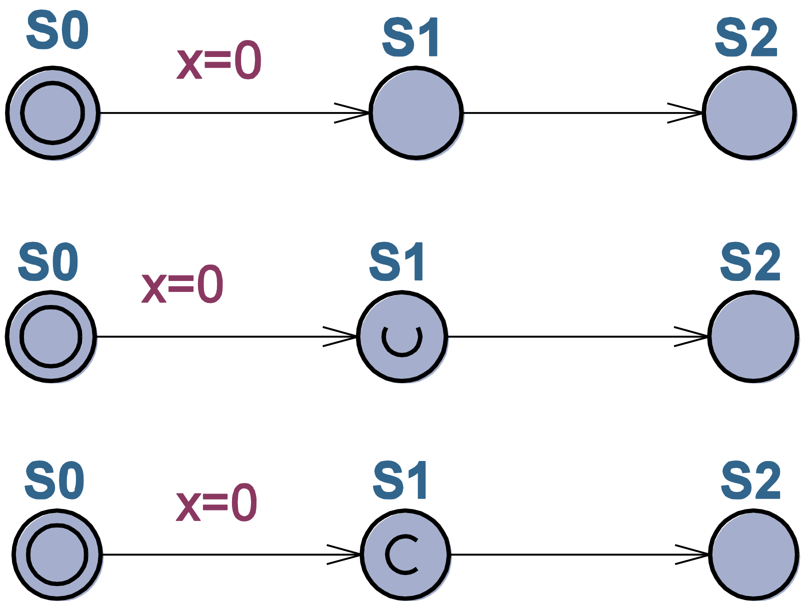
Faculty of Computer Systems & Software Engineering

Formal methods.

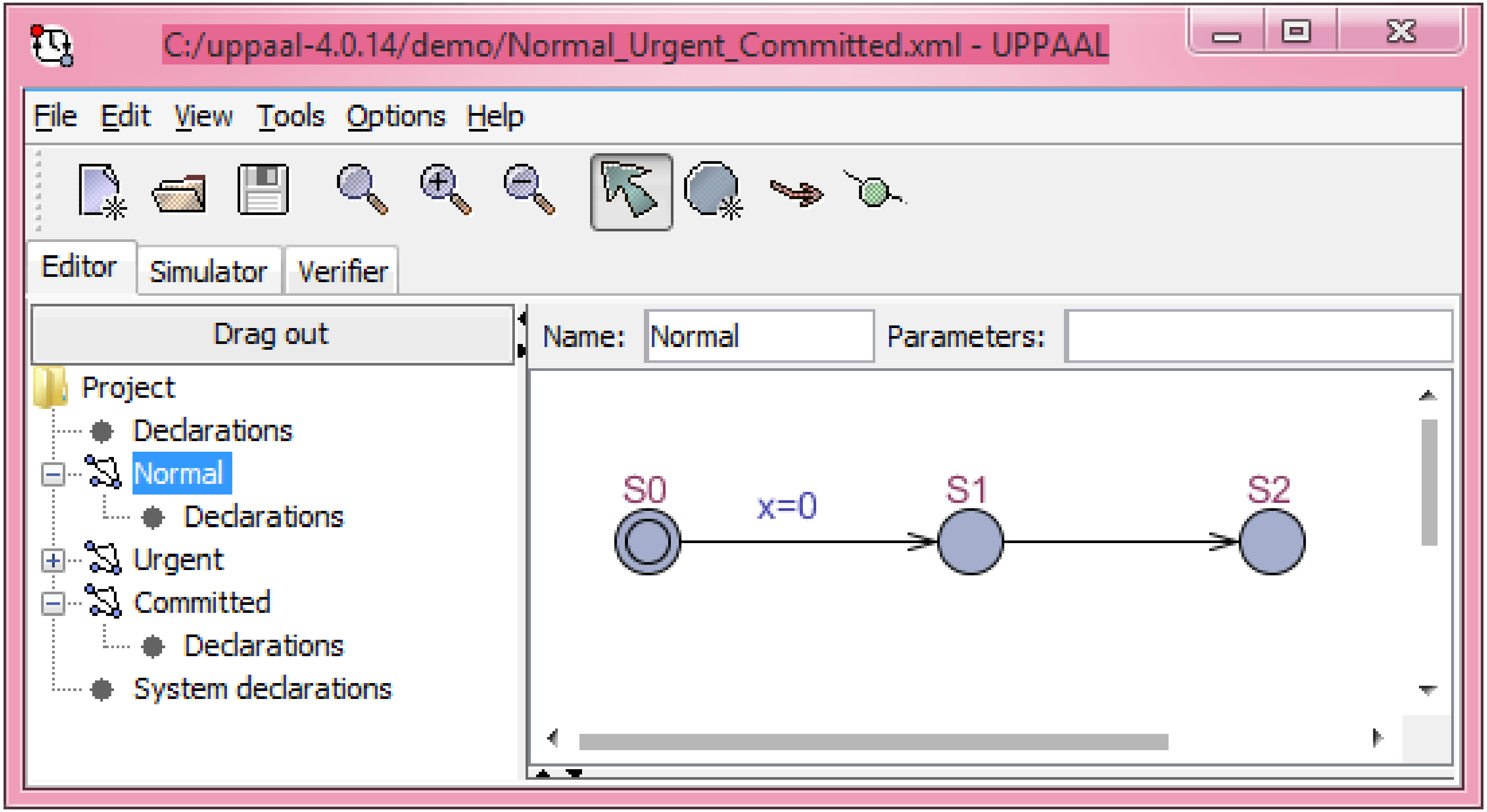
Modelling with UPPAAL

Vitaliy Mezhuyev

Normal, Urgent and Committed Locations

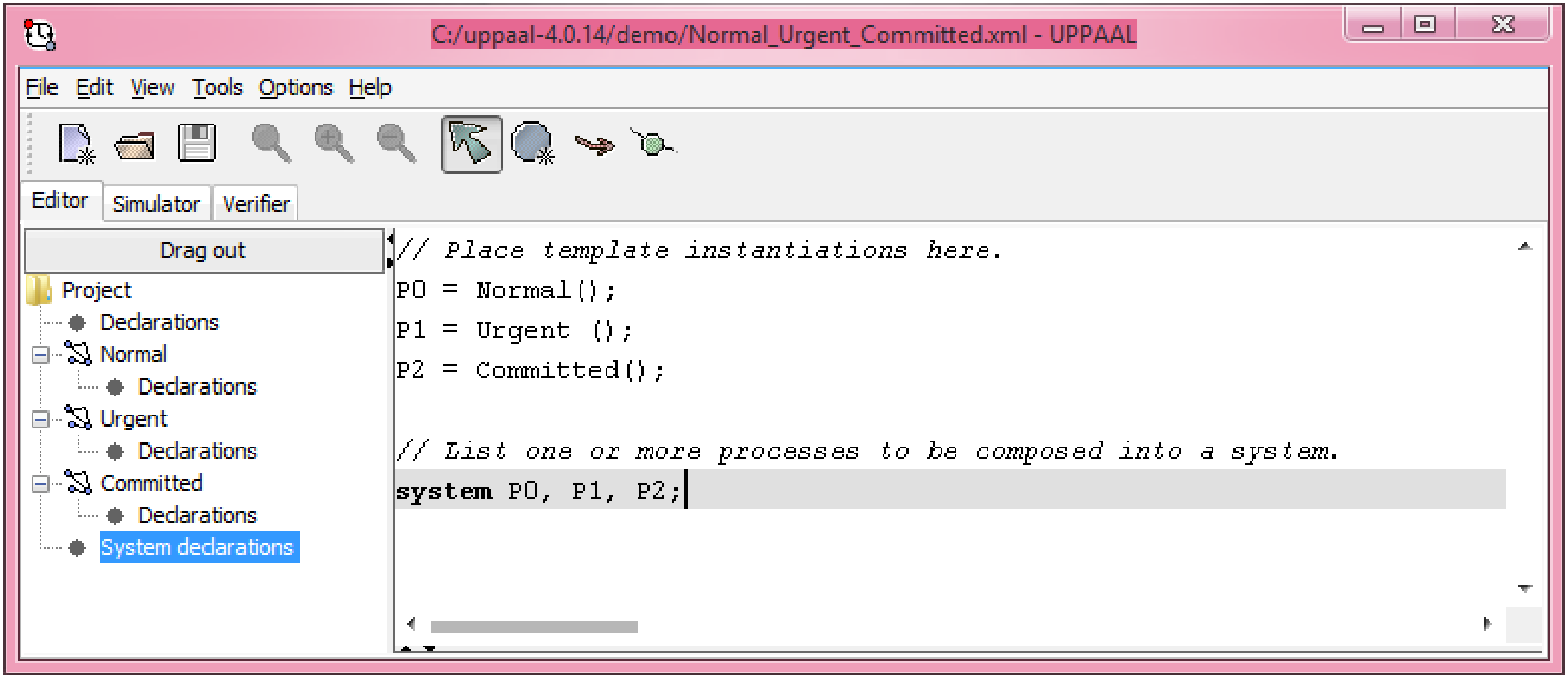


The model of automata with normal locations

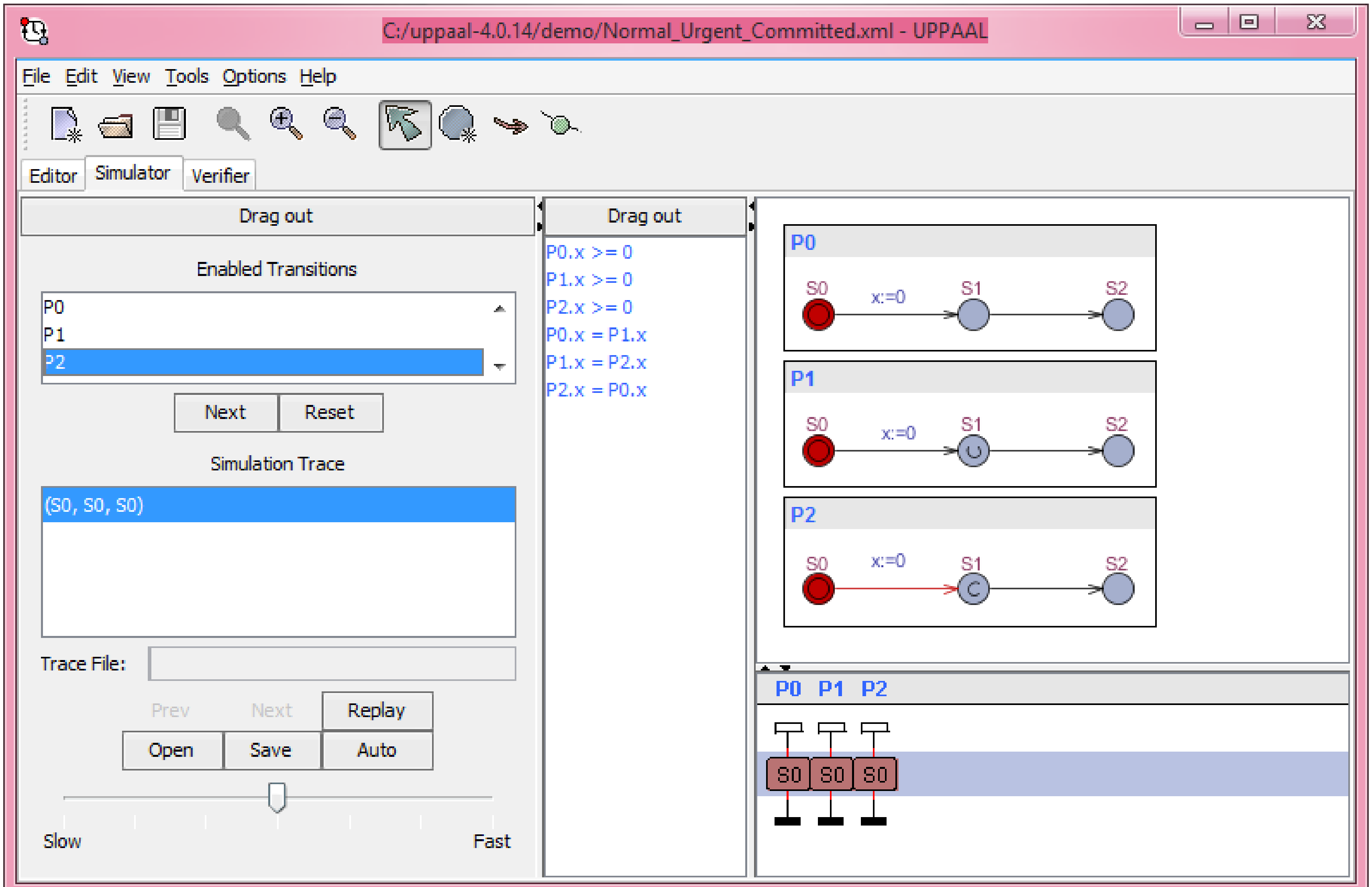


Use Edit -> Insert Template to create two others automata
Define **clock x;** for each automata (local vs global declarations)

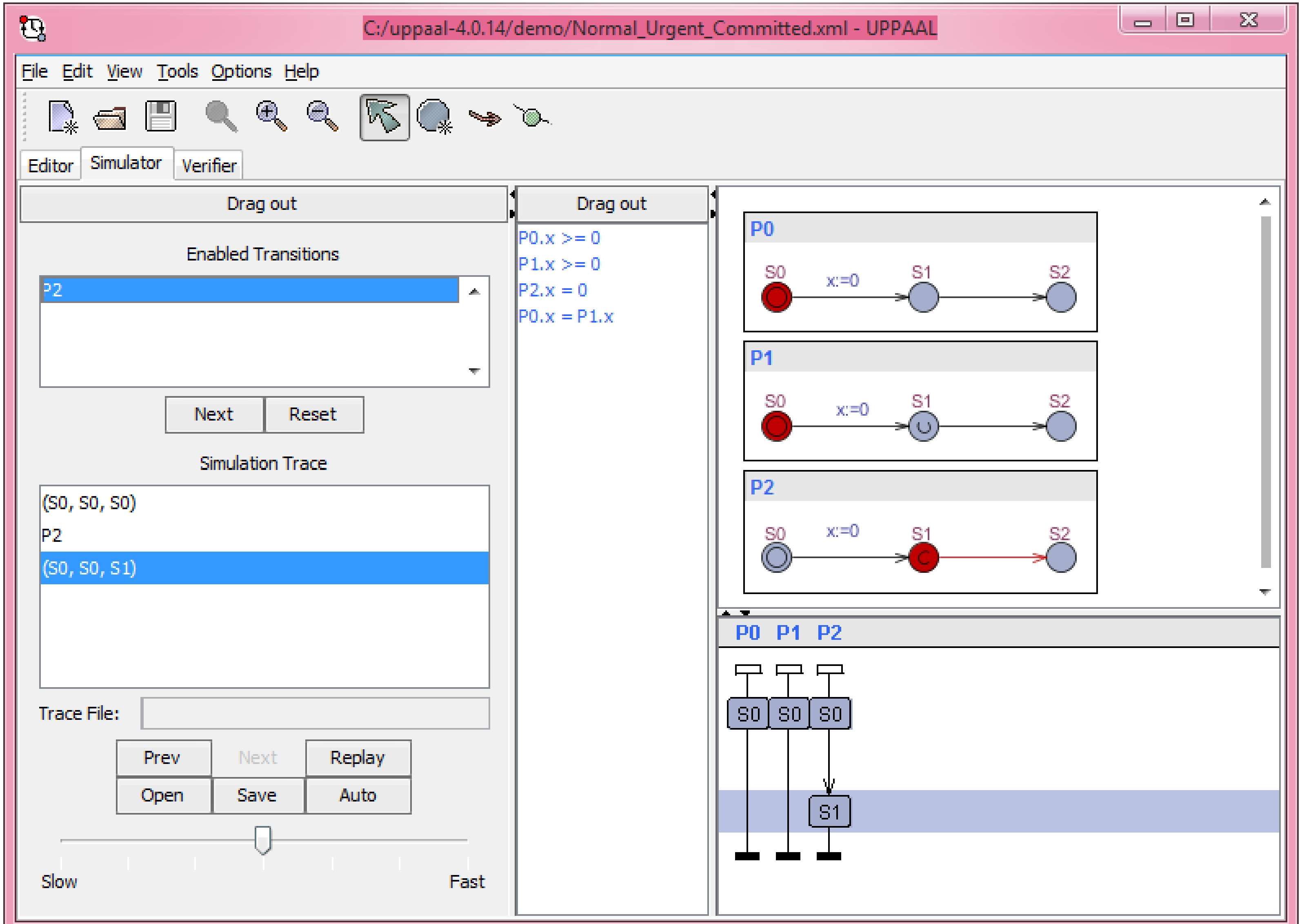
The model of system



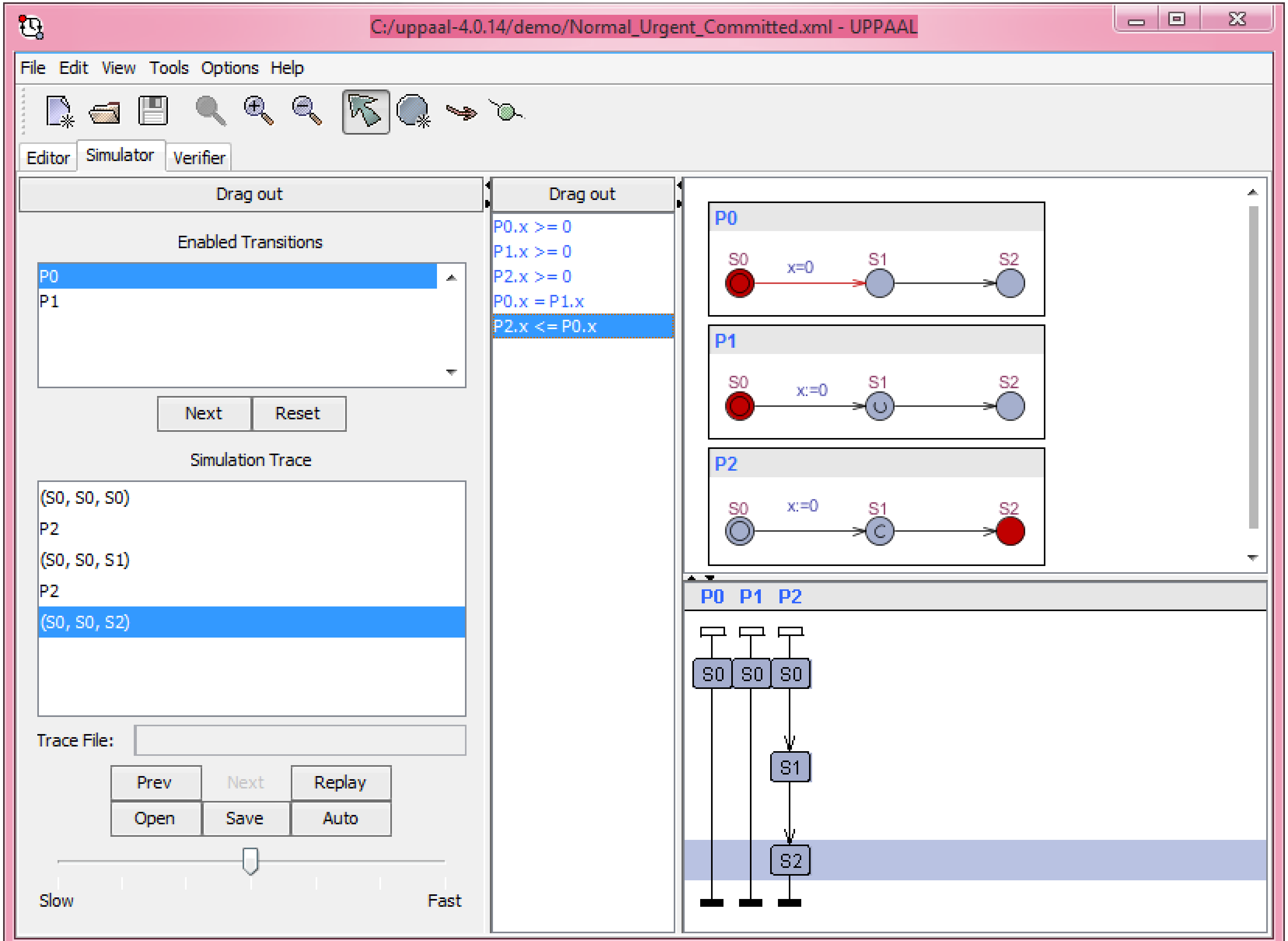
Possible state transitions



State transition from committed node



Possible state transitions after committed node



Difference between normal and urgent state

Is it possible to wait in S1 of P0 (normal location)
E<> P0.S1 and P0.x>0 // TRUE

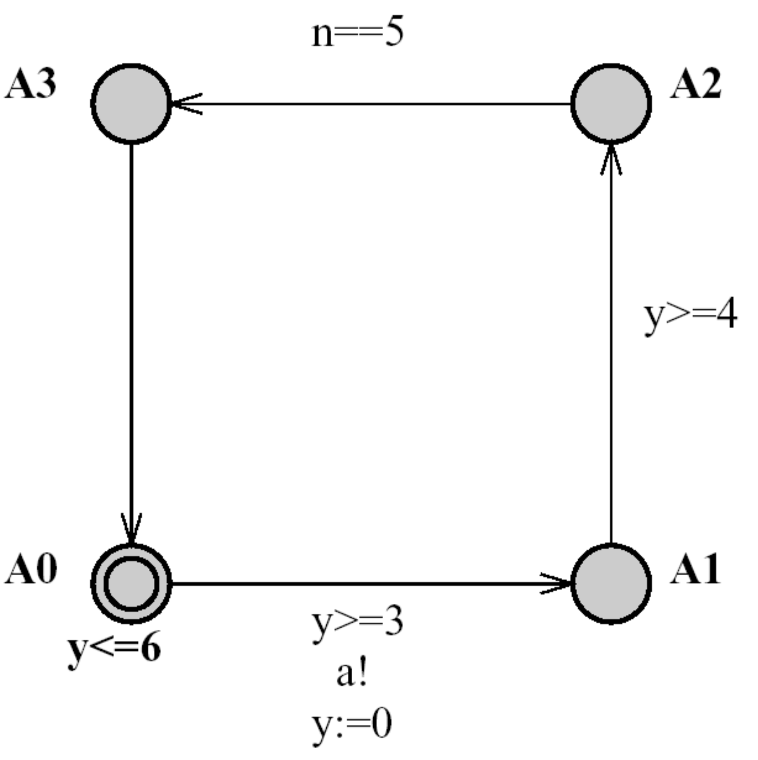
It is not possible to wait in S1 of P1 (urgent location)
A[] P1.S1 imply P1.x==0 // TRUE

A[] P1.S1 imply P1.x>0 // FALSE

Time may not pass in an urgent state, but interleavings with normal states are possible. Thus, urgent locations are “less strict” than committed.

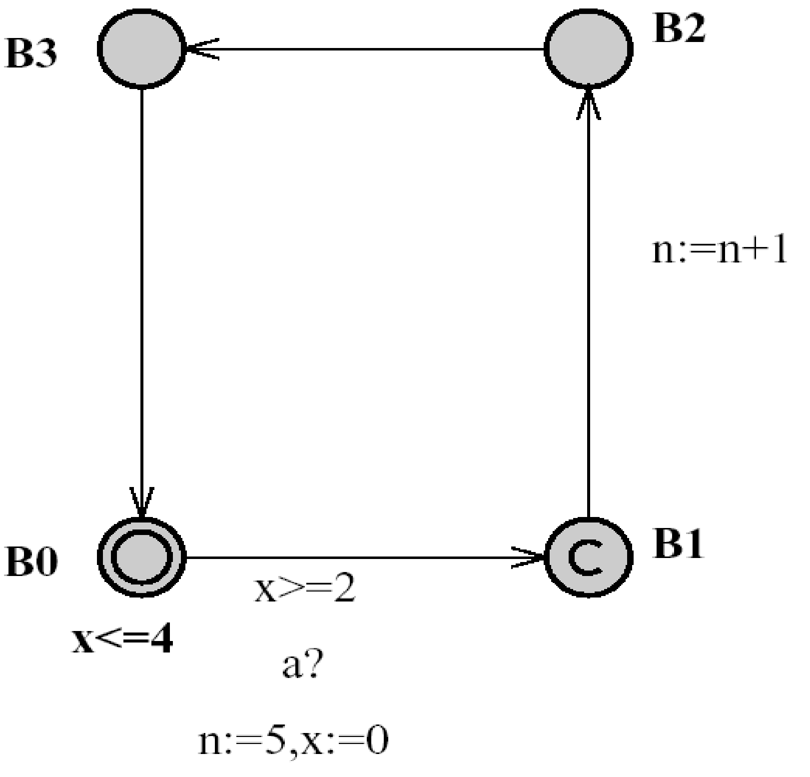
Example – textual format (.xta file)

```
clock x, y;
int n;
chan a;

process A {
  state A0 { y<=6 }, A1, A2, A3;
  init A0;
  trans A0 -> A1 {
    guard y>=3;
    sync a!;
    assign y:=0;
  },
  A1 -> A2 {
    guard y>=4;
  },
  A2 -> A3 {
  guard n==5;
  }, A3 -> A0;
}
```

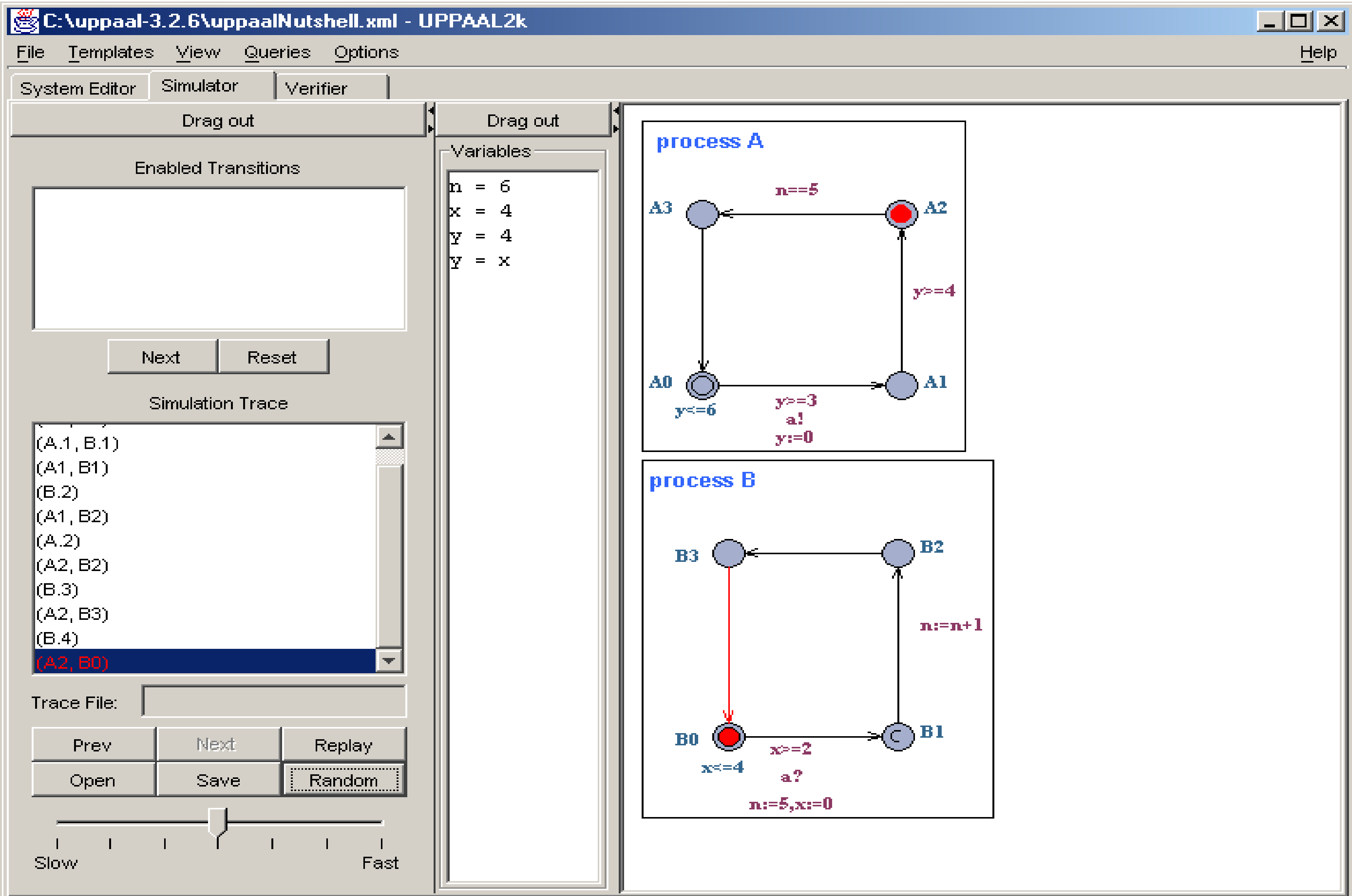


Example (cont.)

```
process B {
  state B0 { x<=4 }, B1, B2, B3;
  commit B1;
  init B0;
  trans B0 -> B1 {
    guard x>=2;
    sync a?;
    assign n:=5,x:=0;
  },
  B1 -> B2 {
    assign n:=n+1;
  },
  B2 -> B3 {
  }, B3 -> B0;
}

system A, B;
```

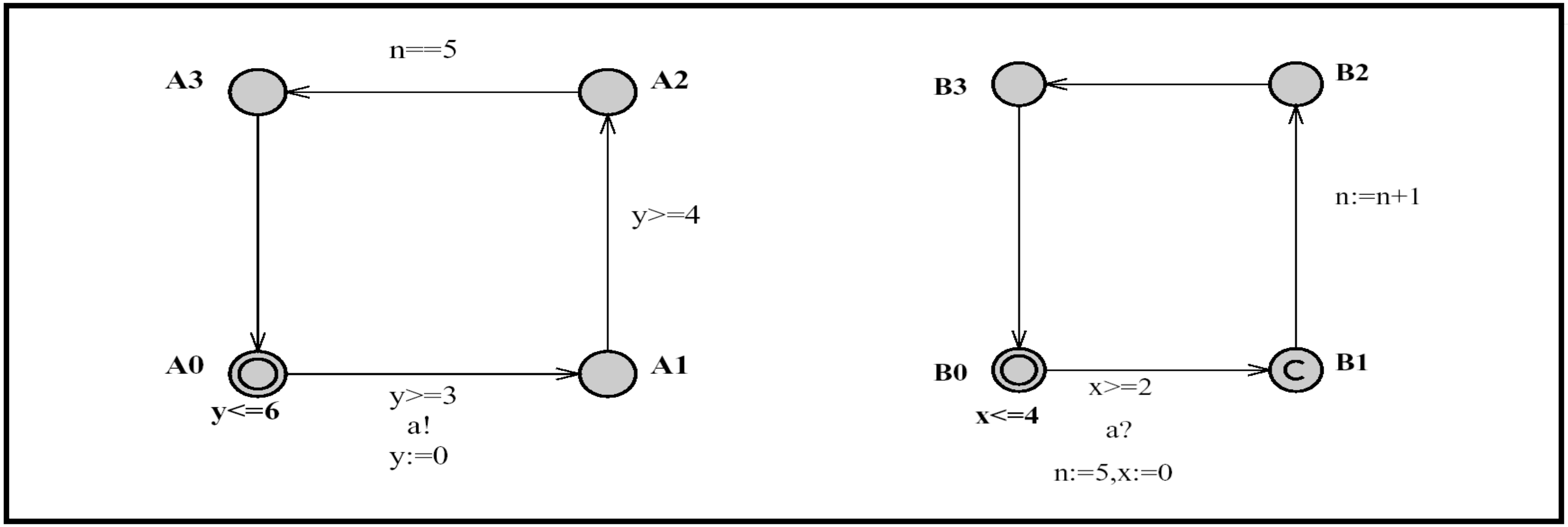


Example (cont.)

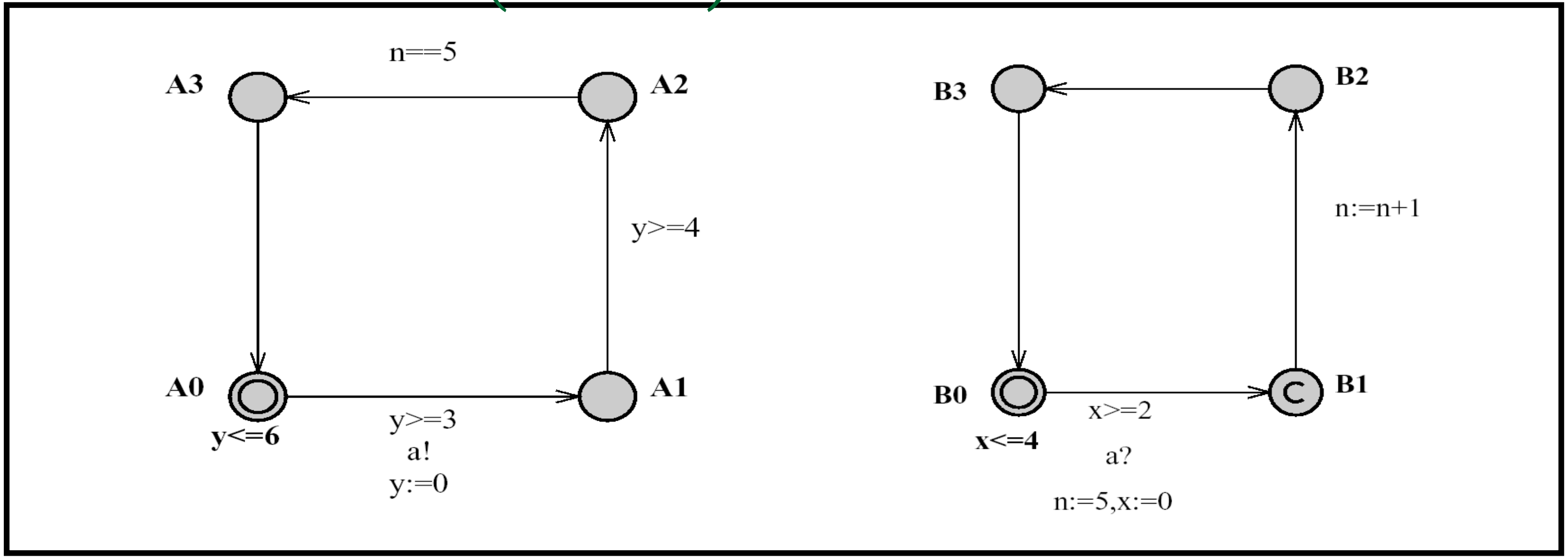


Transitions

- **Delay transitions** – if none of the invariants of the nodes in the current state are violated, time may progress without making a transition; e.g., from $((A_0,B_0),x=0,y=0,n=0)$, time may elapse 3.5 units to $((A_0,B_0),x=3.5,y=3.5,n=0)$, but time cannot elapse 5 time units because that would violate the invariant on B_0.



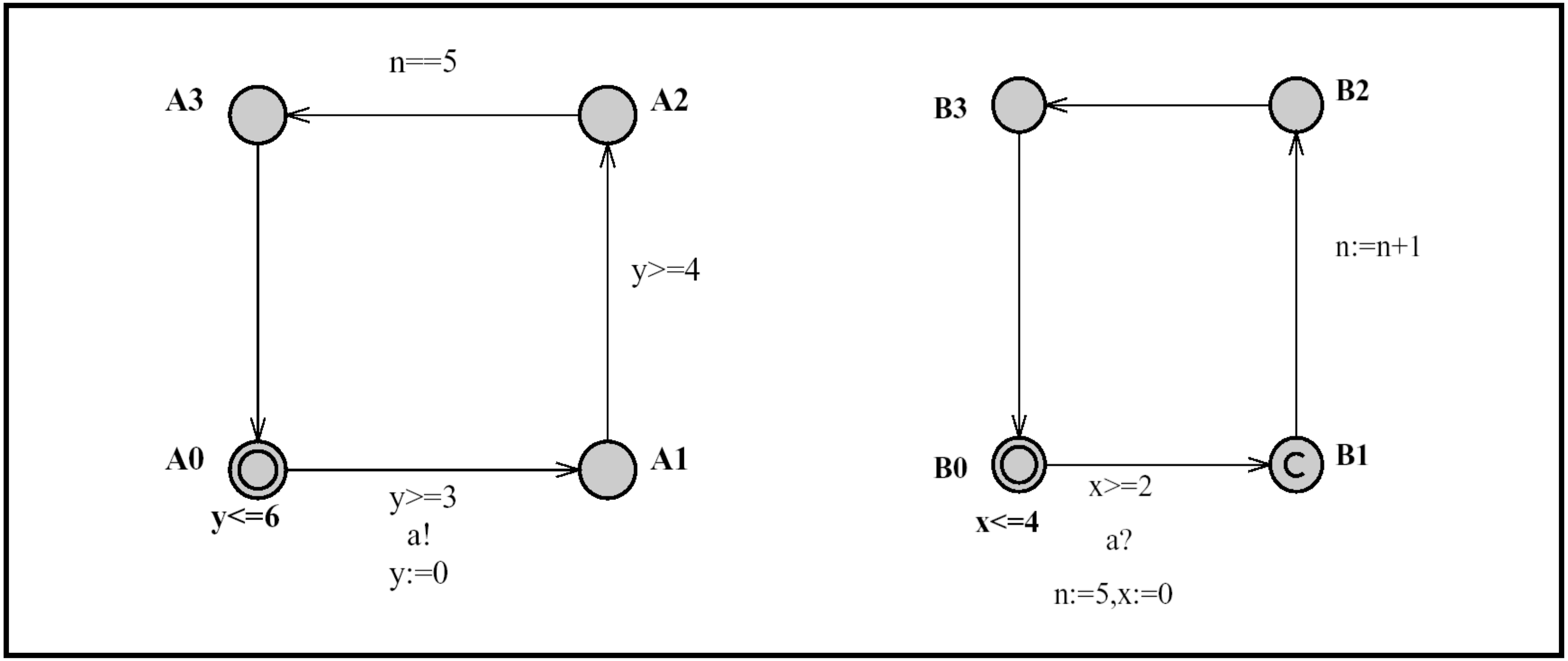
Transitions (cont.)



- **Action transitions** – if two complementary edges of two different components are enabled in a state, then they can synchronize; e.g., from $((A_0,B_0),x=0,y=0,n=0)$ the two components can synchronize to $((A_1,B_1),x=0,y=0,n=5)$.

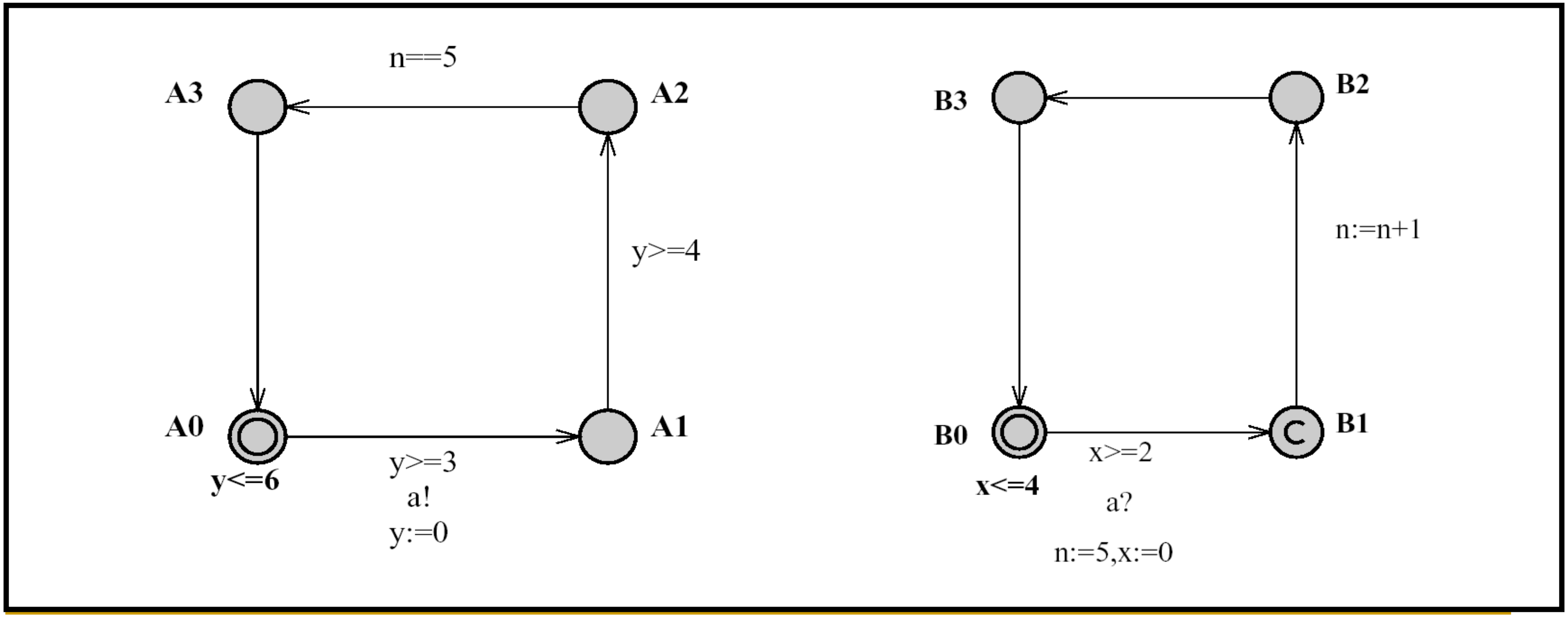
Urgent Channels

- When two components can synchronize on an **urgent channel,** no further delay is allowed; e.g., if channel **a** is urgent, time could not elapse beyond 3, e.g. in state $((A_0,B_0),x=3,y=3,n=0)$, synchronization on channel a is enabled.



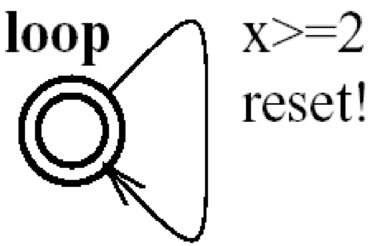
Committed Locations

- If one of the components is in a **committed node**, no delay is allowed to occur and any action transition must involve the component committed to continue; e.g., in the state $((A_1,B_1),x=0,y=0,n=5)$, B_1 is committed, so the next state of the network is $((A_1,B_2),x=0,y=0,n=6)$.

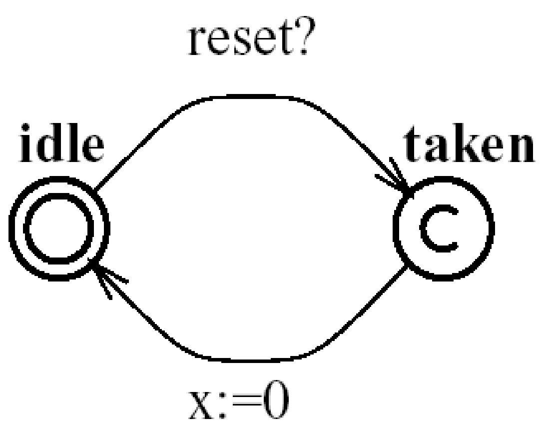


Modelling Time in UPPAAL Observer/Observable example

P1



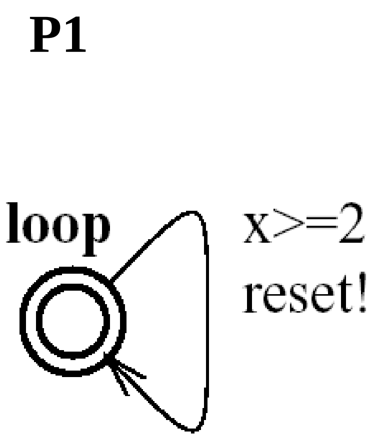
Obs1

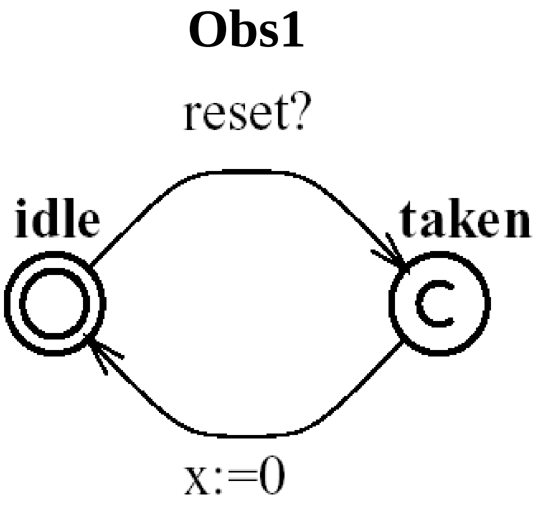



```
P1 = P();
Obs1 = Obs();
system P1,Obs1;
```

```
clock x;
chan reset;
```

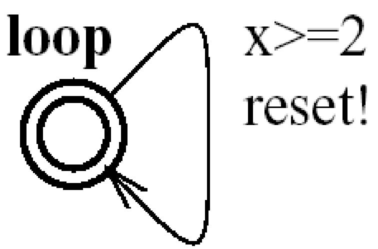
Observer/Observable example

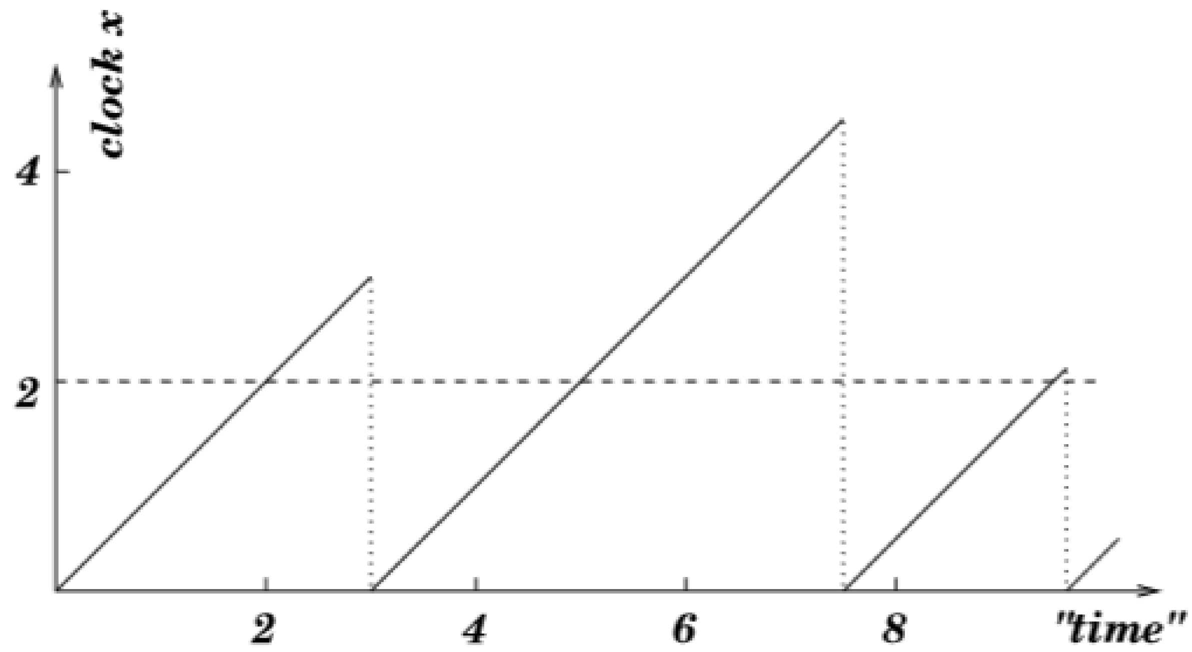




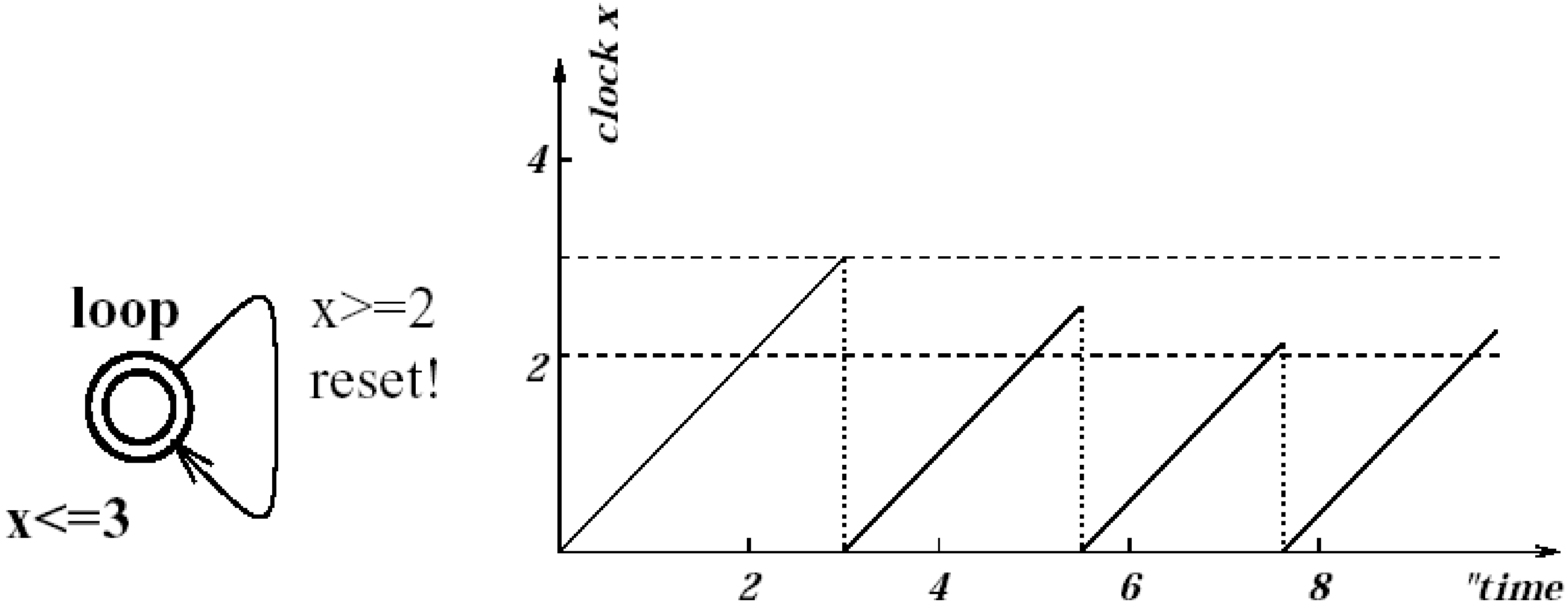
- Verification
  - A[](Obs1.taken imply x>=2)
  - E<>(Obs1.idle and x>3)
  - E<>(Obs1.idle and x>3000)

Observer/Observable example
Time diagram



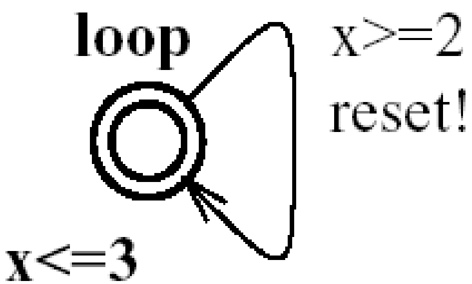


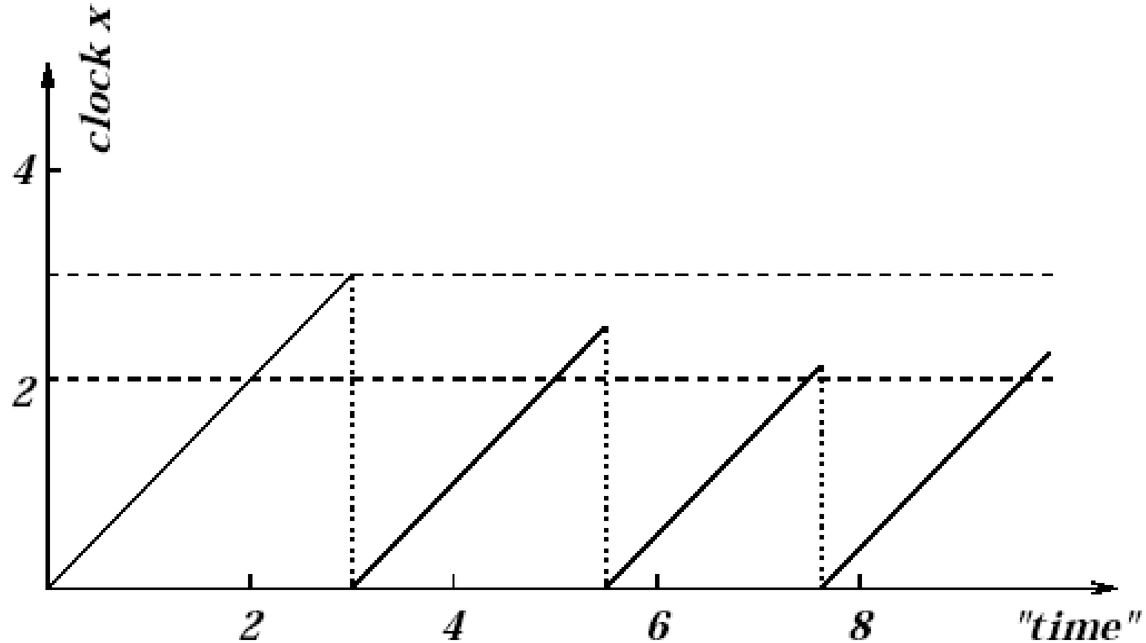
Observer/Observable example
Adding an Invariant – the new behavior



The invariant is a ***progress condition***: the system is not allowed to stay in the **loop** state more than 3 time units

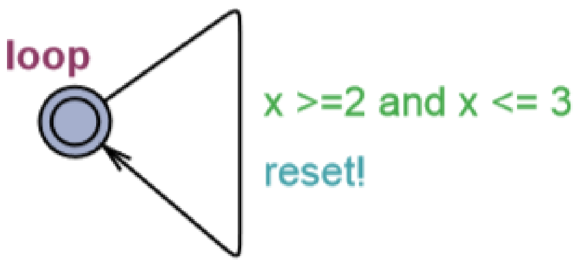
Observer/Observable example
Adding an Invariant – the new behavior

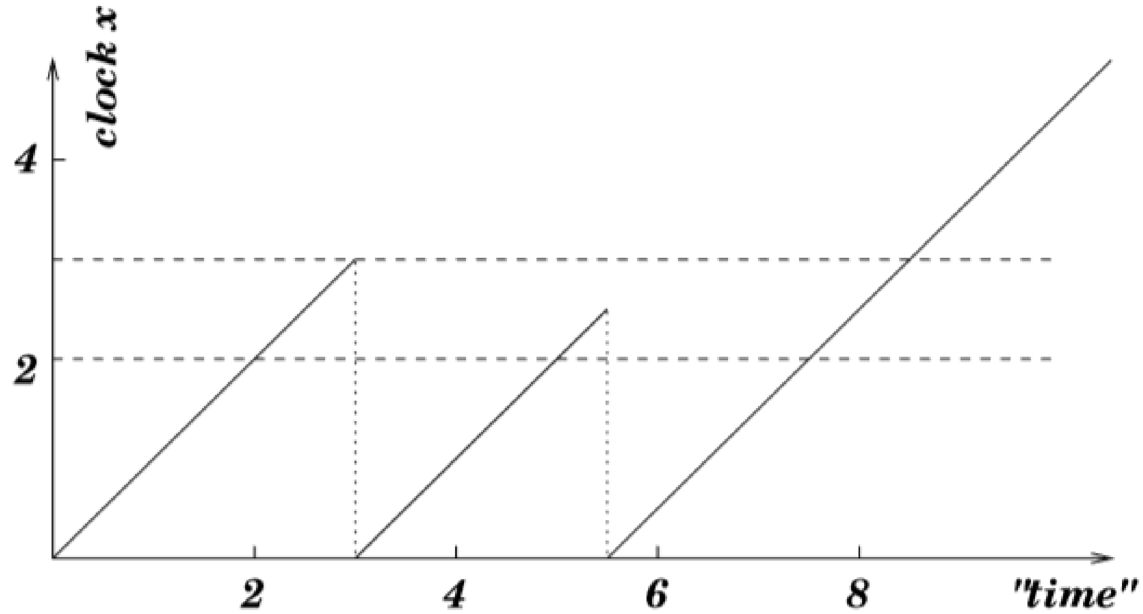




A[] Obs.taken imply (x>=2 and x<=3)
E<> Obs.idle and x>2
A[] Obs.idle imply x<=3
E<> Obs.idle and x>3 // FALSE

Observer/Observable example
No invariant but a new guard: the new behavior



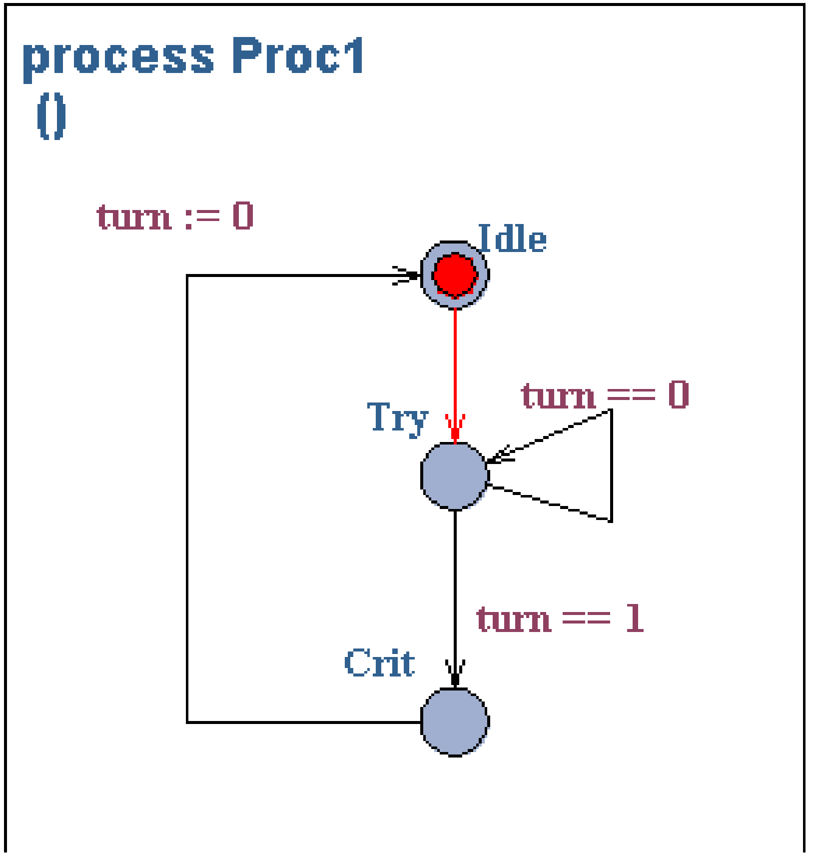


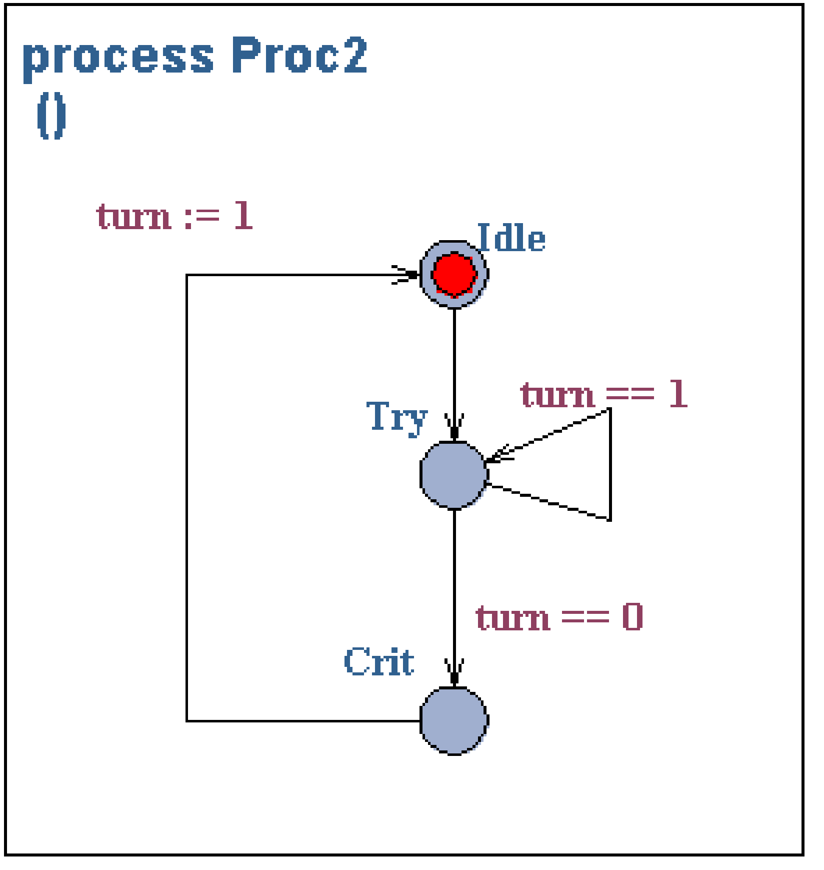
We can wait in a **loop** more than 3 units of time
Because no progress condition deadlock is possible:
after 3 time units the transition can not be taken anymore.
A[] x>3 imply not Obs.taken

Mutual Exclusion Program

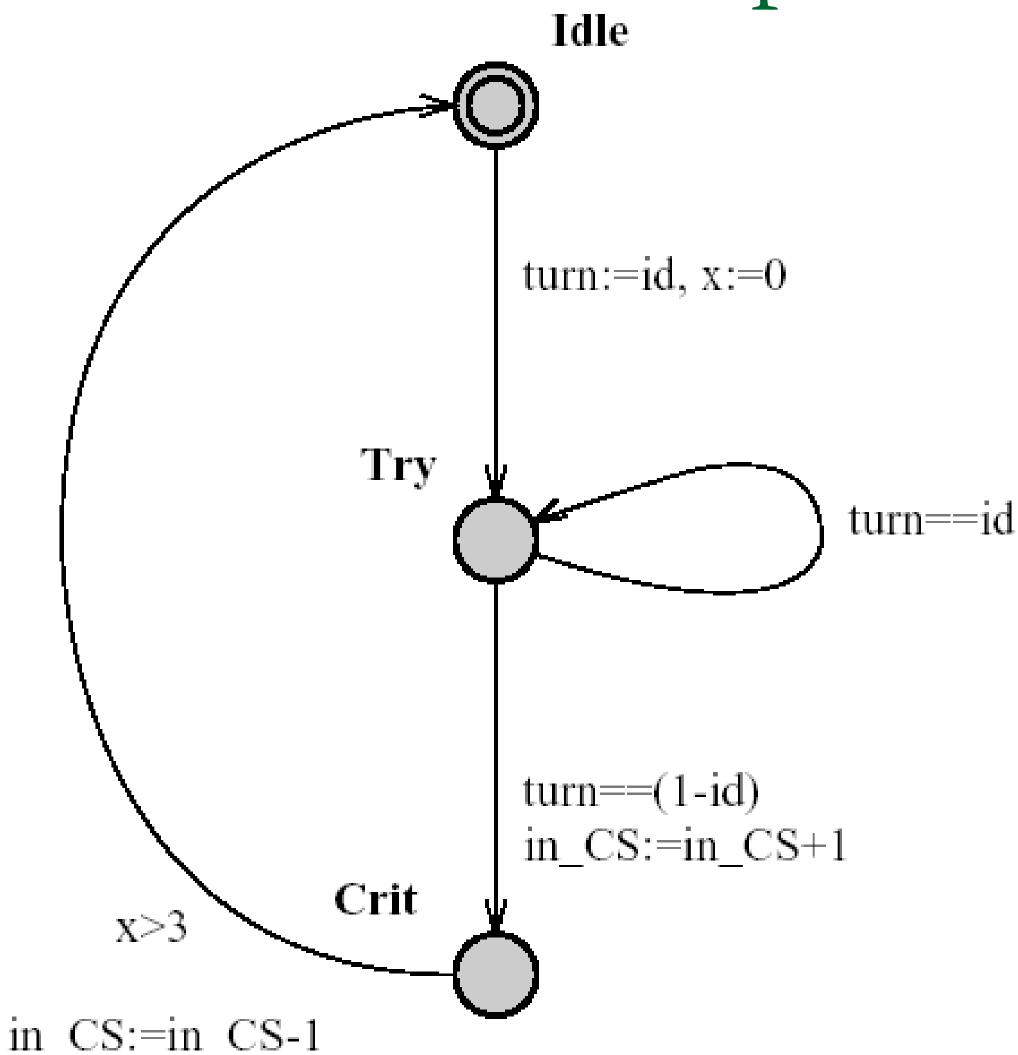
```
P1 :: while True do
        wait(turn==0);
        Critical_section;
        turn:=0;
      endwhile
||
P2 :: while True do
        wait(turn==1);
        Critical_section;
        turn:=1;
      endwhile
```

Translation to UPPAAL

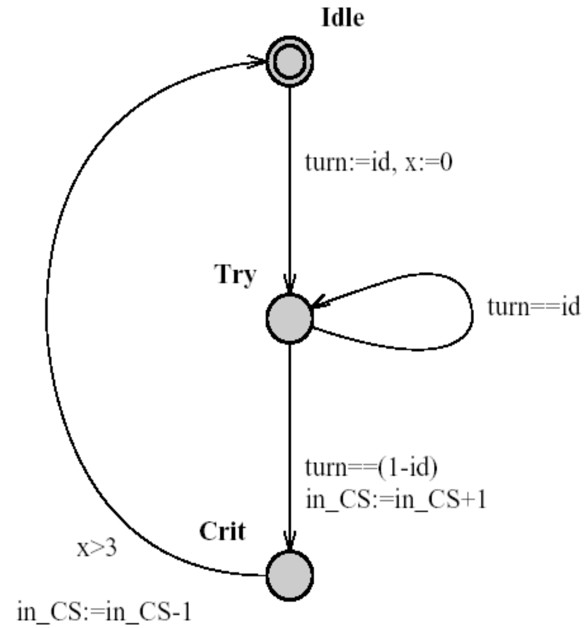




Mutual Exclusion for n processes



Example (mutex2.xta)




```
//Global declarations
int turn;
int in_CS;

//Process template
process P(const id){
clock x;
state Idle, Try, Crit;
init Idle;
trans Idle -> Try{assign turn:=id, x:=0; },
Try -> Crit{guard turn==(1-id); assign in_CS:=in_CS+1; },
Try -> Try{guard turn==id; },
Crit -> Idle{guard x>3; assign in_CS:=in_CS-1; };
}

//Process assignments
P1:=P(1);
P2:=P(0);

//System definition.
system P1, P2;
```

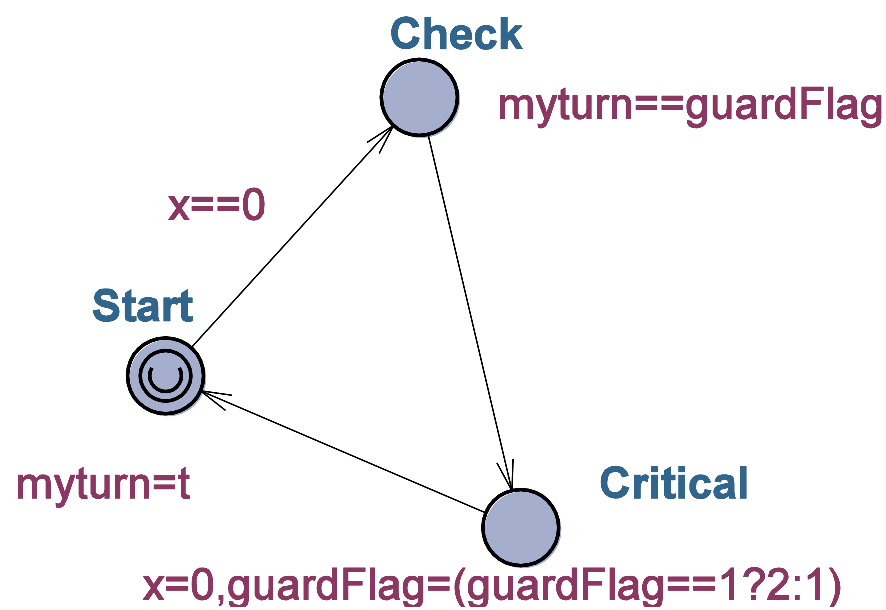
Peterson's Mutual Exclusion Algorithm

The algorithm

```
flag[0]    = 0
flag[1]    = 0
turn       = 0

P0: flag[0] = 1
    turn = 1
    while( flag[1] && turn == 1 );
            // do nothing
    // critical section
    ...
    // end of critical section
    flag[0] = 0

P1: flag[1] = 1
    turn = 0
    while( flag[0] && turn == 0 );
            // do nothing
    // critical section
    ...
    // end of critical section
    flag[1] = 0
```

Example: Mutual Exclusion Algorithm

```
typedef int[1,2]
  turn;
typedef int[1,2]
  flag;

flag guardFlag=1;

P1 = T1(1);
P2 = T1(2);

system P1,P2;
```



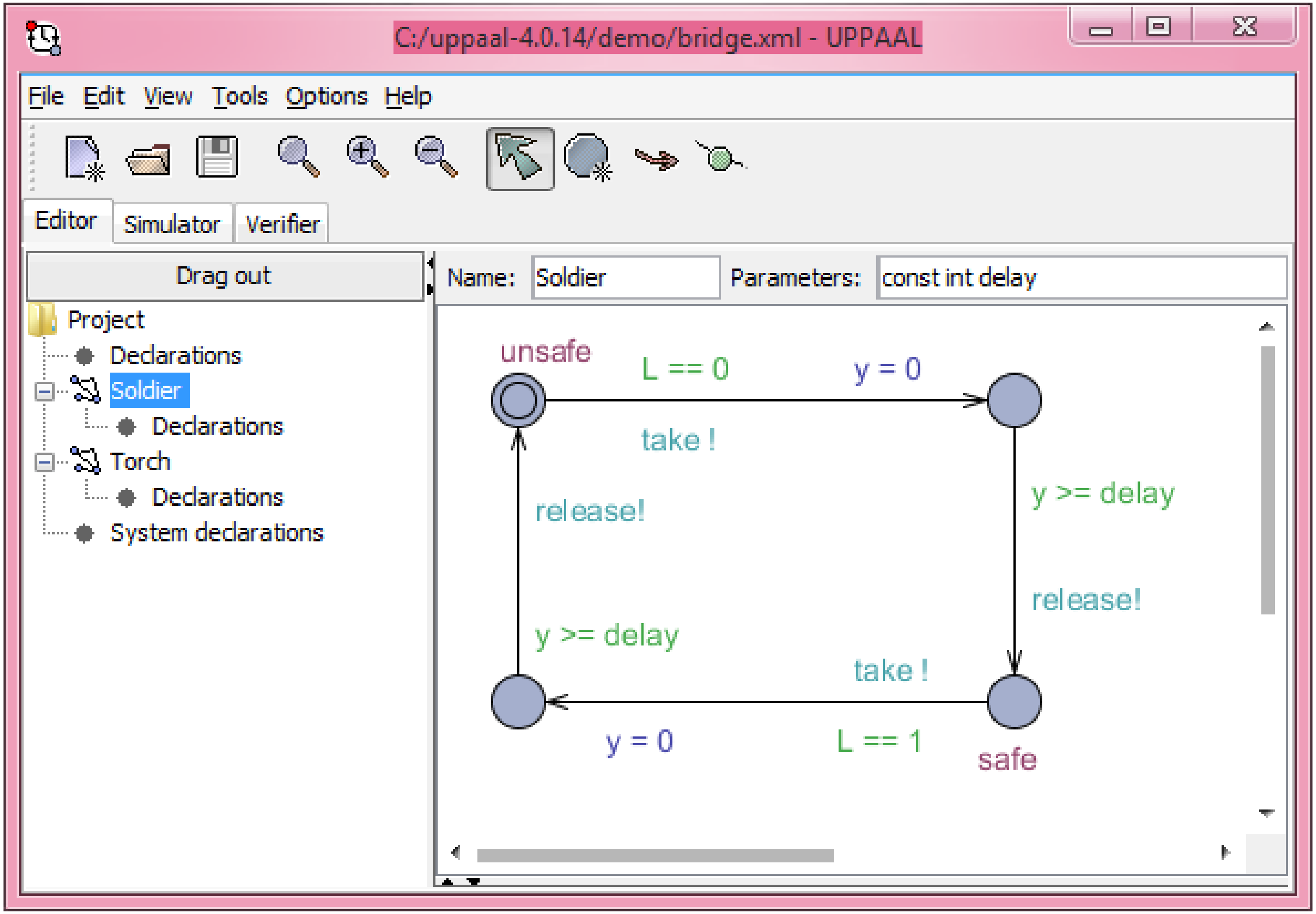
Four Vikings problem

- Four vikings are about to cross a damaged bridge in the middle of the night. The bridge can only carry two of the vikings at the time and to find the way over the bridge the vikings need to bring a torch. The vikings need 5, 10, 20 and 25 minutes (one-way) respectively to cross the bridge.
- Does a schedule exist which gets all four vikings over the bridge within 60 minutes?

Global Declarations

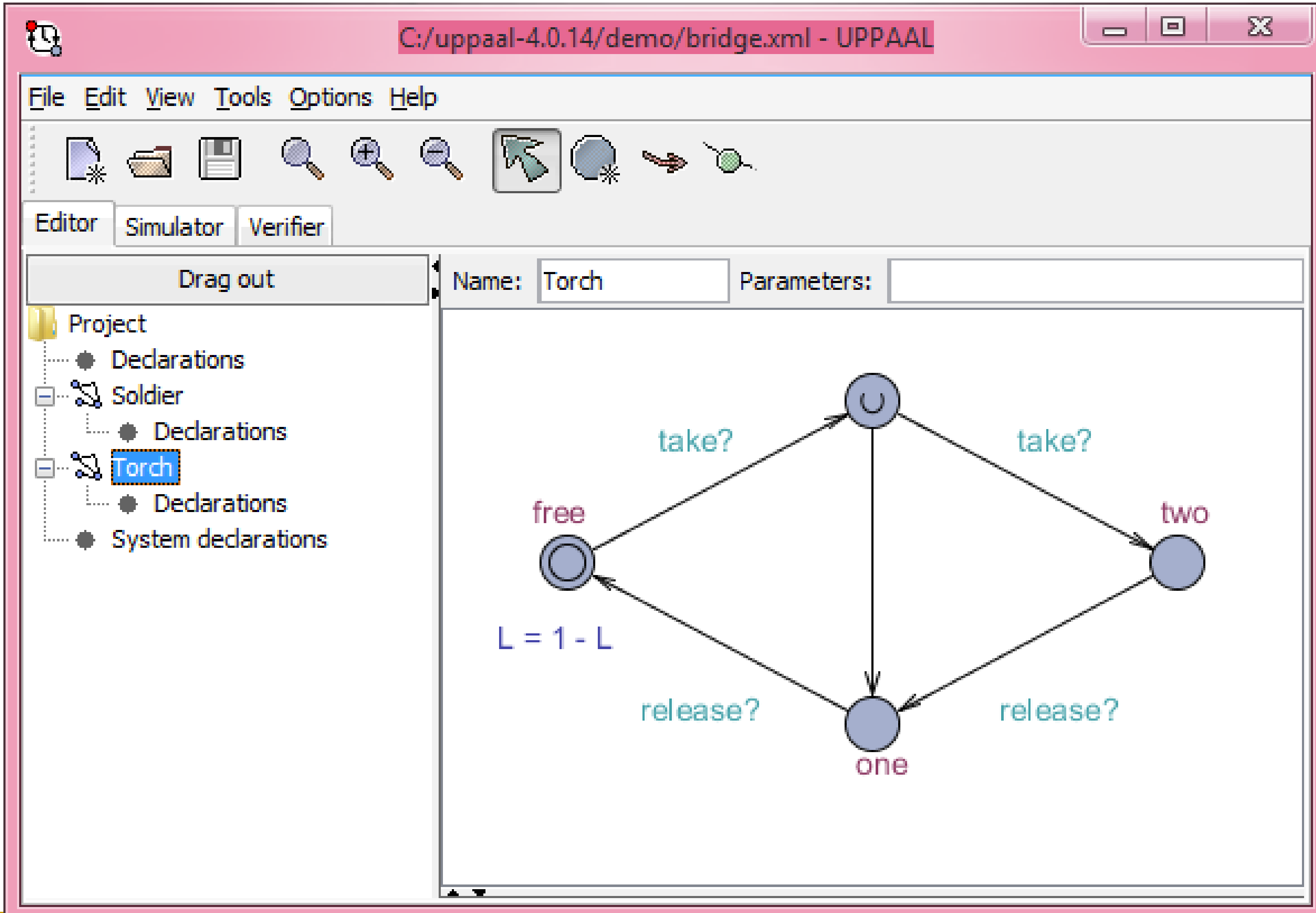
- chan take, release; // Take and release torch
- int [0,1] L; // The side the torch is on
- clock time; // Global time

Model of a Soldier (Viking)

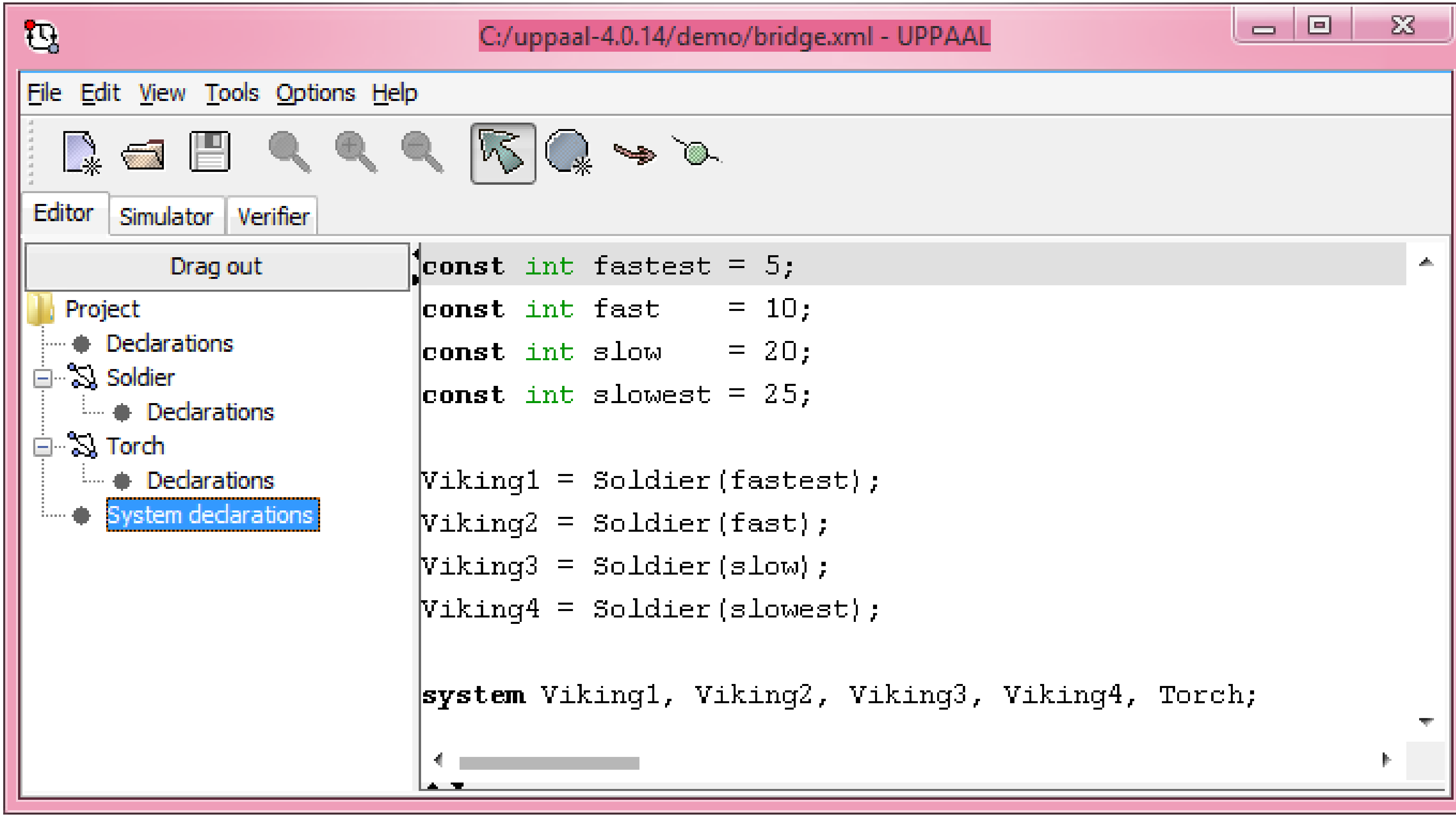


Declaration of a parameter delay (const int delay)
Declaration of clock y;

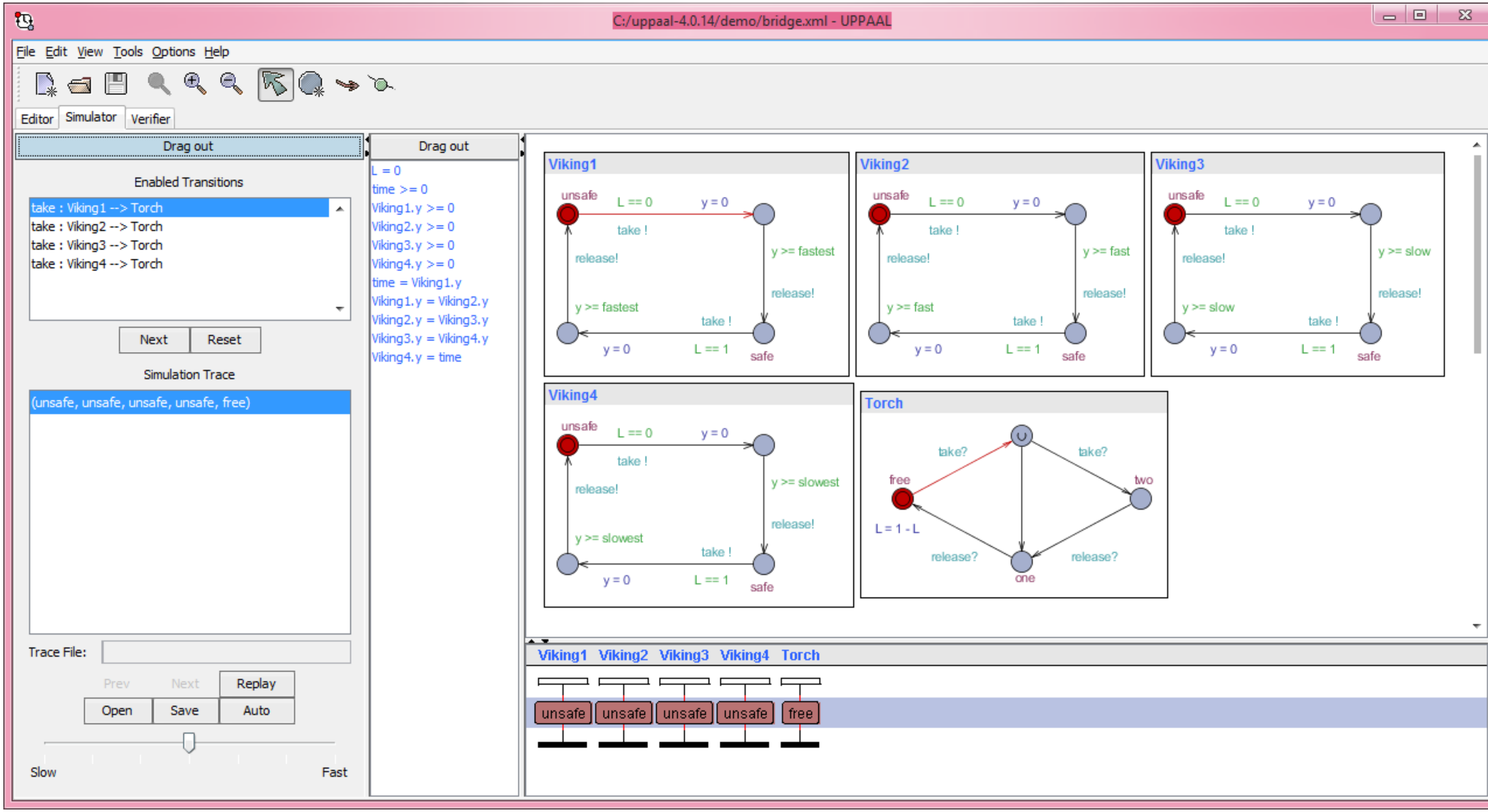
Model of a Torch (Viking)



System Declarations



Simulation trace



Verification of properties

A[] not deadlock // The system is deadlock free.

E<> Viking1.safe // Viking 1 can cross the bridge.
E<> Viking2.safe
E<> Viking3.safe

A[] not (Viking4.safe and time<slowest)

E<> Viking4.safe imply time>=slowest

E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe

Terima Kasih