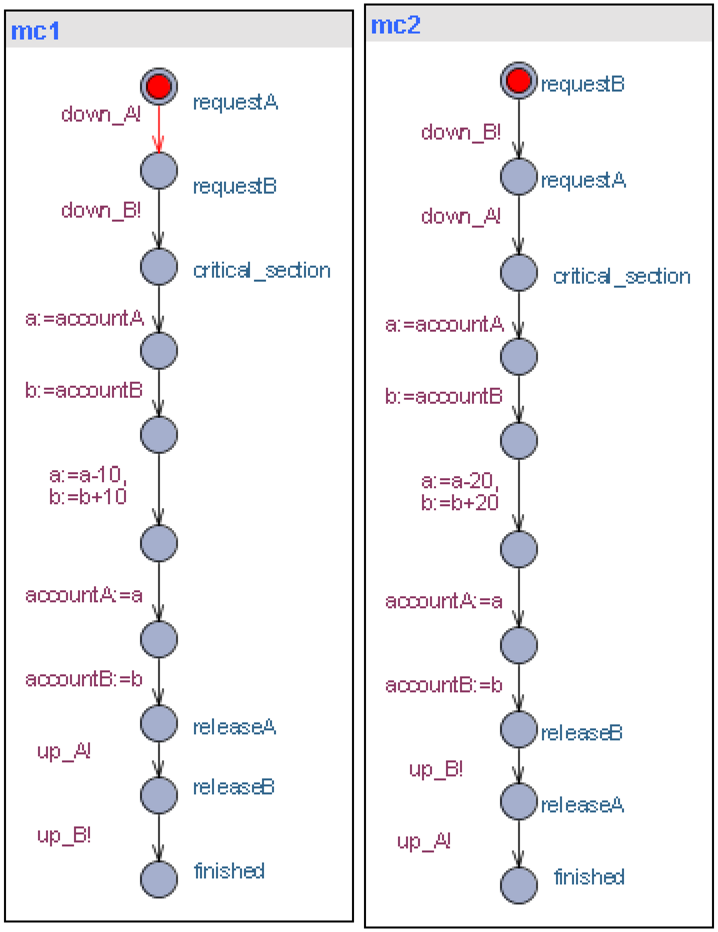
Binary Semaphore

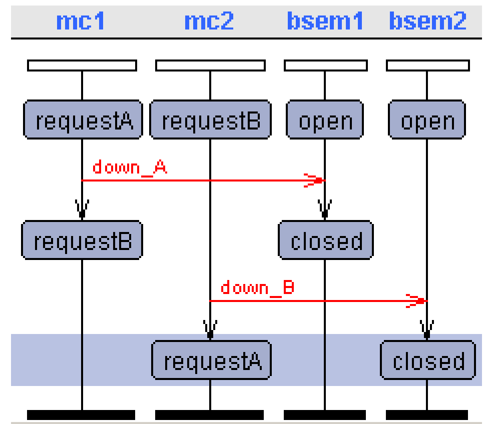


Counting Semaphore



Semaphore Solution?





1. Race conditions?
2. Consistency? (Balance)
3. Deadlock?

1. A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200) ✓
2. E<> mc1.critical_section and mc2.critical_section ✓
3. A[] not (mc1.finished and mc2.finished) imply not deadlock ÷

Reachability Analysis

- Compute *all* possible execution sequences
- And consequently *all* states of the system
- *Exhaustive search => proof*
- Check if each state encountered has the (un)-desired property

UPPAAL Property Specification Language

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

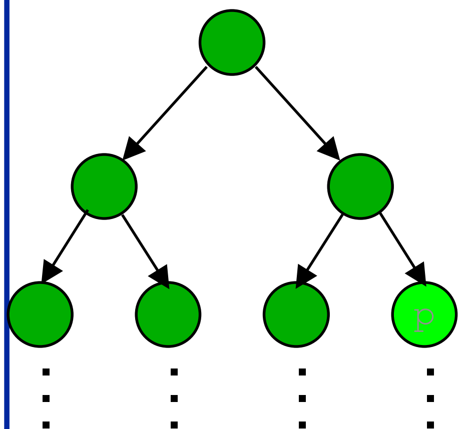
process location · data guards · clock guards

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

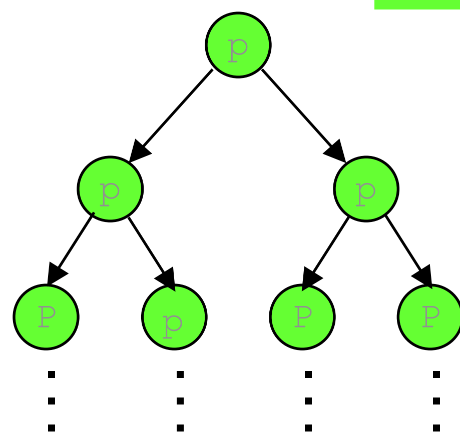
`A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)`

Uppaal "Computation Tree Logic"

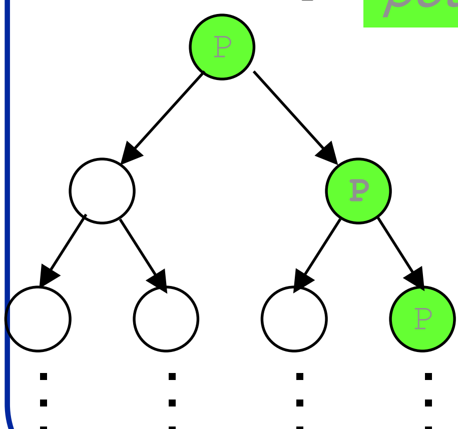
`E<> p` *Possible*



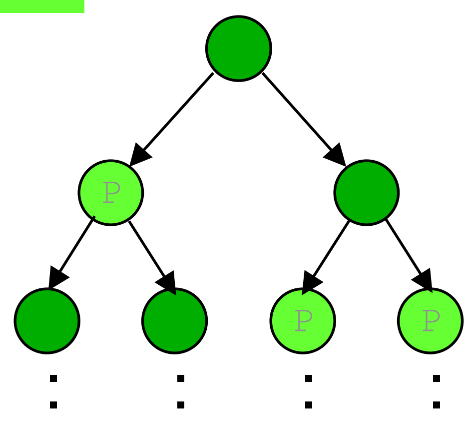
`A[] p` *always*



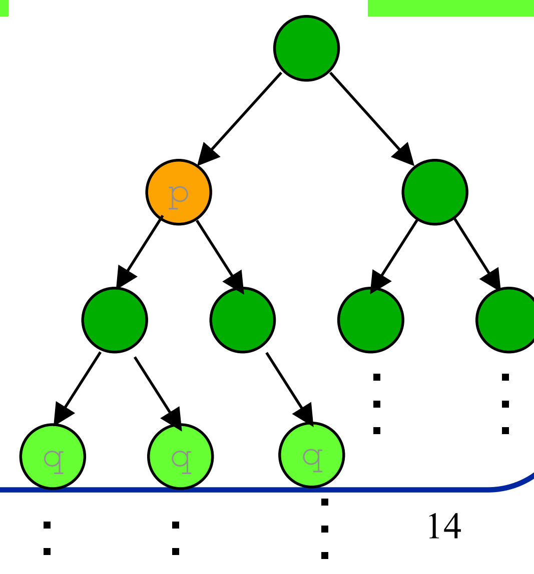
`E[] p` *potentially always*



`A<> p` *inevitable*

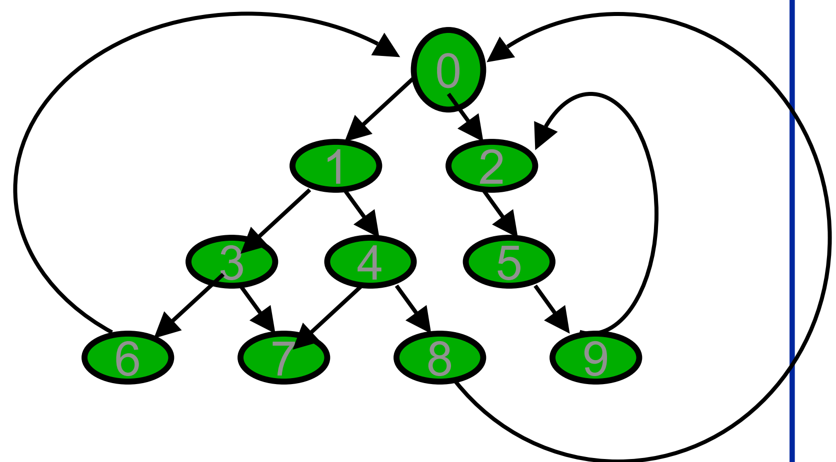


`p --> q` *leads-to*



Reachability Analysis

```
Passed:=Ø           //already seen states
Waiting:={S_0}      //states not examined yet
While(waiting!=Ø) {
  Waiting:=Waiting\{s_i}
  if s_i ∉ Passed
     whenever (s_j → s_j) then
        waiting:=waiting ∪ s_j
  }
```



Depth First: maintain waiting as a stack

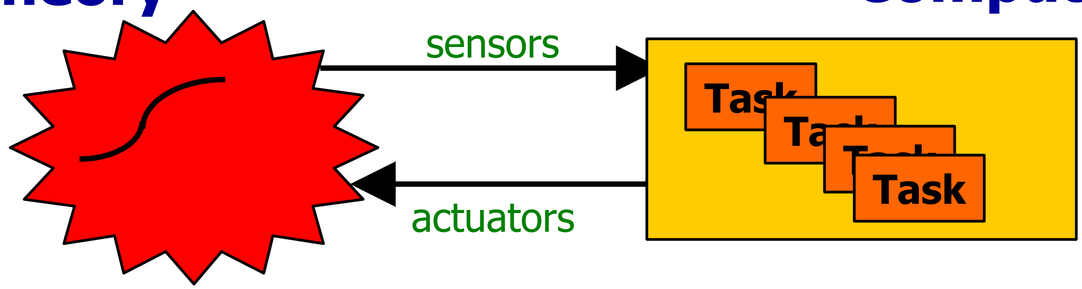
Order: 0 1 3 6 7 4 8 2 5 9

Breadth First: maintain waiting as a queue (shortest counter example)

Order: 0 1 2 3 4 5 6 7 8 9

Hybrid & Real Time Systems

Control Theory | **Computer Science**



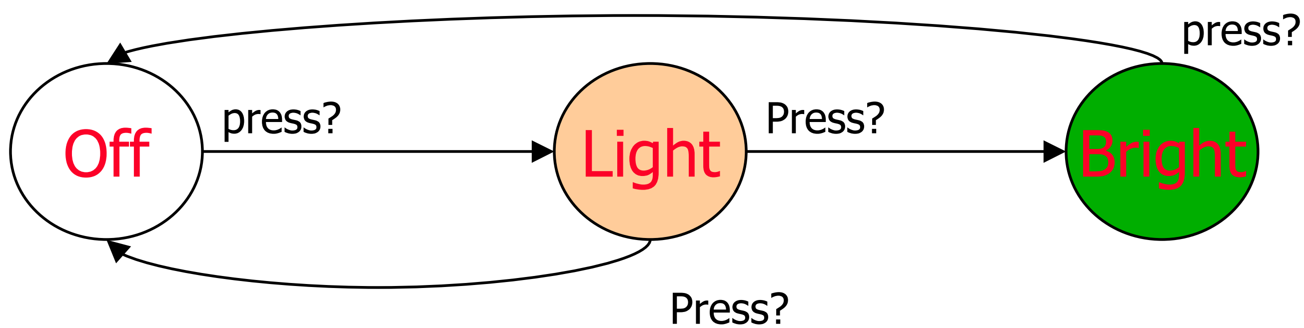
Plant
Continuous

Controller Program
Discrete

Eg.:
Pump Control
Air Bags
Robots
Cruise Control
ABS
CD Players
Production Lines

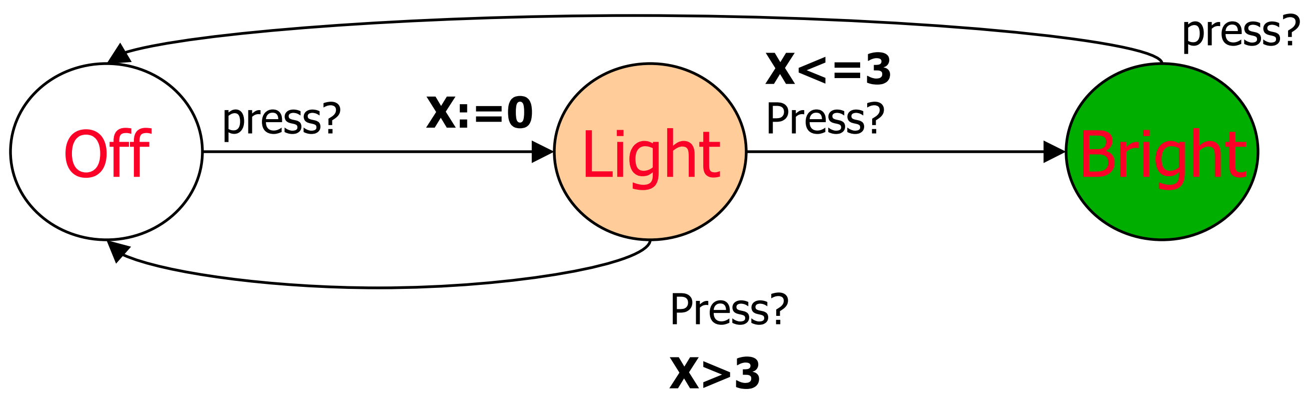
Real Time System
A system where correctness not only depends on the logical order of events but also on their timing

Timed Automata
Intelligent Light Control



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

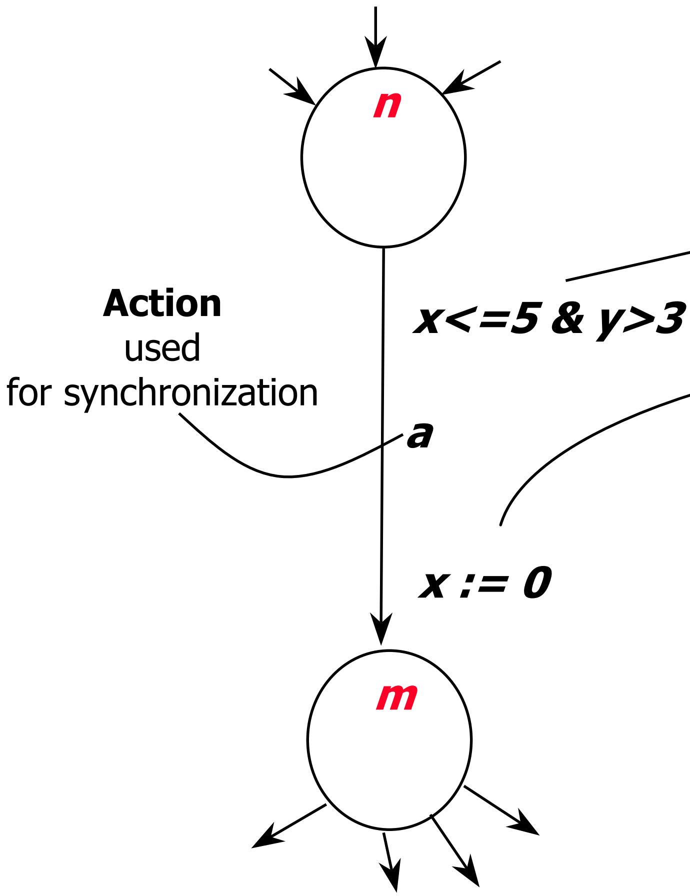
Timed Automata
Intelligent Light Control



Solution: Add real-valued clock **x**

Timed Automata

(Alur & Dill 1990)



Clocks: *x, y*

Guard
Boolean combination of comp with integer bounds

Reset
Action performed on clocks

State

(***location*** , ***x***=v , ***y***=u)   where v,u are in $\mathbf{R}$

Transitions

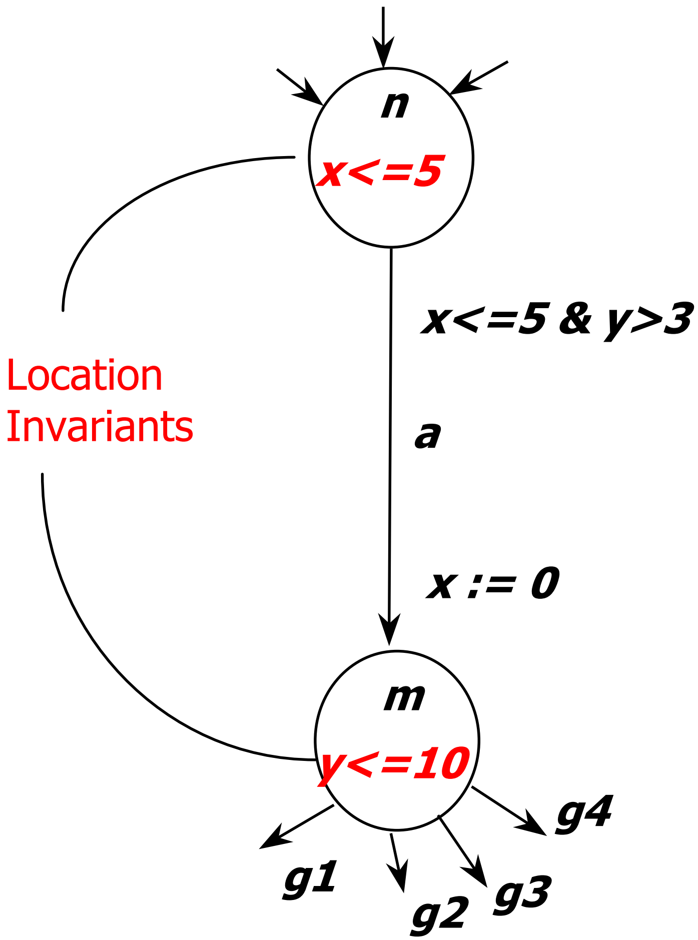
(***n*** , ***x***=2.4 , ***y***=3.1415) $\xrightarrow{a}$ (***m*** , ***x***=0 , ***y***=3.1415)

(***n*** , ***x***=2.4 , ***y***=3.1415) $\xrightarrow{e(1.1)}$ (***n*** , ***x***=3.5 , ***y***=4.2415)

(Henzinger et al, 1992)

Timed Safety Automata =

Timed Automata + Invariants



Clocks: ***x, y***

Transitions

(***n*** , ***x***=2.4 , ***y***=3.1415) ~~e(3.2)~~ →

(***n*** , ***x***=2.4 , ***y***=3.1415) —*e(1.1)*→ (***n*** , ***x***=3.5 , ***y***=4.2415)

Invariants ensure progress!!

Clock Constraints

For set C of clocks with $x, y \in C$, the set of *clock constraints* over C, $\Psi(C)$, is defined by

$$\alpha ::= x \prec c \;\Big|\; x - y \prec c \;\Big|\; \neg\alpha \;\Big|\; (\alpha \;\wedge\; \alpha)$$

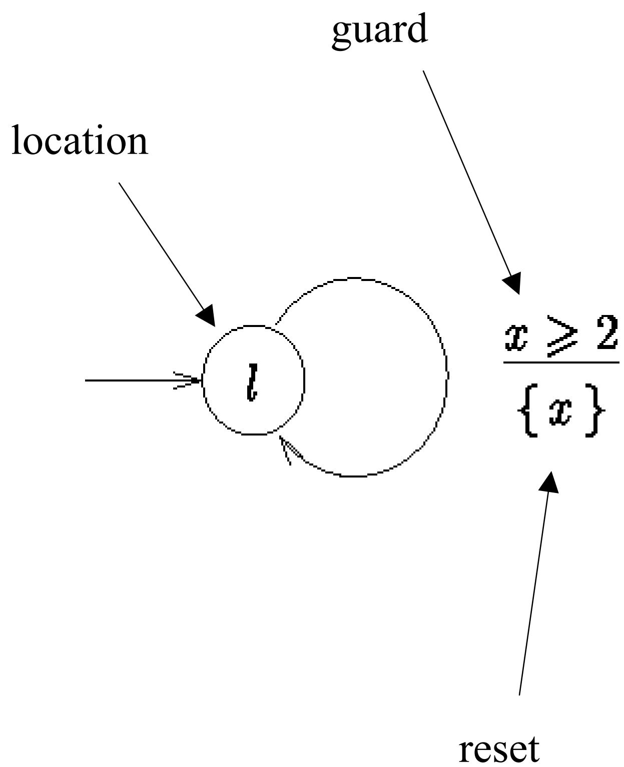
where $c \in \mathbb{N}$ and $\prec\; \in \{<, \leqslant\}$.

Timed (Safety) Automata

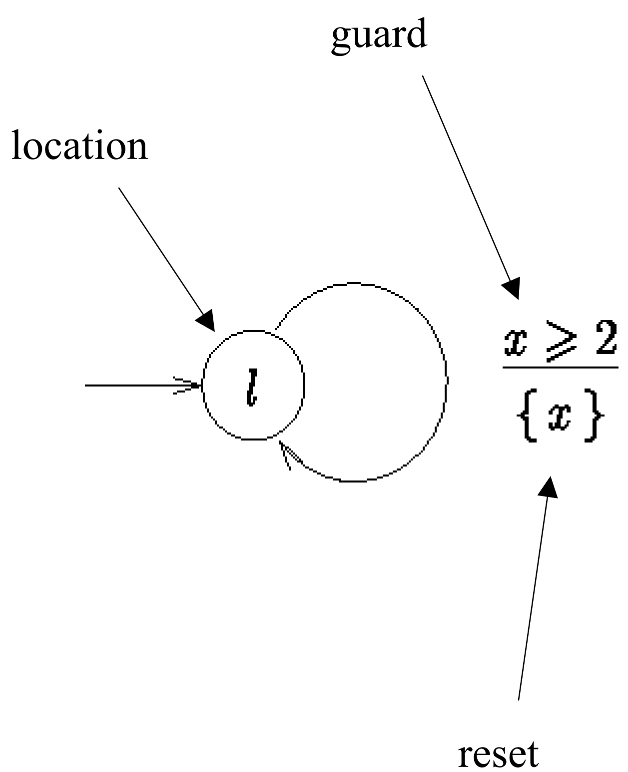
A *timed automaton* $\mathcal{A}$ is a tuple $(L, l_0, E, Label, C, clocks, guard, inv)$ with

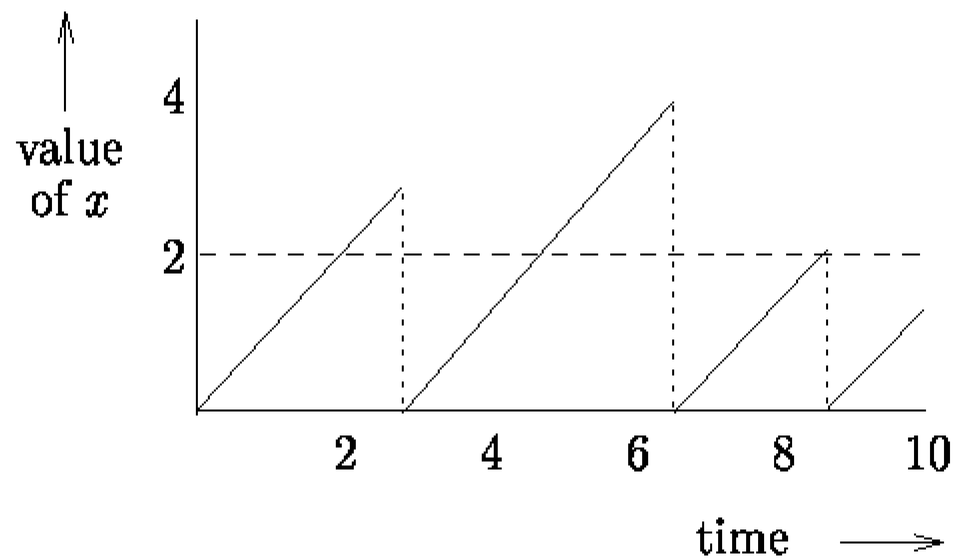
- L, a non-empty, finite set of locations with initial location $l_0 \in L$
- $E \subseteq L \times L$, a set of edges
- $Label : L \longrightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set $Label(l)$ of atomic propositions
- C, a finite set of clocks
- $clocks : E \longrightarrow 2^C$, a function that assigns to each edge $e \in E$ a set of clocks $clocks(e)$
- $guard : E \longrightarrow \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint $guard(e)$ over C, and
- $inv : L \longrightarrow \Psi(C)$, a function that assigns to each location an *invariant*.

Timed Automata: Example

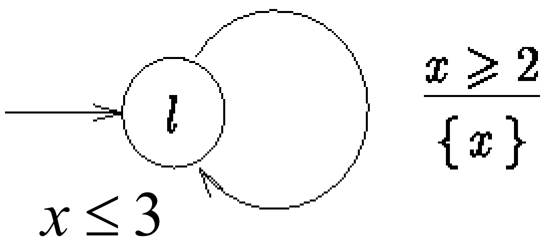


Timed Automata: Example

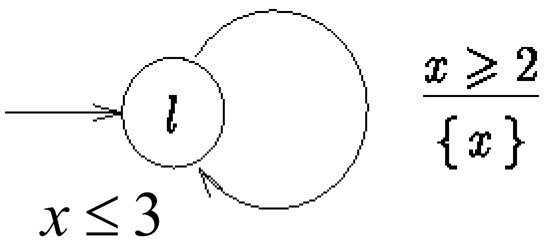


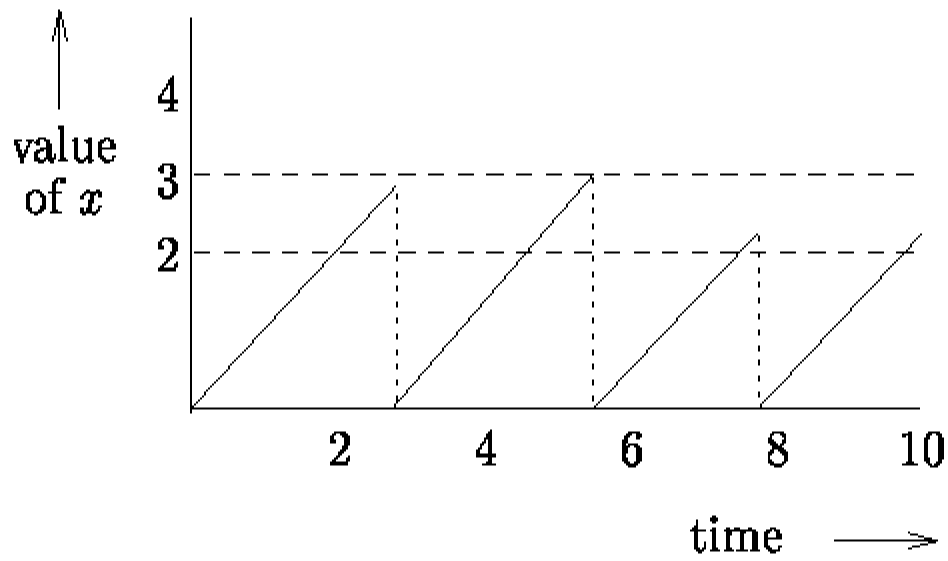


Timed Automata: Example

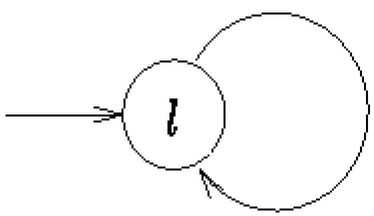


Timed Automata: Example



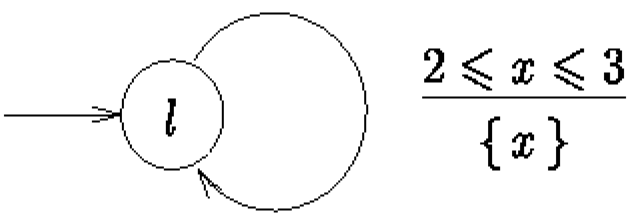


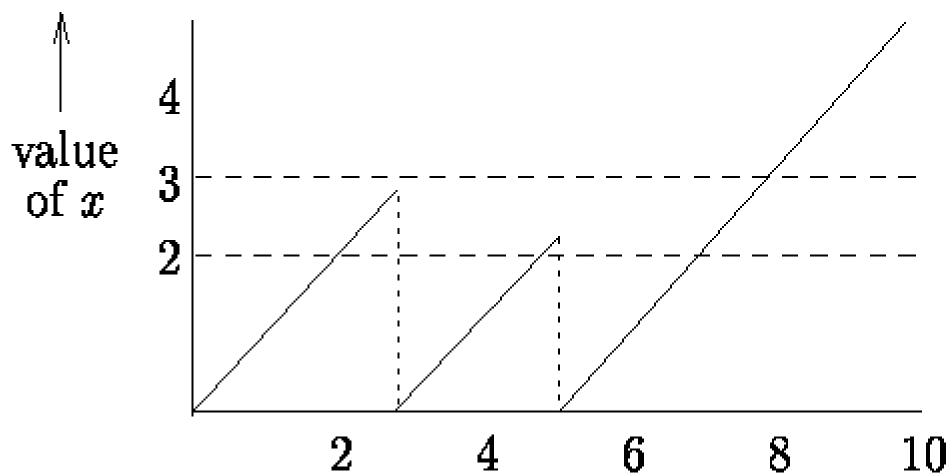
Timed Automata: Example



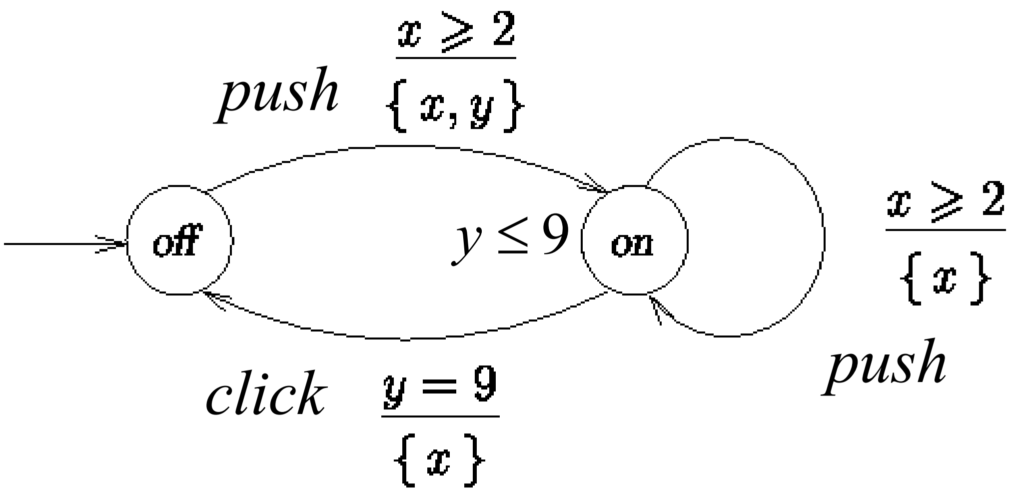
$$\frac{2 \leqslant x \leqslant 3}{\{x\}}$$

Timed Automata: Example

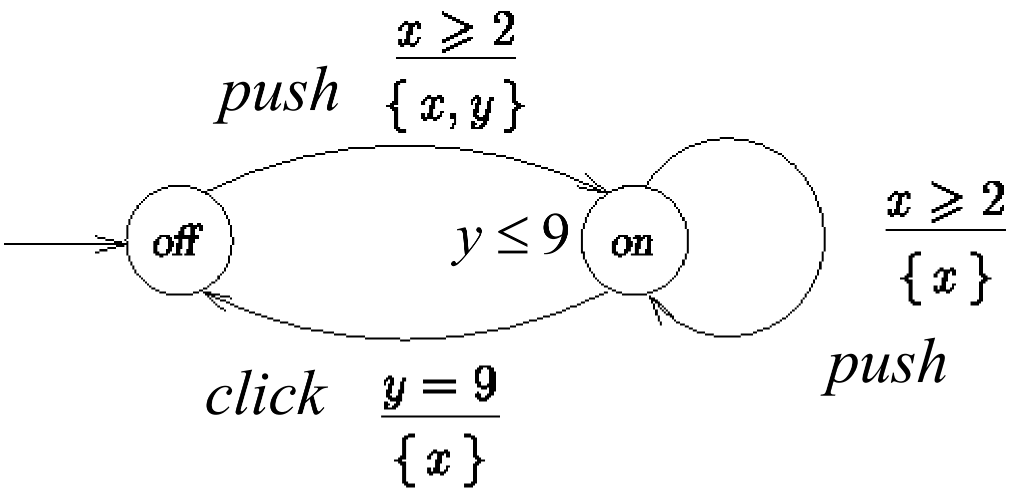




Light Switch

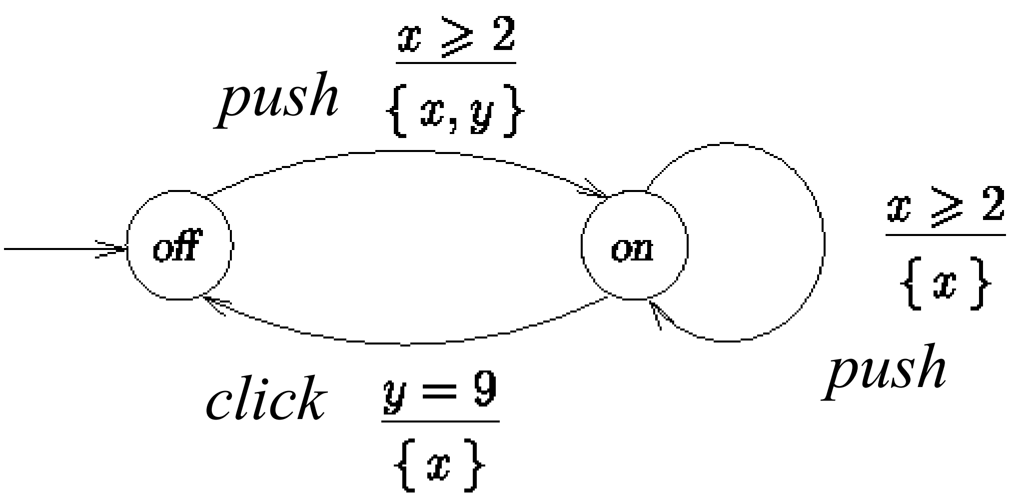


Light Switch



- Switch may be turned on whenever at least 2 time units has elapsed since last “turn off”

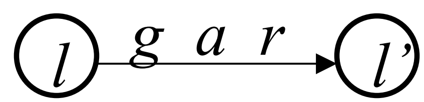
Light Switch



- Switch may be turned on whenever at least 2 time units has elapsed since last "turn off"

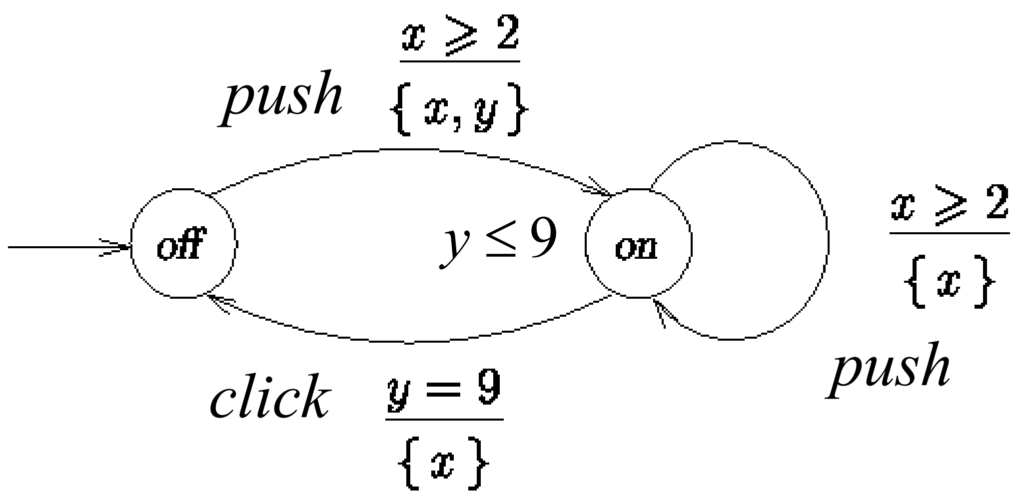
- Light automatically switches off after 9 time units.

Semantics

- *clock valuations*: $V(C) \quad v : C \rightarrow R_{\geq 0}$
- *state*: (l, v) where $l \in L$ and $v \in V(C)$
- Semantics of timed automata is a *labeled transition system* $(S, \rightarrow)$ where $S = \{ (l, v) \mid v \in V(C) \text{ and } l \in L \}$
- *action transition* $(l, v) \xrightarrow{a} (l', v')$ iff 

$g(v)$ and $v' = v[r]$ and $Inv(l')(v')$
- *delay Transition* $(l, v) \xrightarrow{d} (l, v + d)$ iff
$Inv(l)(v + d')$ whenever $d' \leq d \in R_{\geq 0}$

Semantics: Example



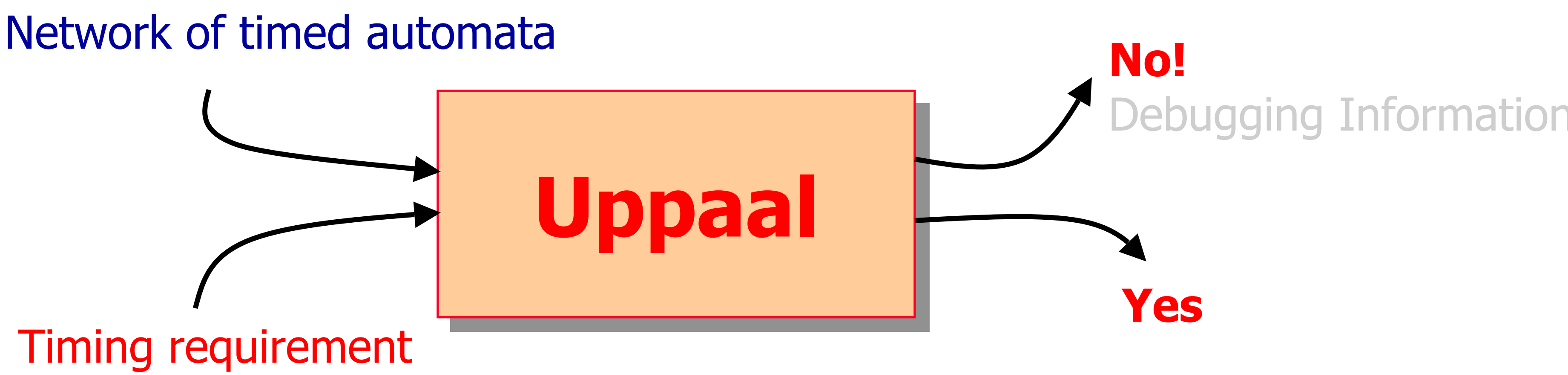
$$(off, x = y = 0) \xrightarrow{3.5} (off, x = y = 3.5) \xrightarrow{push}$$

$$(on, x = y = 0) \xrightarrow{\pi} (on, x = y = \pi) \xrightarrow{push}$$

$$(on, x = 0, y = \pi) \xrightarrow{3} (on, x = 3, y = \pi + 3) \xrightarrow{9-(\pi+3)}$$

$$(on, x = 9 - (\pi + 3), y = 9) \xrightarrow{click} (off, x = 0, y = 9) \ldots$$

Uppaal



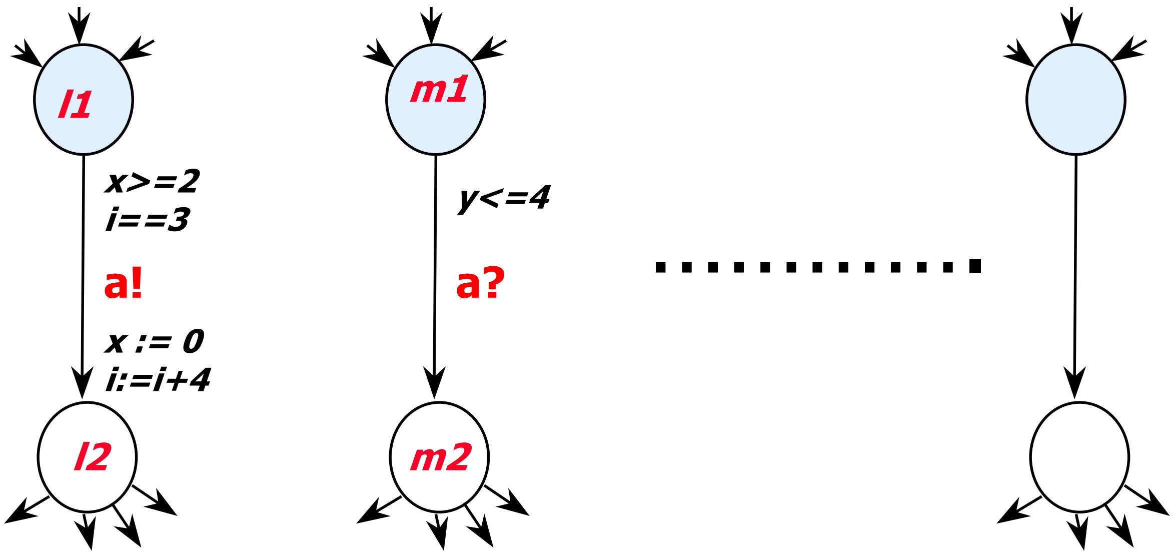
Uppsala (6 persons), Aalborg (10 persons), 1995-
21 papers, 6 invited talks/tutorials
9 industrial case studies
http://www.docs.uu.se/docs/rtmv/uppaal/index.shtml
(or www.uppaal.com)

Timed Automata in UPPAAL

- Networks of Timed Safety Automata
  + urgent actions
  + urgent locations
  (i.e. zero-delay locations)
  + committed locations
  (i.e. zero-delay and **atomic** locations)
  + data-variables (integers with bounded domains)
  + arrays of data-variables
  + guards and assignments over data-variables and arrays...

Networks of Timed Automata

+ Integer Variables + arrays ….



Two-way synchronization on *complementary* actions.

Closed Systems!

Example transitions

(***l1***, ***m1***,………, *x=2, y=3.5, i=3,…..)* —tau→ (***l2,m2***,……..,*x=0, y=3.5, i=7,…..)*

~~—0.2→ (***l1,m1***,………,*x=2.2, y=3.7, I=3,…..)*~~

If **a** URGENT CHANNEL

Timed Automata in UPPAAL

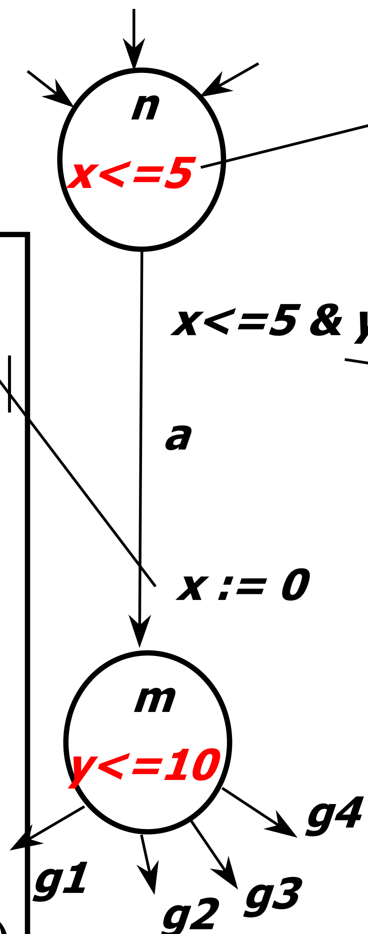
clock assignments

$$x := n$$

clock assignments

$$i := Expr$$

$$Expr ::= i \mid i[Expr] \mid n \mid -Expr \mid Expr + Expr \mid Expr - Expr \mid Expr * Expr \mid Expr \,/\, Expr \mid (g_d \,?\, Expr : Expr)$$



location invariants

$$inv ::= x < n \mid x \leq n \mid inv, inv$$

clock — natural number — and

$$g ::= g_c \mid g_d \mid g, g$$

$$g_c ::= x \prec n \mid x \prec y + n$$ **clock guards**

$$g_d ::= Expr \;\; op \;\; Expr$$ **data guards**

$$\prec \, \in \{<, <=, =, >=, >\}$$

$$op \in \{<, <=, =, >=, >, != \}$$

Urgent Channels

```
urgent chan hurry;
```

Informal Semantics:

• There will be no delay if transition with urgent action can be taken.

Restrictions:

• No clock guard allowed on transitions with urgent actions.

• Invariants and data-variable guards are allowed.

Urgent Locations

Click "Urgent" in State Editor.

Informal Semantics:

- No delay in urgent location.

Note: the use of urgent locations **reduces** the number of states in a model, and thus the complexity of the analysis.

Committed Locations

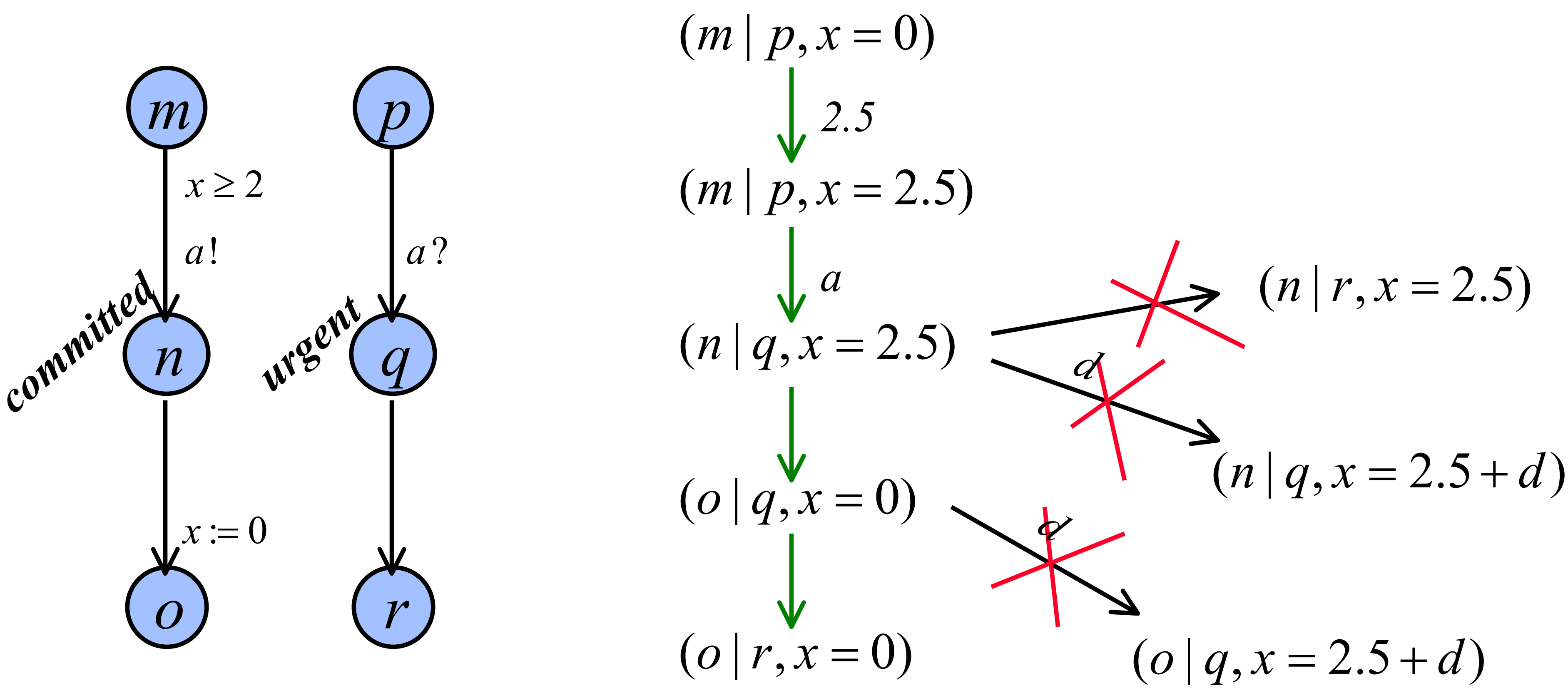
Click "Committed" in State Editor.

Informal Semantics:

- No delay in committed location.
- Next transition must involve automata in committed location.

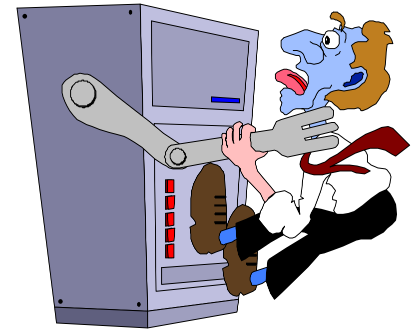
Note: the use of committed locations **reduces** the number of states in a model, and allows for more space and time efficient analysis.

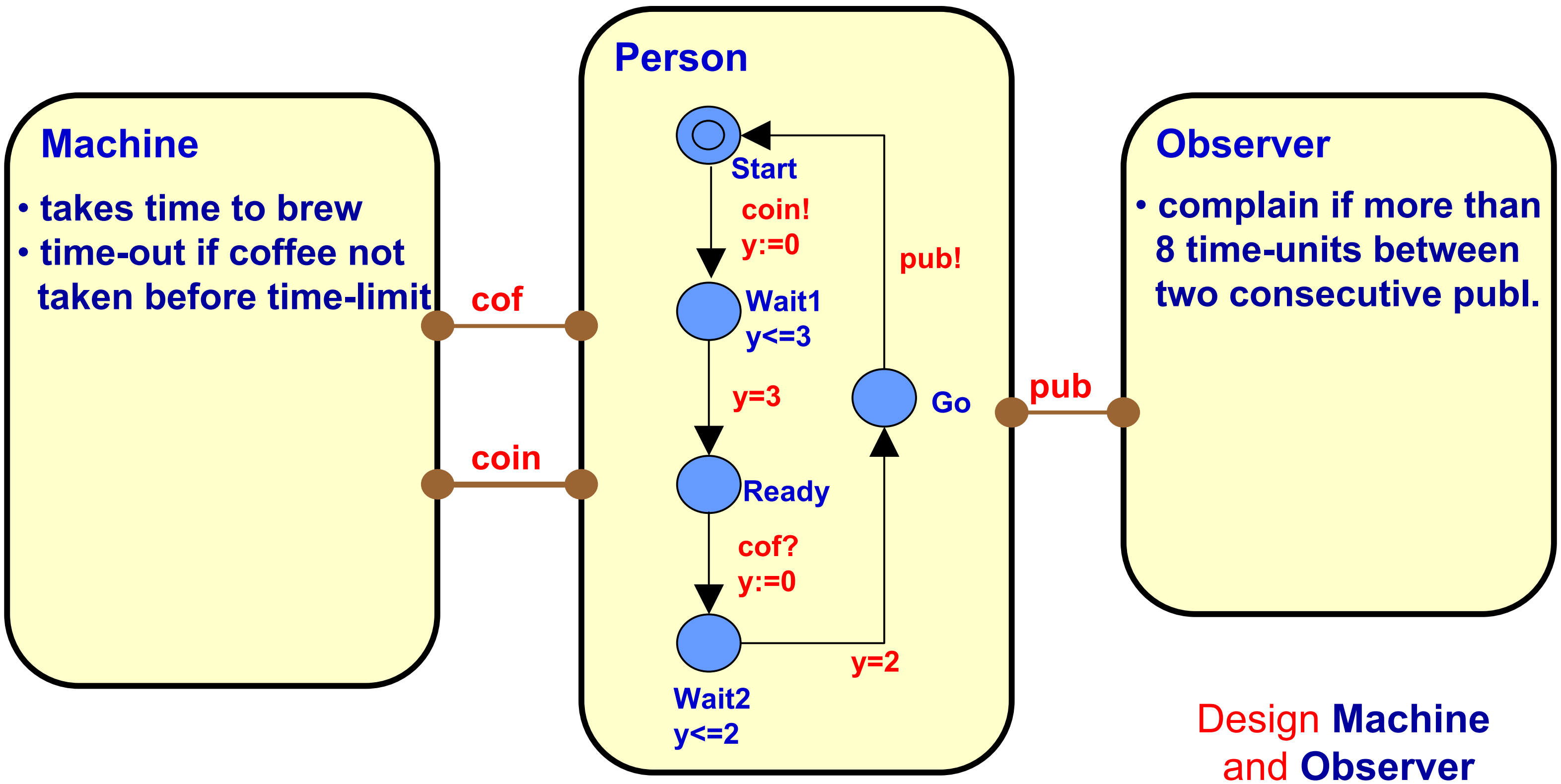
Urgent and Committed Locations



Uppaal Demo

Exercise: The Coffee Machine





Design **Machine** and **Observer**