Advanced System Software
（先端システムソフトウェア）

#7 (2018/10/29)

CSC.T431, 2018-3Q
Mon/Thu 9:00-10:30, W832

Instructor: Takuo Watanabe（渡部卓雄）
Department of Computer Science

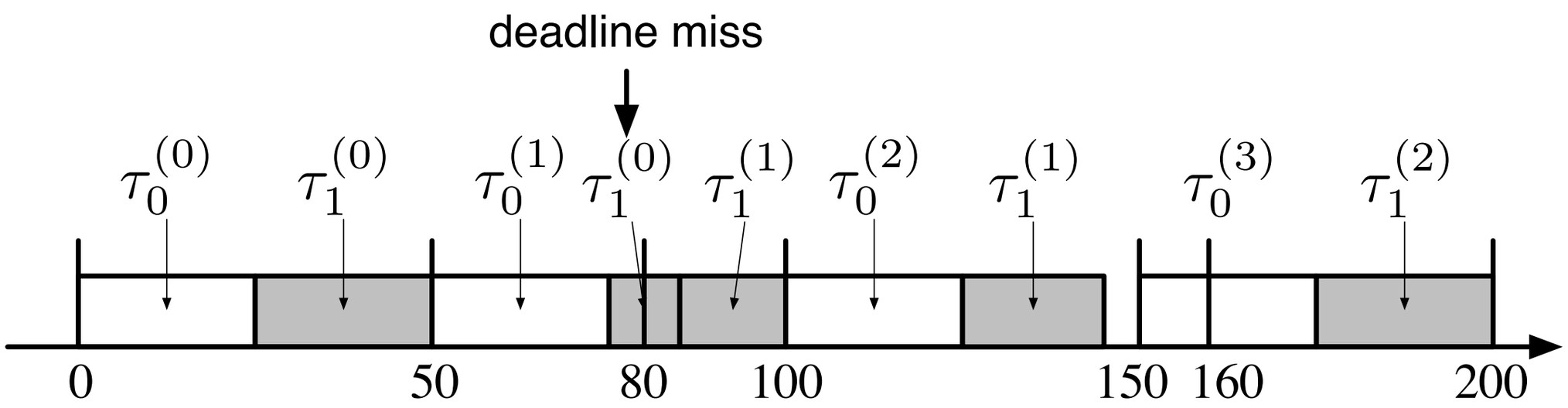
e-mail: takuo∅c.titech.ac.jp
http://www.psg.c.titech.ac.jp/~takuo/
ext: 3690, office: W8E-805

Agenda

- Real-Time Task Scheduling Algorithm (2)
- Specifying and Verifying Real-Time Systems

Example 2

$$\Gamma = \{\tau_0, \tau_1\},\ \tau_0 = (50, 25, 0),\ \tau_1 = (80, 35, 0)$$

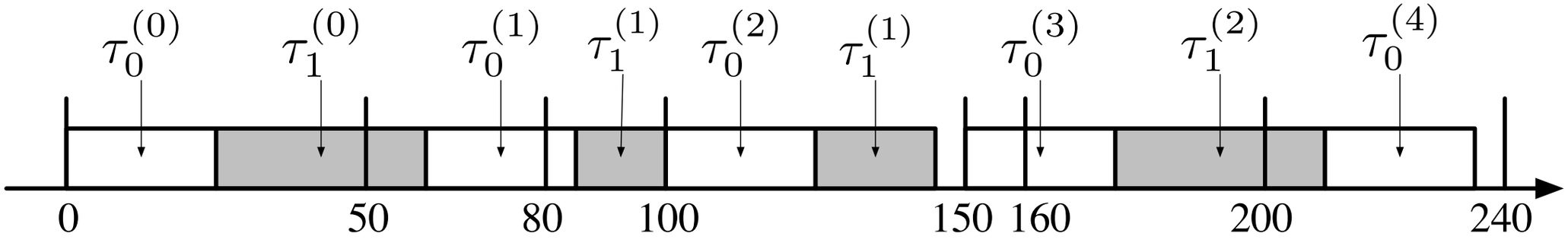


Γ is not feasible (deadline misses occur) with RM algorithm.

Earliest Deadline First (EDF) Algorithm

- A preemptive, dynamic algorithm
- "The earlier the deadline, the higher priority"

- Example 2



Properties of EDF Scheduling

- A set of tasks $\Gamma = \{ \tau_1, \tau_2, ..., \tau_n \}$ is schedulable with EDF if and only if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

- EDF is optimal in a sense that if there is a schedule for Γ, then EDF can schedule it.

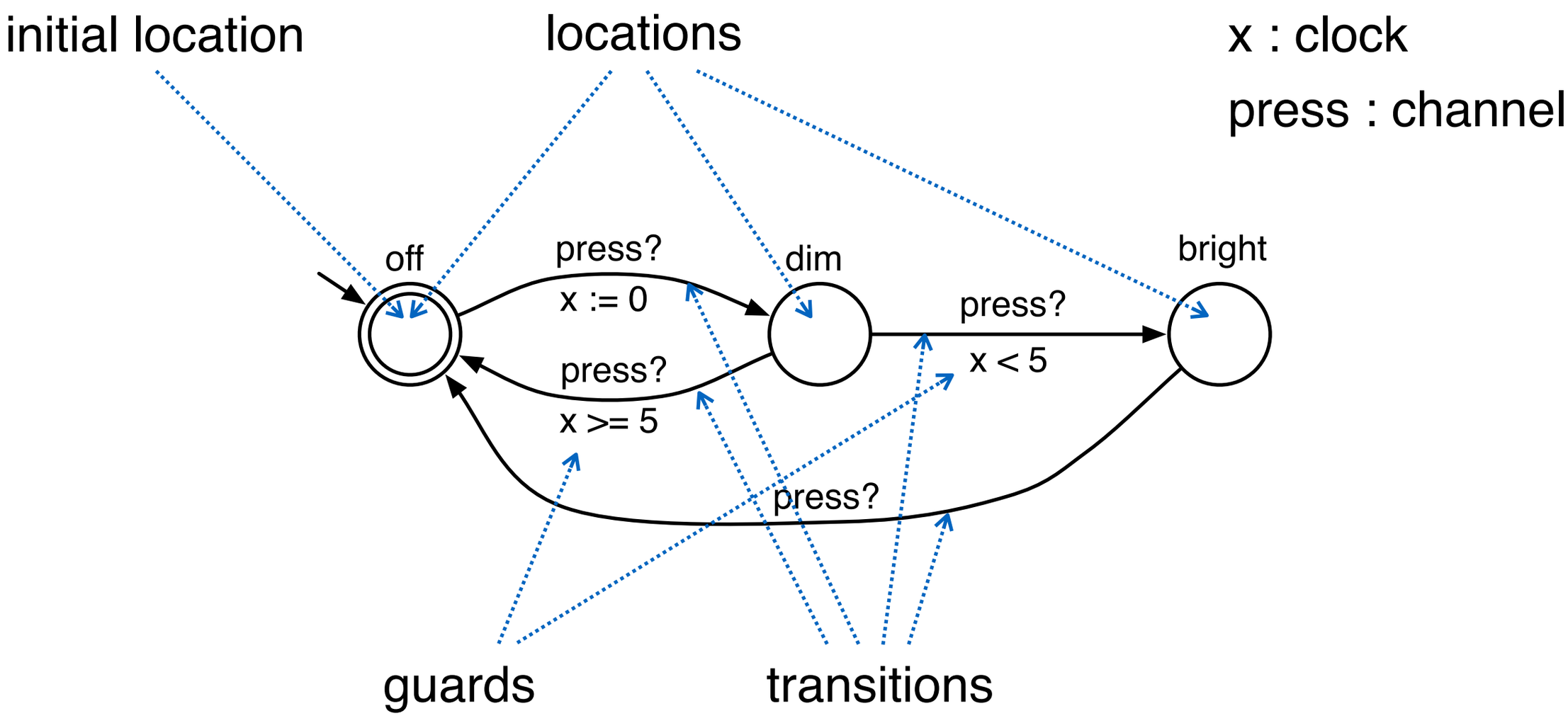
Specifying and Verifying Real-Time Systems

Timed Automata

- An operational model of real-time systems
  - by Rajeev Alur and David L. Dill
- An extension of Büchi-automata (finite-state automata over infinite strings)
  - Real-Valued Clocks
  - Guards and (Location) Invariants
  - Channels
  - Variables

- Automatic verification tool: UPPAAL
  - It is decidable whether a given control state is reachable.

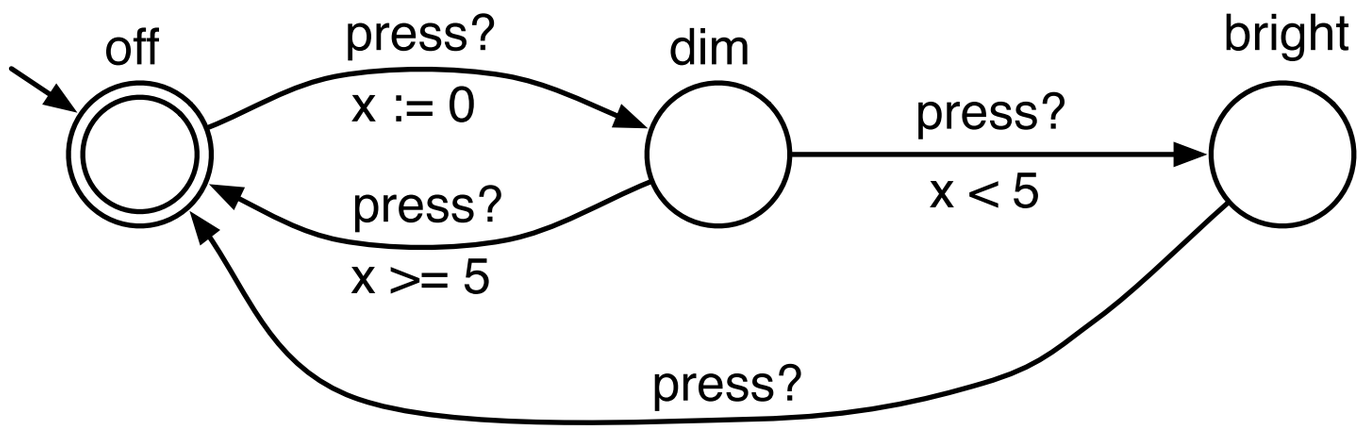
Ex: One-Button Lamp

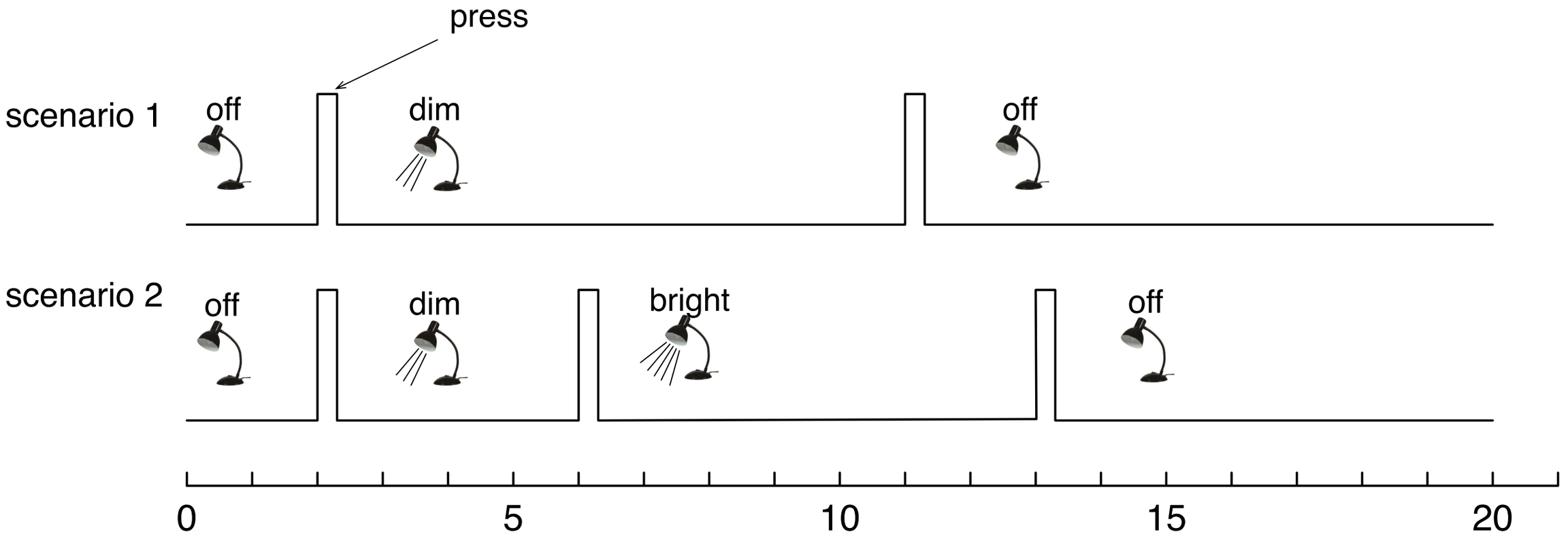


The Behavior of the Lamp (1)

- Suppose that the lamp is off.
- If you press the button once, the lamp turns on in *dim* mode.
- If you press the button two times *quickly*, the lamp turns on in *bright* mode.
  - The second press should be within five time units of the first press.
- If you press the button once when the lamp is on, it turns off.

The Behavior of the Lamp (2)

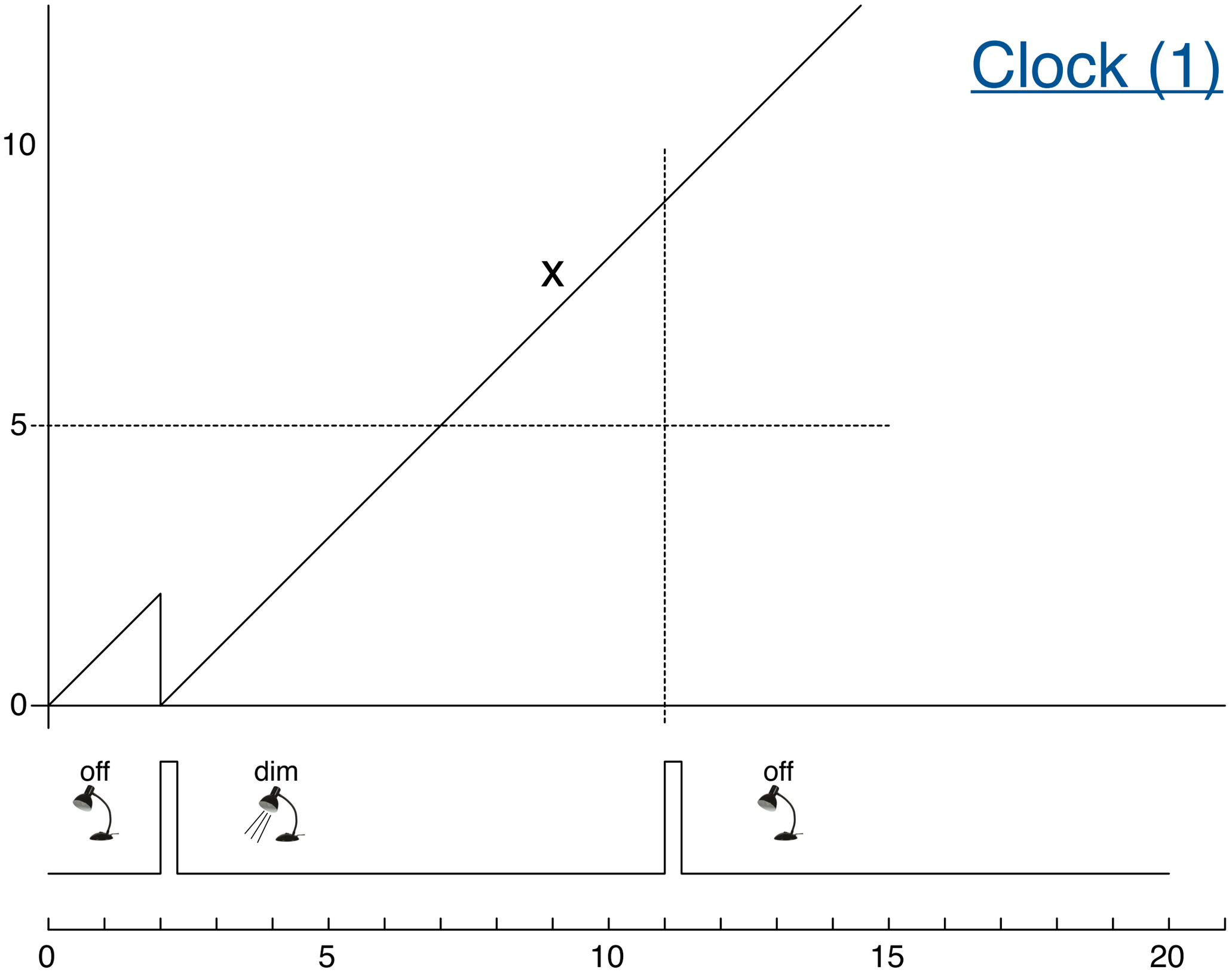




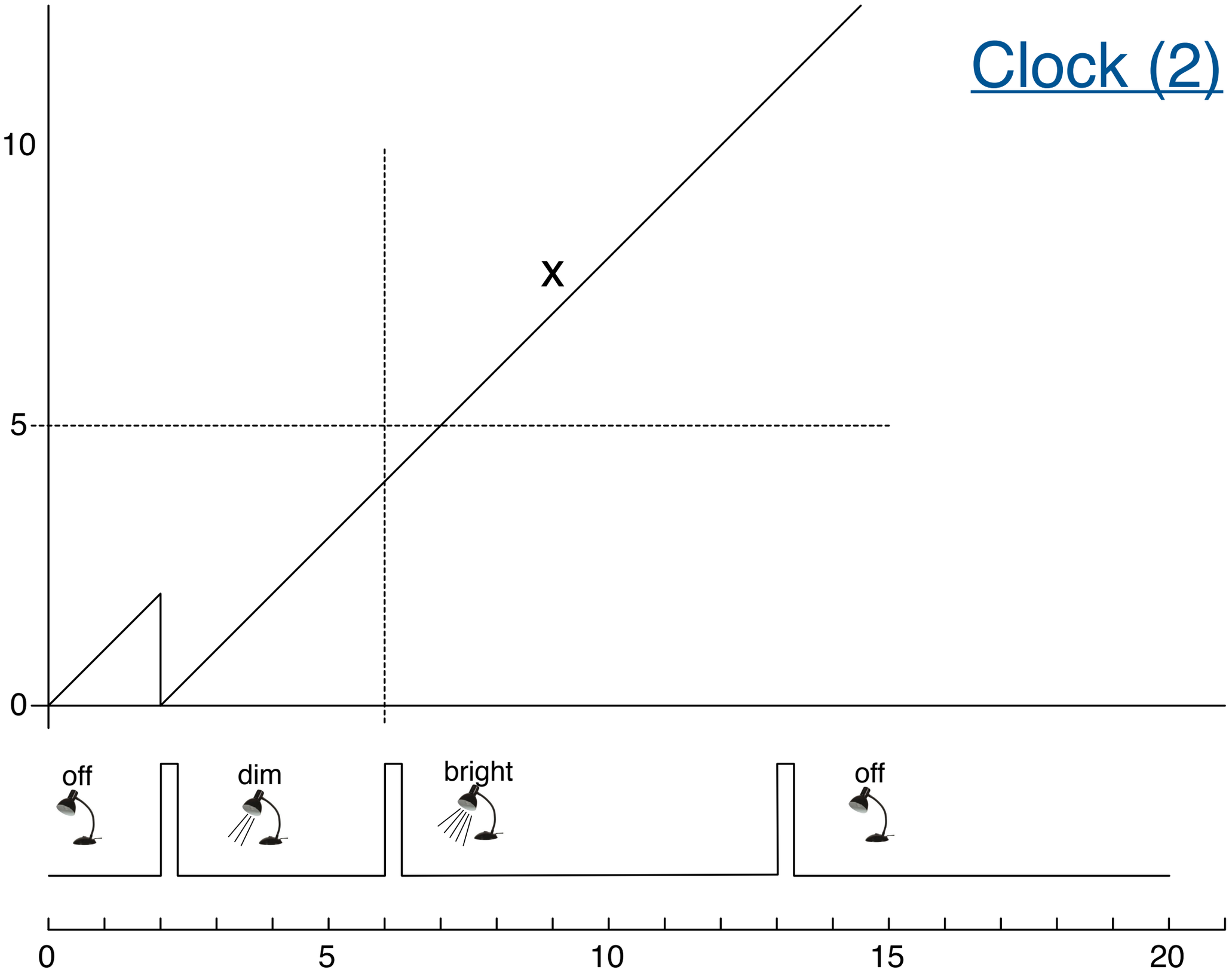
Clock

- A timed automaton may have one or more clocks (or clock variables) that take real values.
- The value of a clock increases at a constant rate while the automaton is in a location.
  - If a system has two or more clocks, the values of them increase at the same rate.
  - The value of a clock does not change on a state transition.
- Reset of Clocks
  - A transition may have one or more assignments to clocks.
  - E.g.: x := 0;

Clock (1)

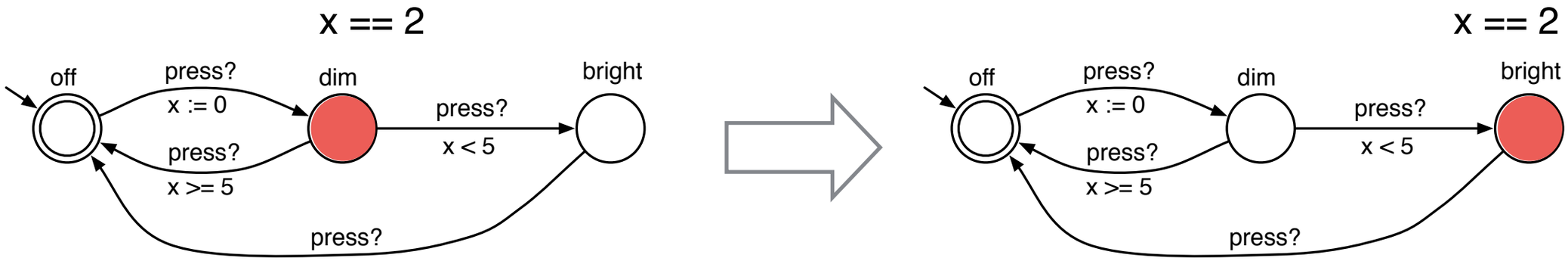


Clock (2)

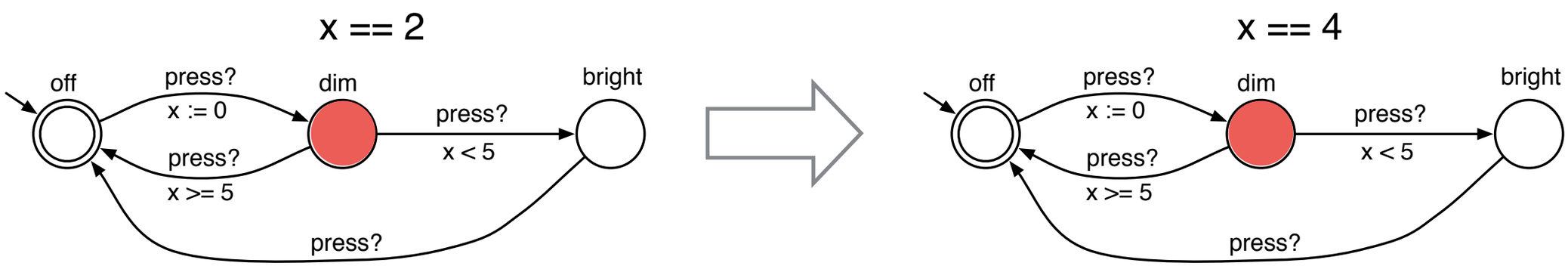


Two Kinds of Transitions

Changing locations (location transition)



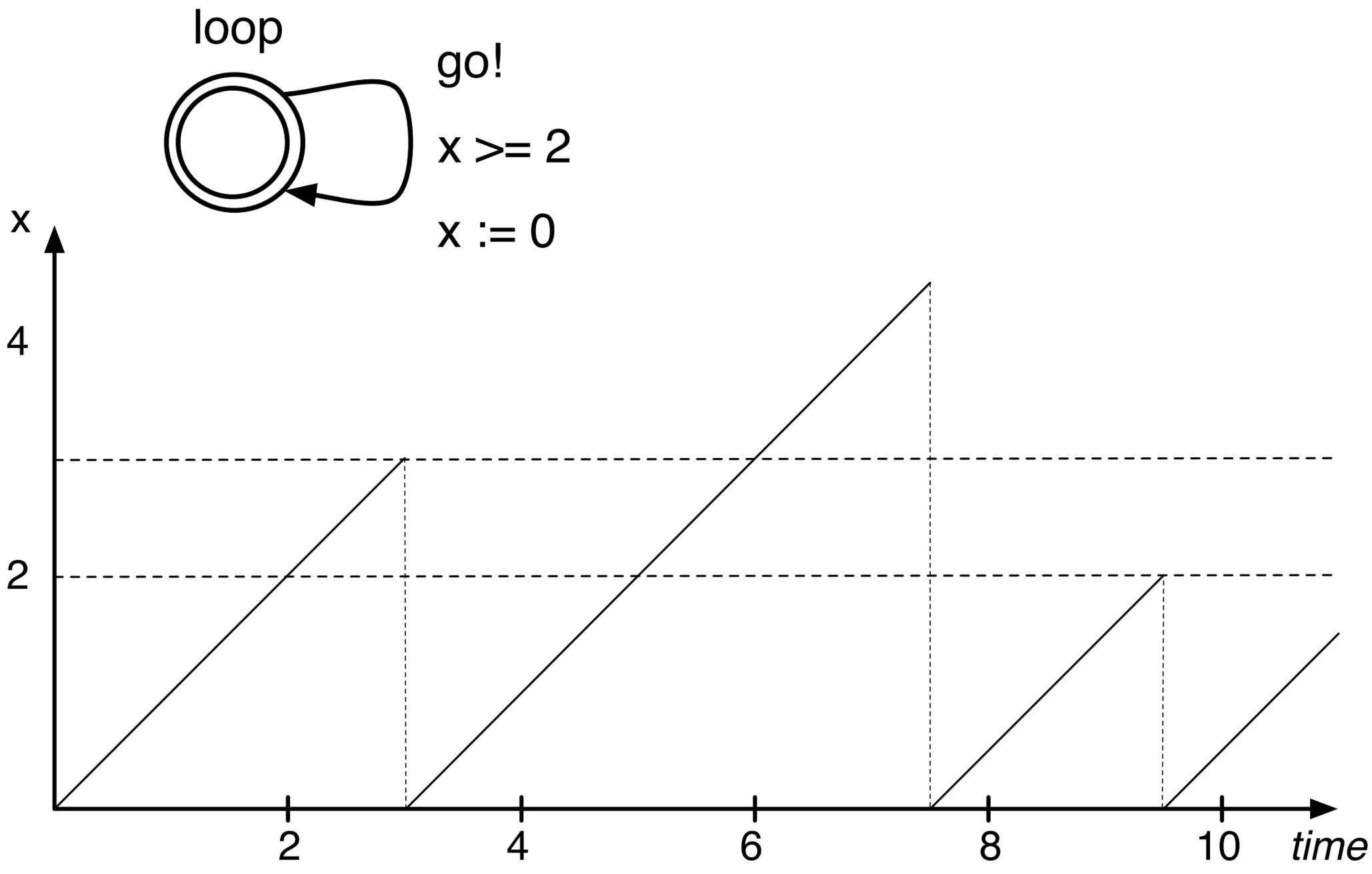
Increasing clocks



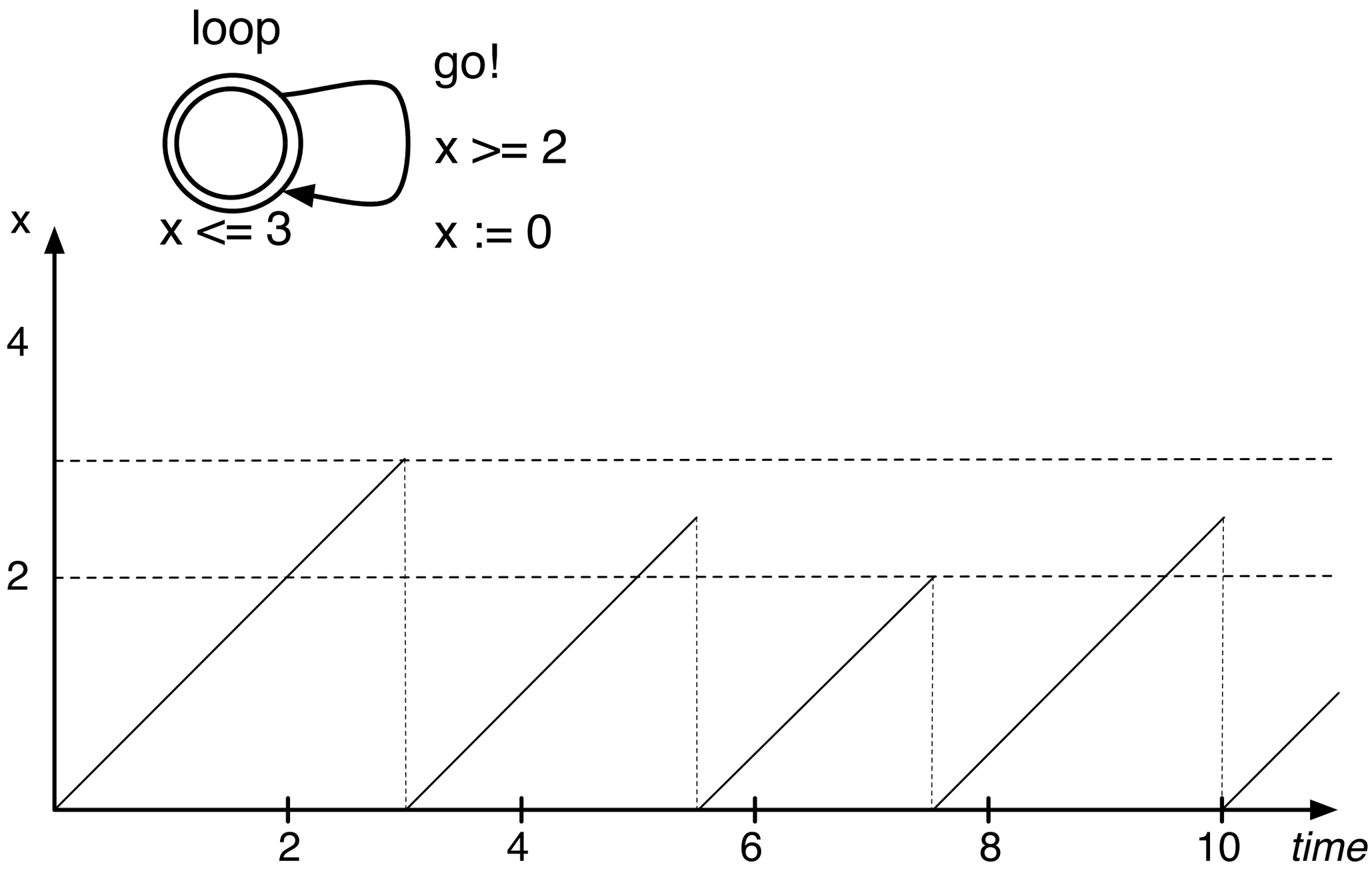
Guards and Location Invariants

- Guards
  - A location transition may have a Boolean expression called a guard.
  - The transition may happen only when the guard is satisfied.
- Location Invariants
  - A location may have a Boolean expression called a location invariant.
  - The automation may not be in the location when the location invariant is not satisfied.

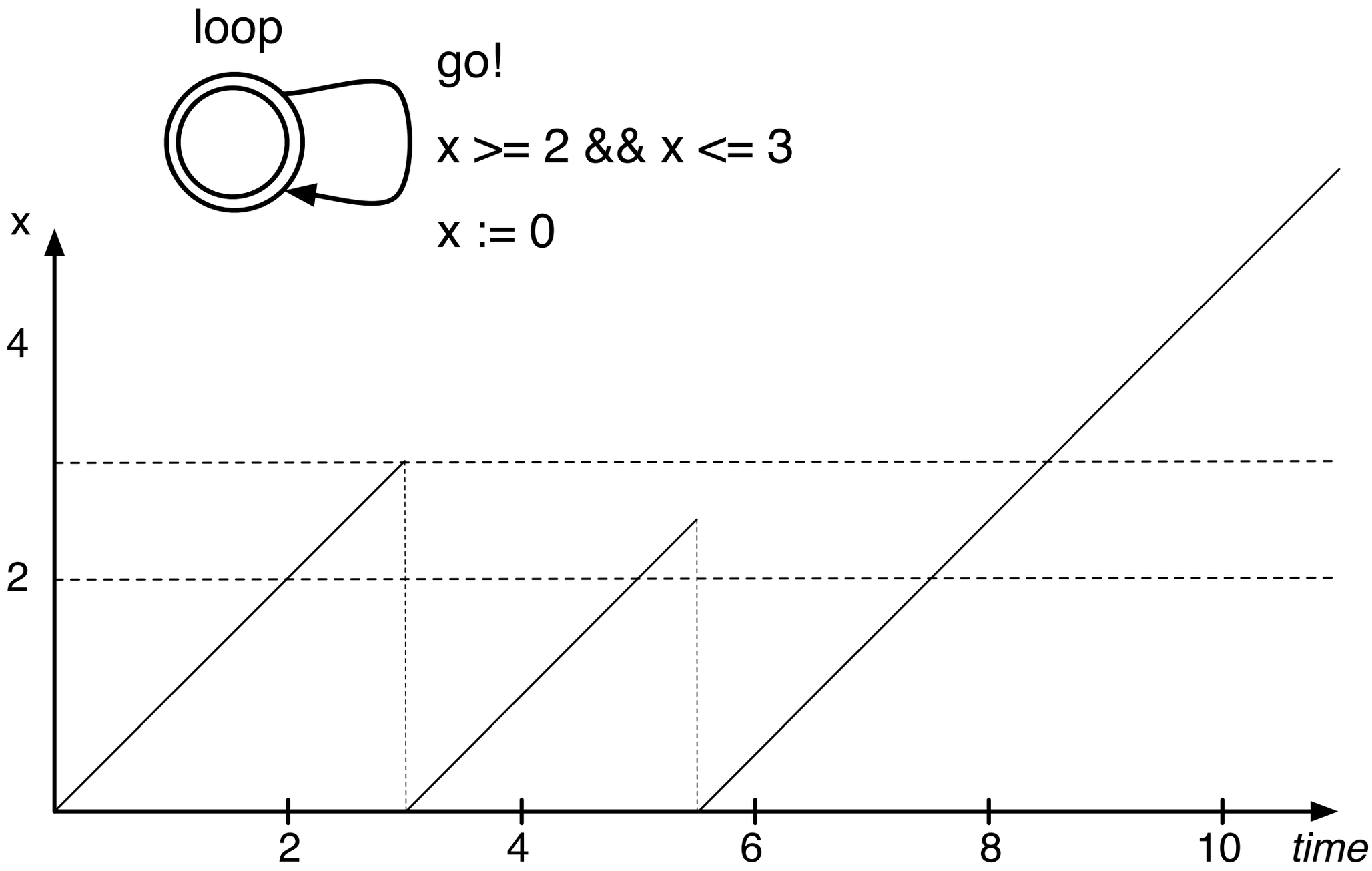
Guard and Reset



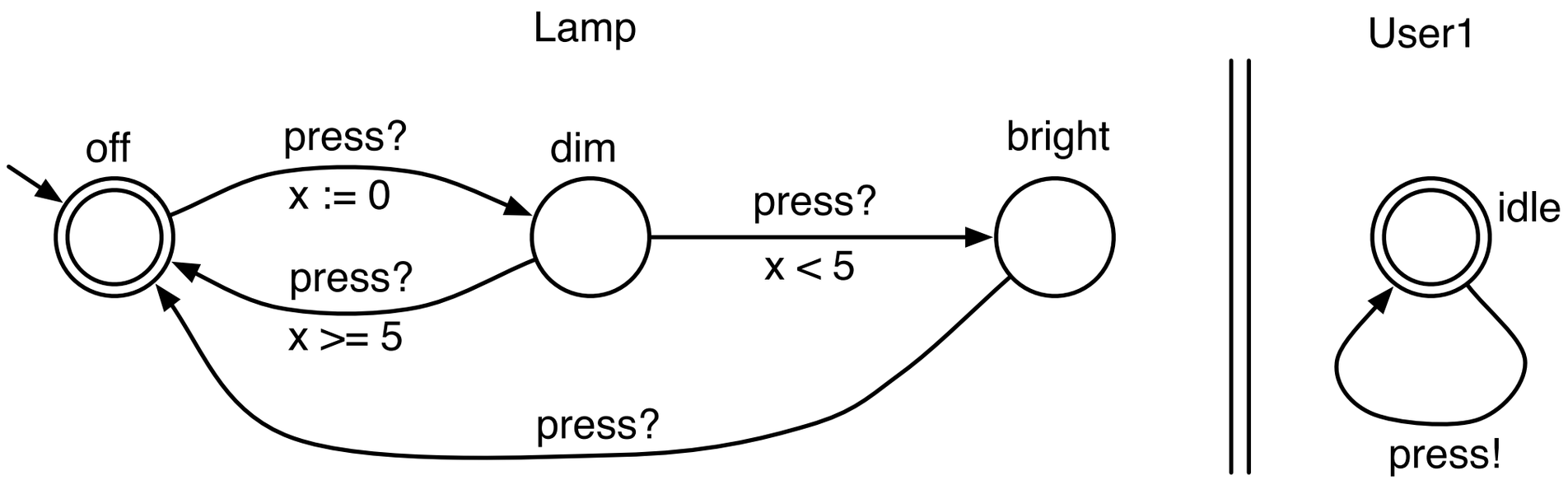
Location Invariant



Location Invariant vs. Guard



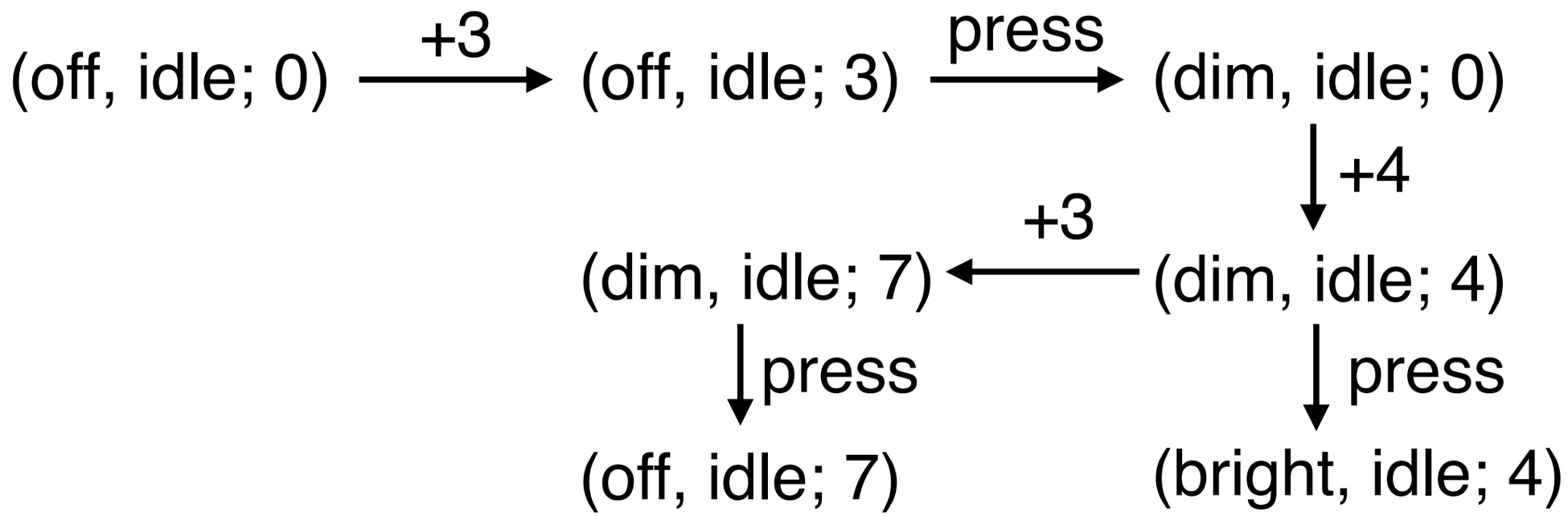
Network of Timed Automata



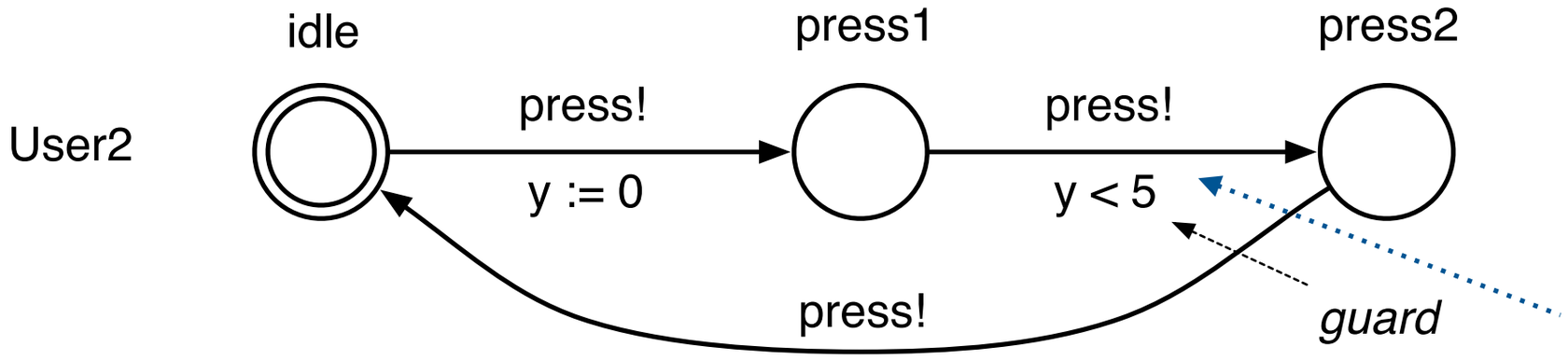
Two or more timed automata form a *network*. The above network models a system that consists of a lamp and a user. The transition label press! and press? respectively correspond to sending and receiving via a synchronous communication channel named *press*.

State (of a Network of TA)

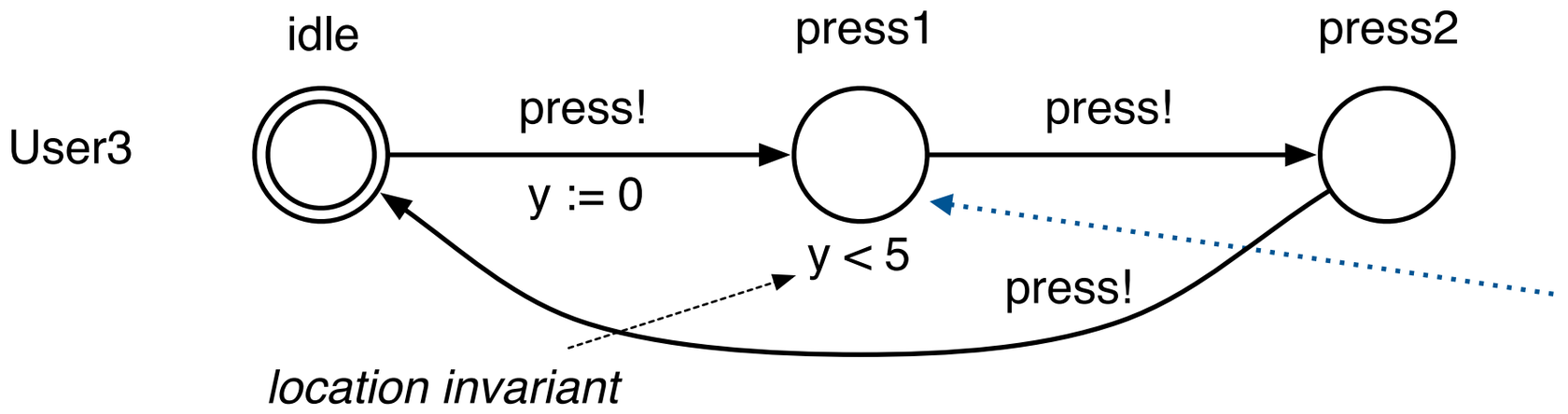
- Tuple of locations and clock values
  - Representing a `state' of a network of TA
  - Ex: (off, idle; 0)
    - Locations of Lamp and User
    - value of clock x of Lamp
- Transition of States



Modeling Other User Behaviors



This transition should take place while y < 5.

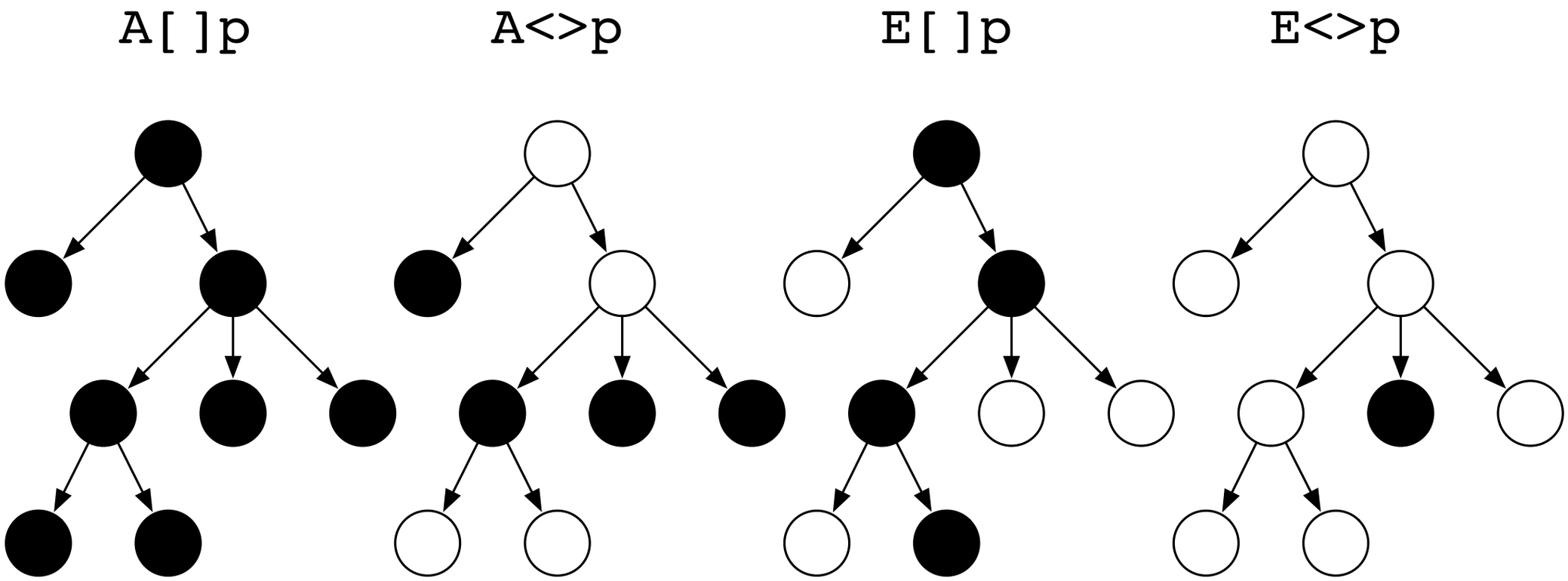


The system should leave this state before this condition becomes false.

System Properties

- Properties to be verified are written as formulae of TCTL (Timed Computational Tree Logic)
  - `A[] P`
    - P always holds in all possible paths
  - `A<> P`
    - P eventually holds in all possible paths
  - `E[] P`
    - P always holds at least in a path
  - `E<> P`
    - P eventually holds at least in a path

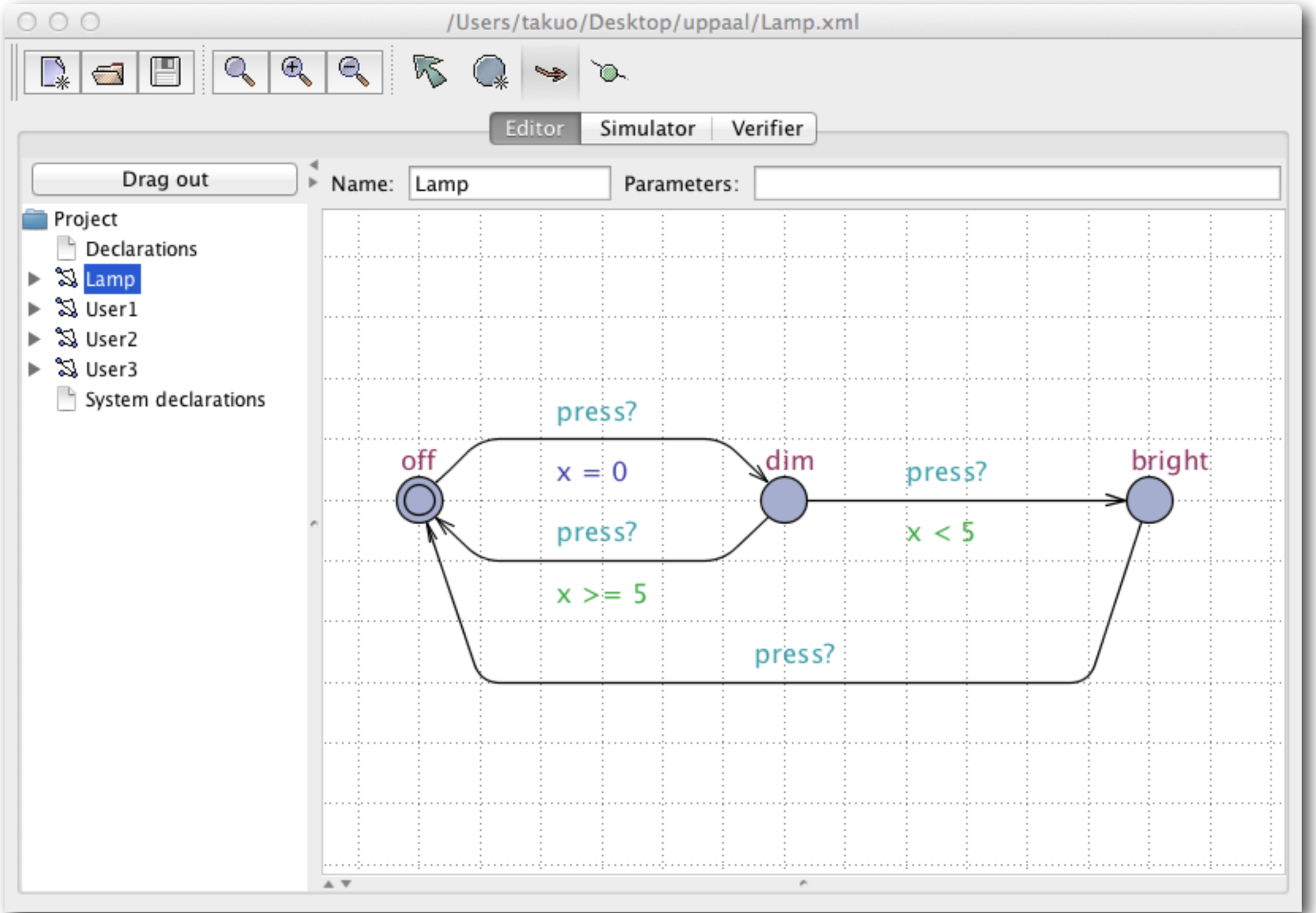
TCTL Formulae



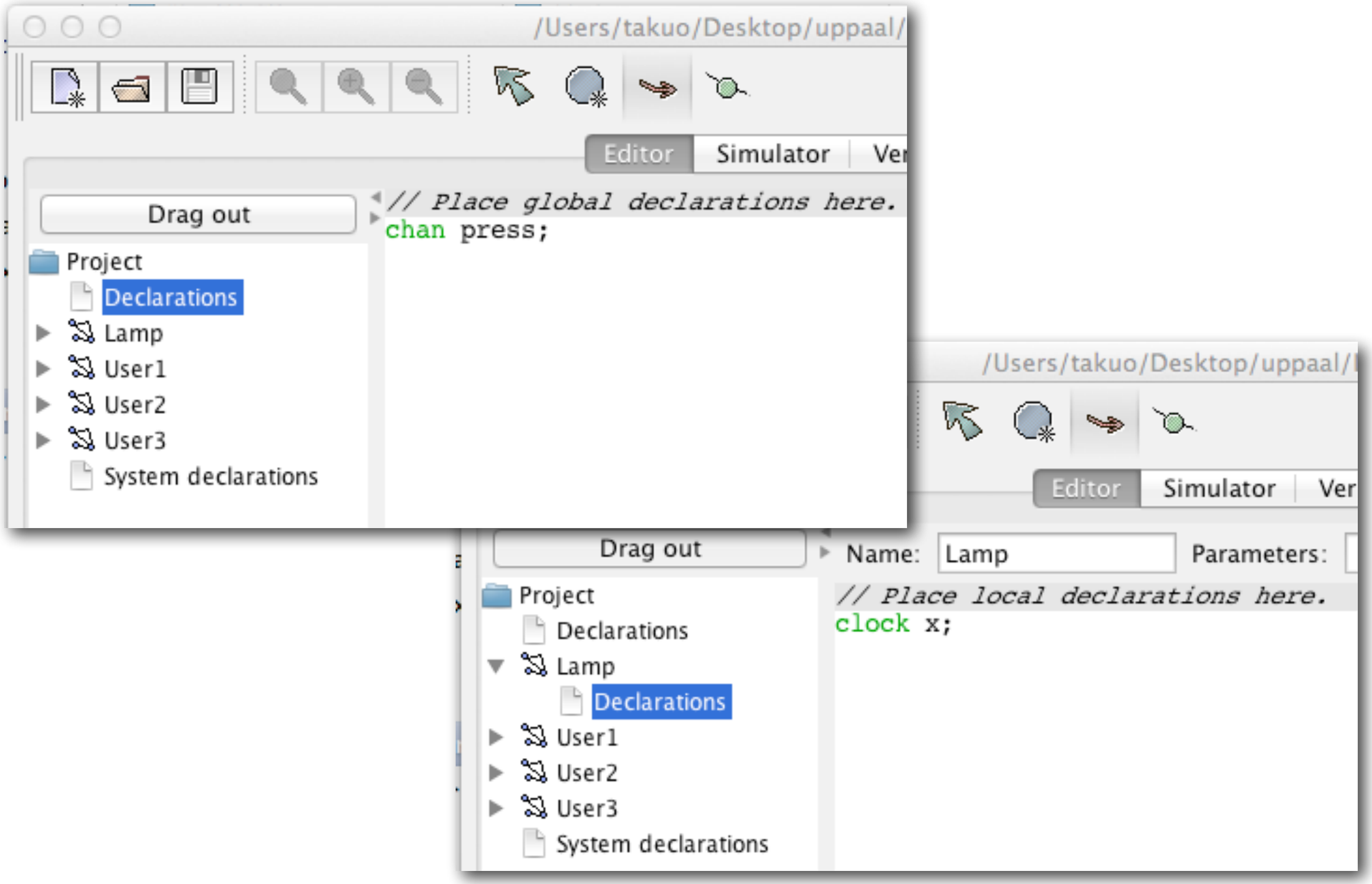
UPPAAL

- An integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata.
  - http://uppaal.org
  - Developed at Uppsala University & Aalborg University

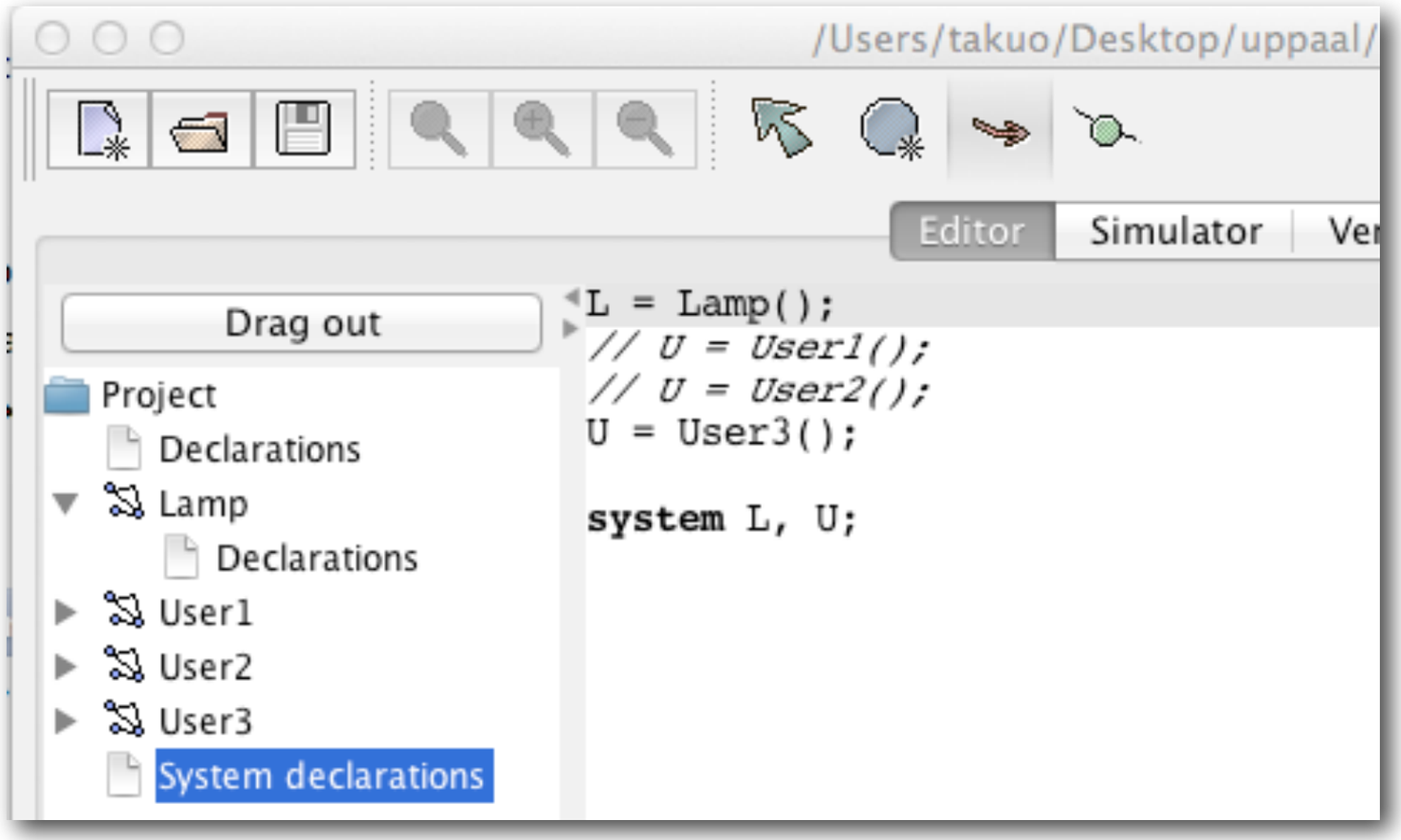
UPPAAL: Template Editor



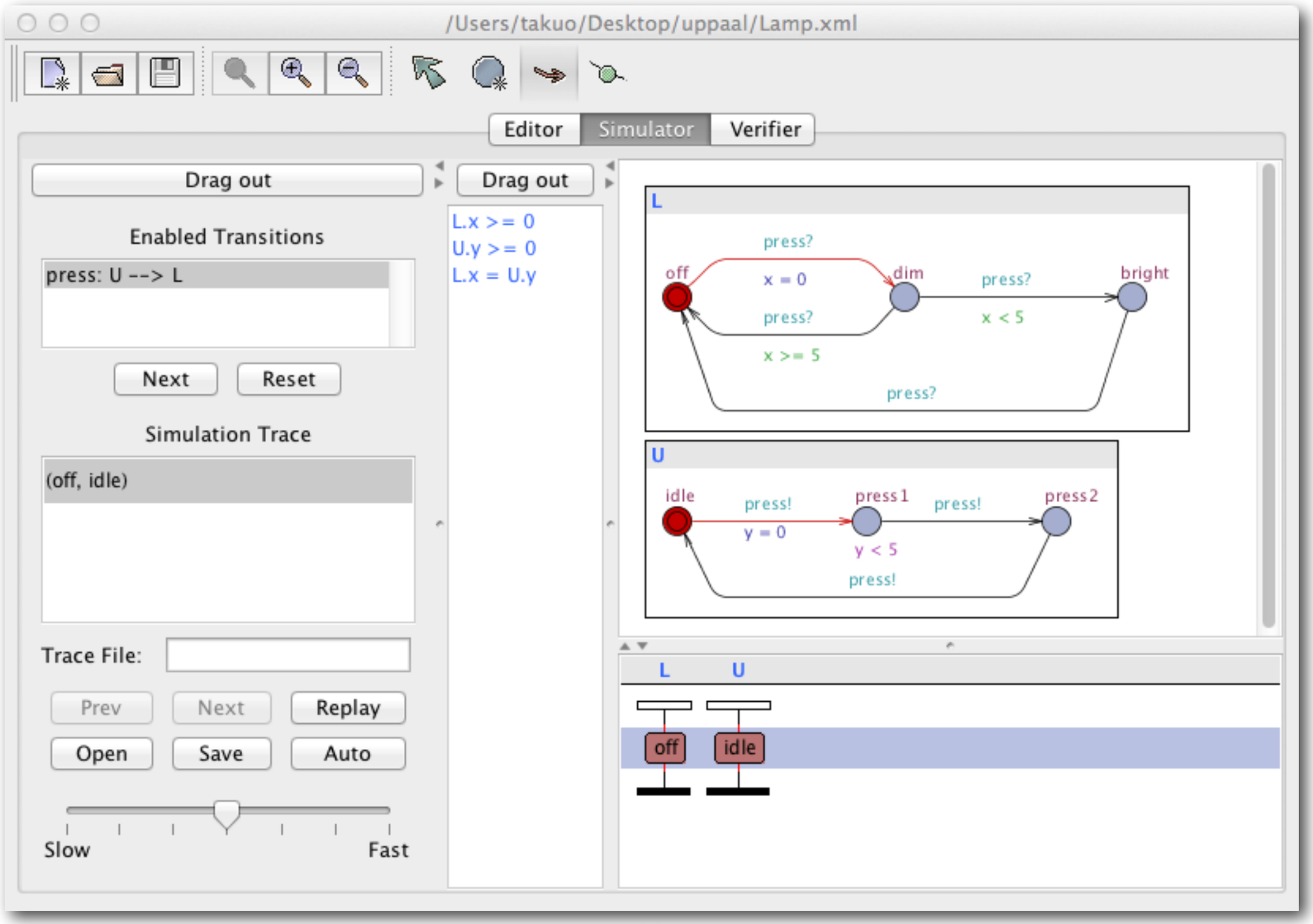
Global & Local Declarations



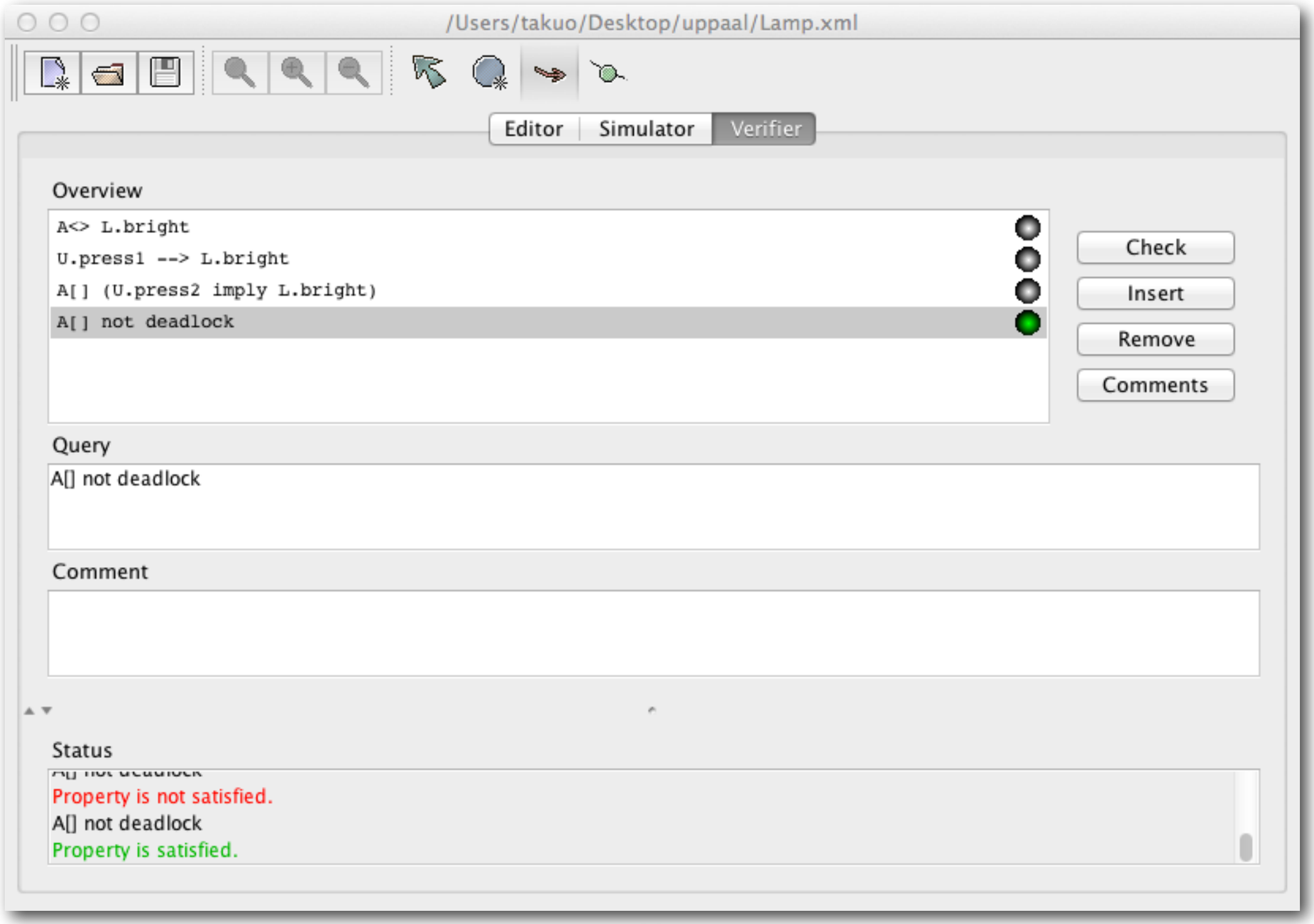
System Declarations



Simulator


/Users/takuo/Desktop/uppaal/Lamp.xml
Editor
Simulator
Verifier
Drag out
Drag out
Enabled Transitions
press: U --> L
Next
Reset
Simulation Trace
(off, idle)
Trace File:
Prev
Next
Replay
Open
Save
Auto
Slow
Fast
L.x >= 0
U.y >= 0
L.x = U.y
L
off
dim
bright
press?
x = 0
press?
x >= 5
press?
x < 5
press?
U
idle
press1
press2
press!
y = 0
press!
y < 5
press!
L
U
off
idle

Verifier (Model Checker)



System Properties in TCTL Formulae

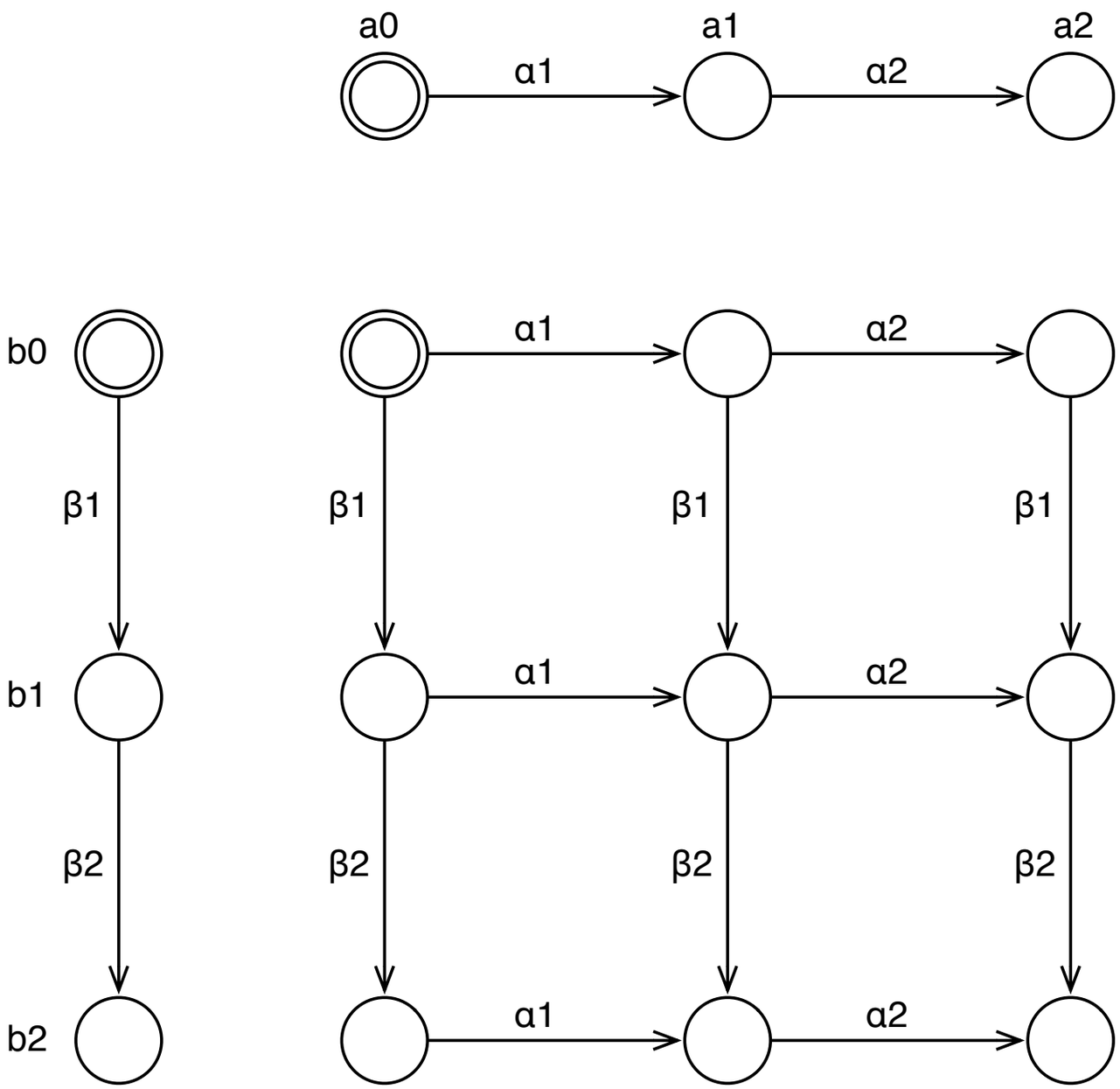
TCTL formula	Lamp II User1	Lamp II User2	Lamp II User3
`A<> L.bright`	×	×	×
`U.press1 --> L.bright`	—	×	○
`A[] (U.press2 imply L.bright)`	—	○	○
`A[] not deadlock`	○	×	○

`p --> q ≡ A[] (p imply A<> q)`

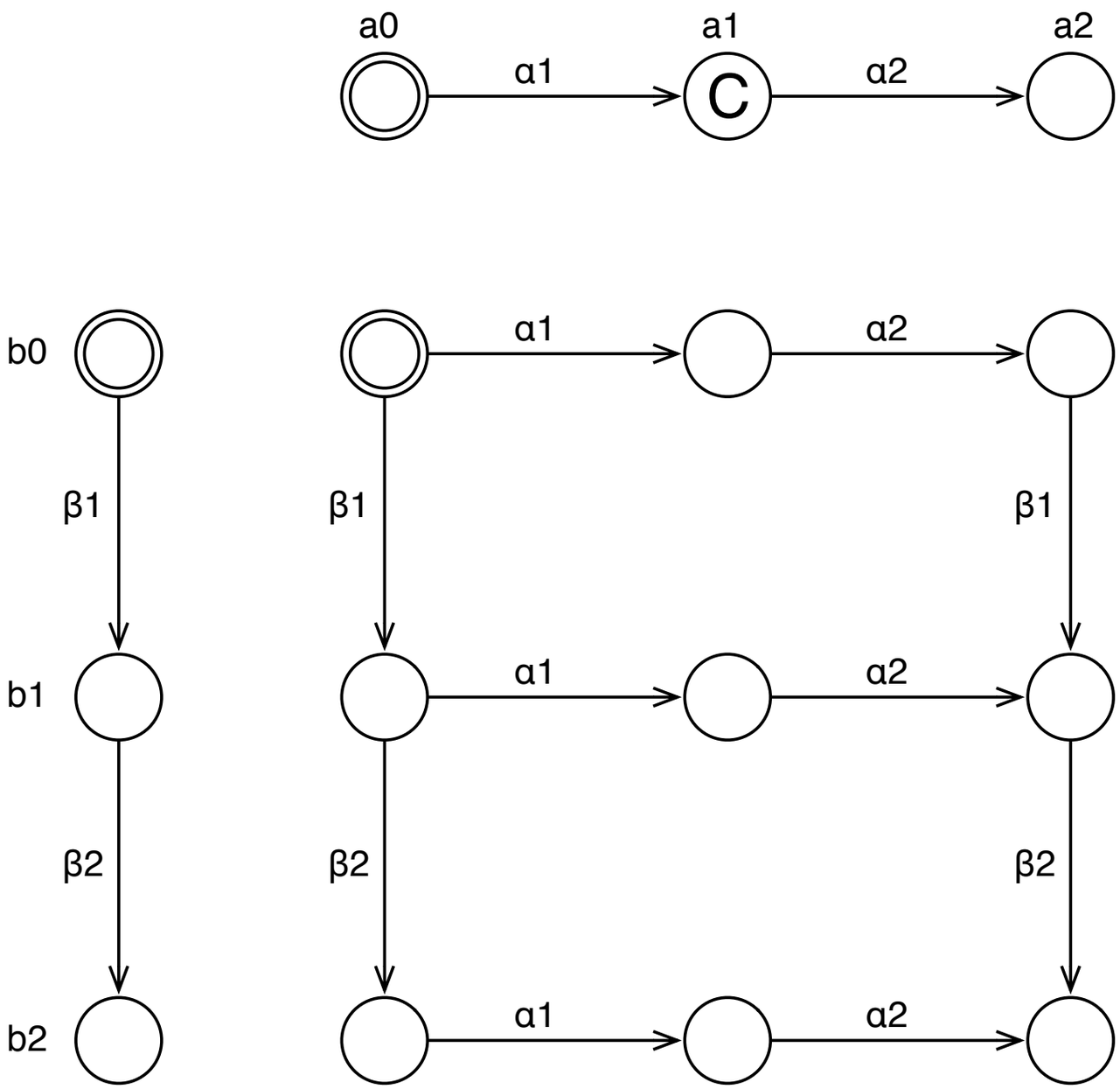
Special Locations

- Committed Location
  - In a state having committed locations active, delay is not allowed and one of the committed location must be left (= one of the outgoing edges from committed locations must be chosen) in the successor states.
- Urgent Location
  - Time may not progress in an urgent location, but interleavings with normal state are allowed.
  - An urgent location is equivalent to a location with incoming edge reseting a designated clock y and labelled with the invariant y <= 0.

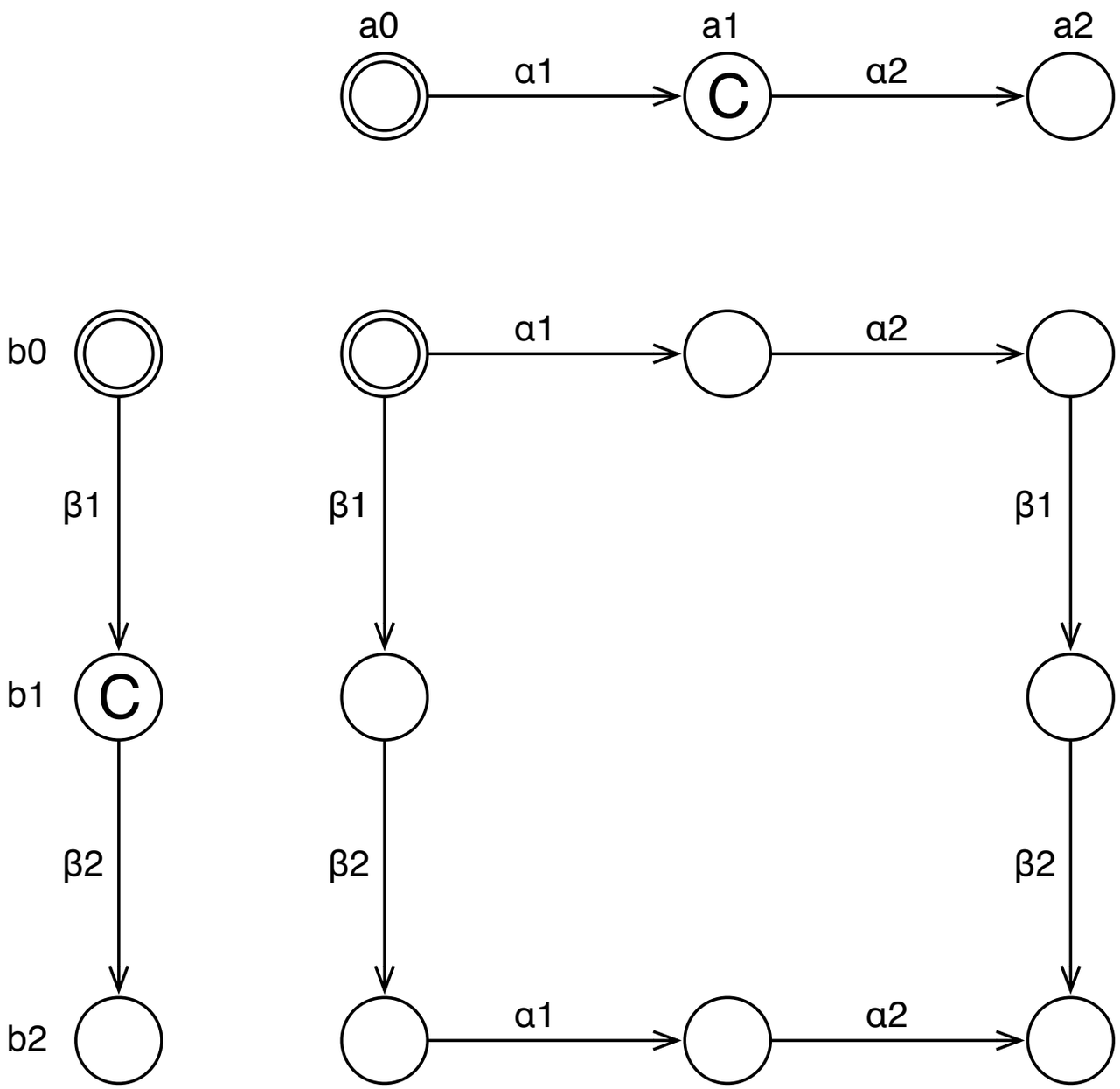
Normal Locations



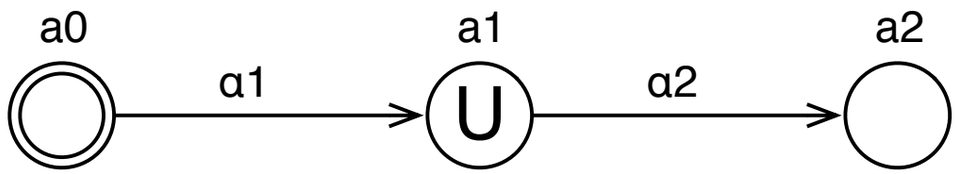
Committed Locations (1)

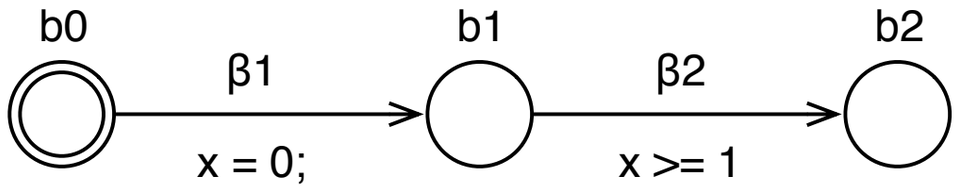


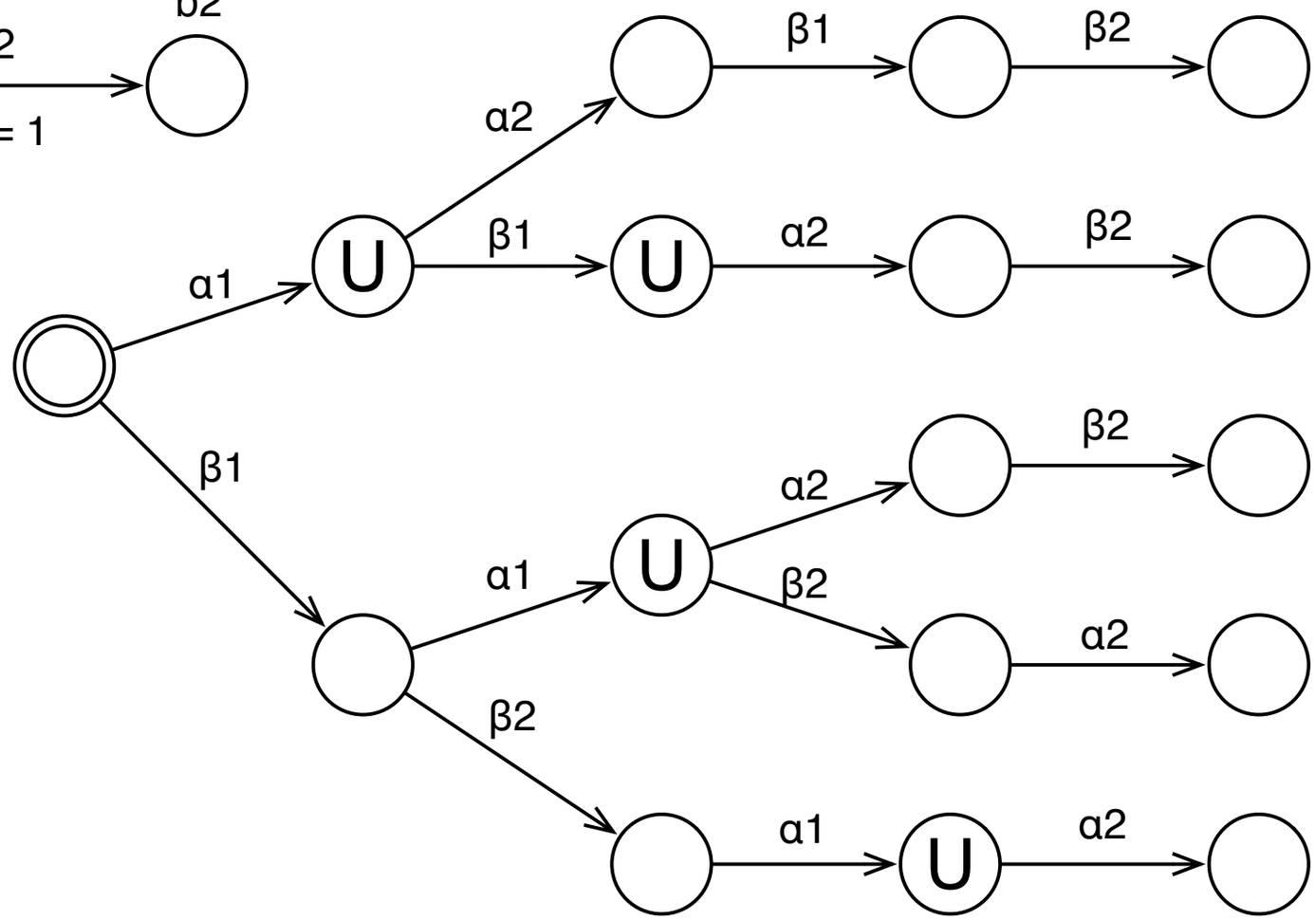
Committed Locations (2)



Urgent Locations







Urgent Channels

- Urgent channels are similar to ordinary channels, except that it is not possible to delay in the source state if it is possible to trigger a synchronization over an urgent channel.
  - Clock guards are not allowed on edges synchronizing over urgent channels.

Modeling/Verifying Real-Time Systems

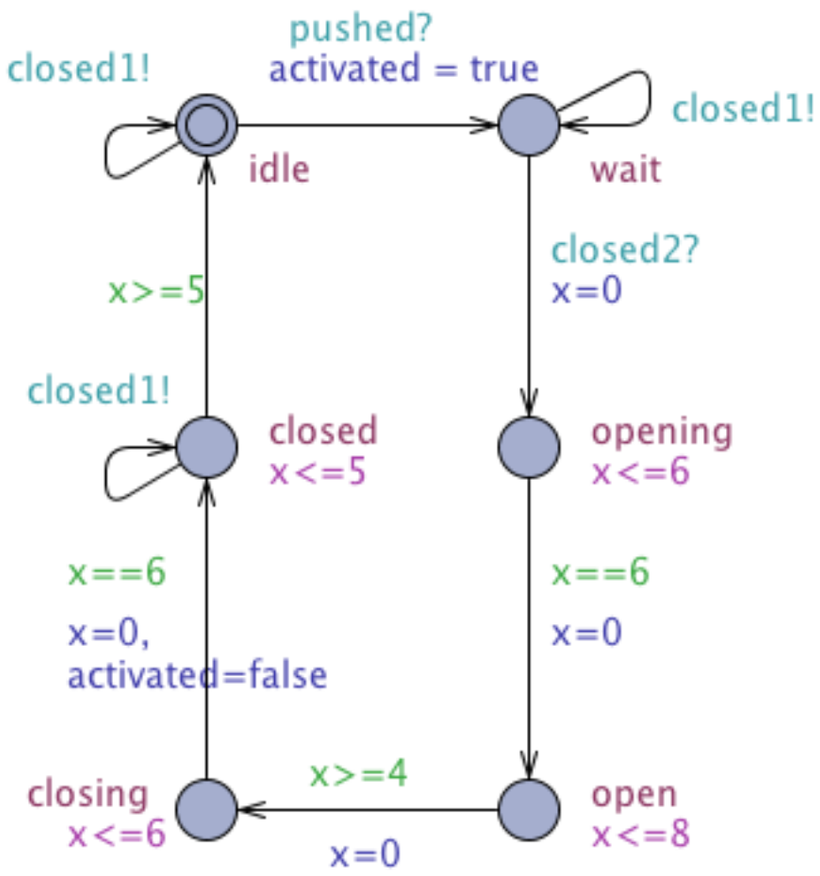
- Two Doors
- Fisher's Mutual Exclusion Algorithm
- Four Vikings
- Train Gate

Two Doors

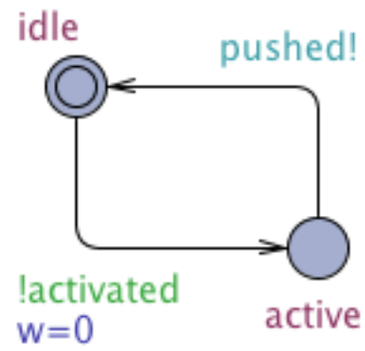
- There is a room that has two doors.
- Two users are standing at different doors.
- Each door has a button to open it.
- It takes 6 time units to open/close a door.
- A door keeps opened for 4 to 8 time units.
- After a door is closed, it takes 5 time units to be able to push the button of the door.
- Each door can be opened only when the other is closed.

Two Doors: Door and User

Door



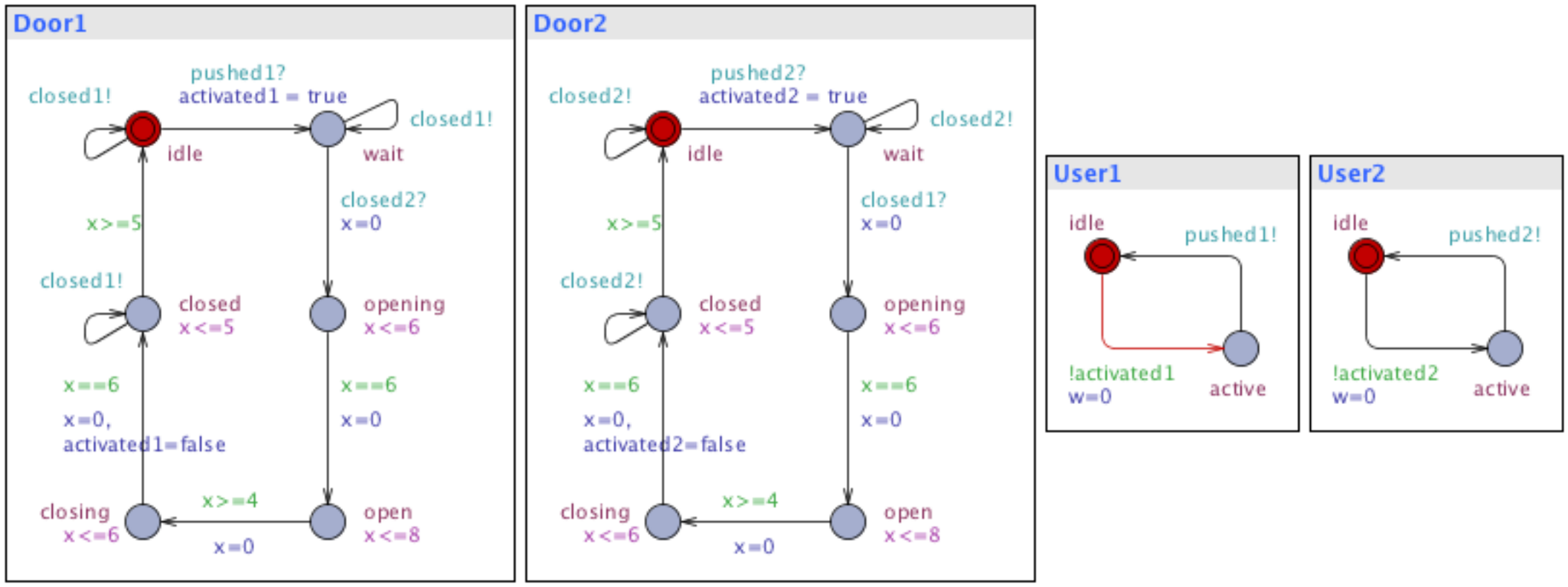
User




```
clock x;
```

```
clock w;
```

Two Doors: System




```
bool activated1, activated2;
urgent chan pushed1, pushed2;
urgent chan closed1, closed2;

Door1 = Door(activated1, pushed1, closed1, closed2);
Door2 = Door(activated2, pushed2, closed2, closed1);
User1 = User(activated1, pushed1);
User2 = User(activated2, pushed2);

system Door1, Door2, User1, User2;
```

Two Doors: Properties

- Mutex
  - `A[] not (Door1.open and Door2.open)`
- Bounded Liveness
  - `A[] (Door1.opening imply User1.w <= 31 and Door2.opening imply User2.w <= 31)`
- Each Door can open
  - `E<> Door1.open`
  - `E<> Door2.open`
- Liveness
  - `Door1.wait --> Door1.open`
  - `Door2.wait --> Door2.open`
- Deadlock Freeness
  - `A[] not deadlock`

Mutual Exclusion Algorithm

- We have two (or more) threads whose behavior can be described as follows.

```
// behavior of i-th thread
void P() {
  while (true) {
    NC          // noncritical section
    EnterCS
    CS          // critical section
    ExitCS
  }
}
```

- Two or more threads must not execute CS at the same time. EnterCS and ExitCS should guarantee such constraint.

Wrong Answer

```
// shared variable
bool in_use = false;  // someone is in CS
```

```
// behavior of i-th thread
void P() {
  while (true) {
    NC
    while (in_use);   // wait until no one is in CS
    in_use = true;    // now I'm in CS
    CS
    in_use = false;   // now I'm out
  }
}
```

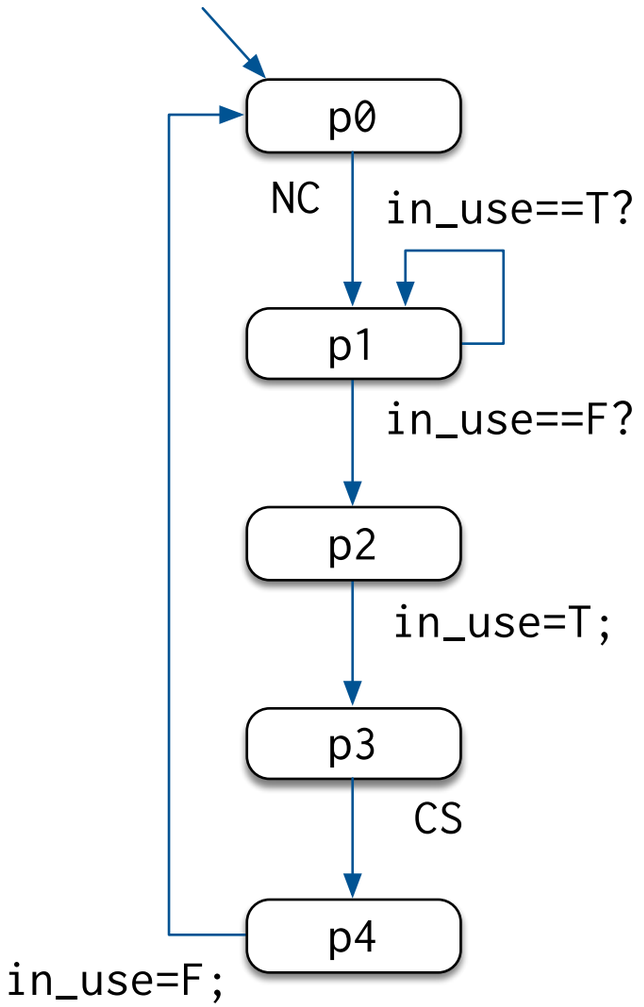
Dekker's Algorithm (for two threads)

```
// shared variables
bool wantp = false, wantq = false;
int turn = 1;  // 1 or 2
```

```
// behavior of thread #1
void P0() {
  while (true) {
    NC
    wantp = true;
    while (wantq) {
      if (turn == 2) {
        wantp = false;
        while (turn == 2);
        wantp = true;
      }
    }
    CS
    turn = 2;
    wantp = false;
  }
}
```

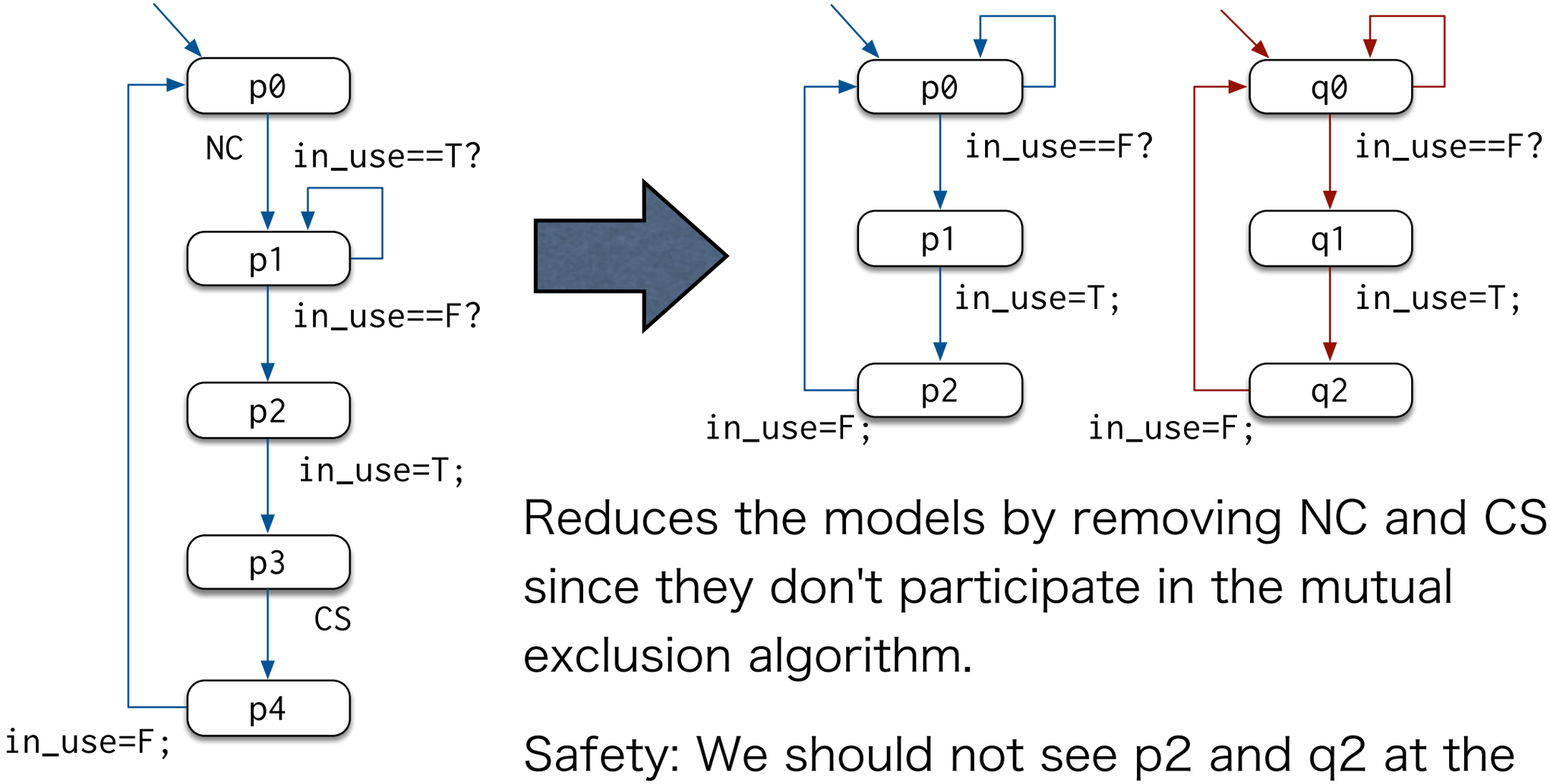
```
// behavior of thread #2
void P1() {
  while (true) {
    NC
    wantp = true;
    while (wantq) {
      if (turn == 1) {
        wantp = false;
        while (turn == 1);
        wantp = true;
      }
    }
    CS
    turn = 1;
    wantp = false;
  }
}
```

Modeling Threads (1)




```
void P() {
    while (true) {
        p0: NC;
        p1: while (in_use);
        p2: in_use = true;
        p3: CS;
        p4: in_use = false;
    }
}
```

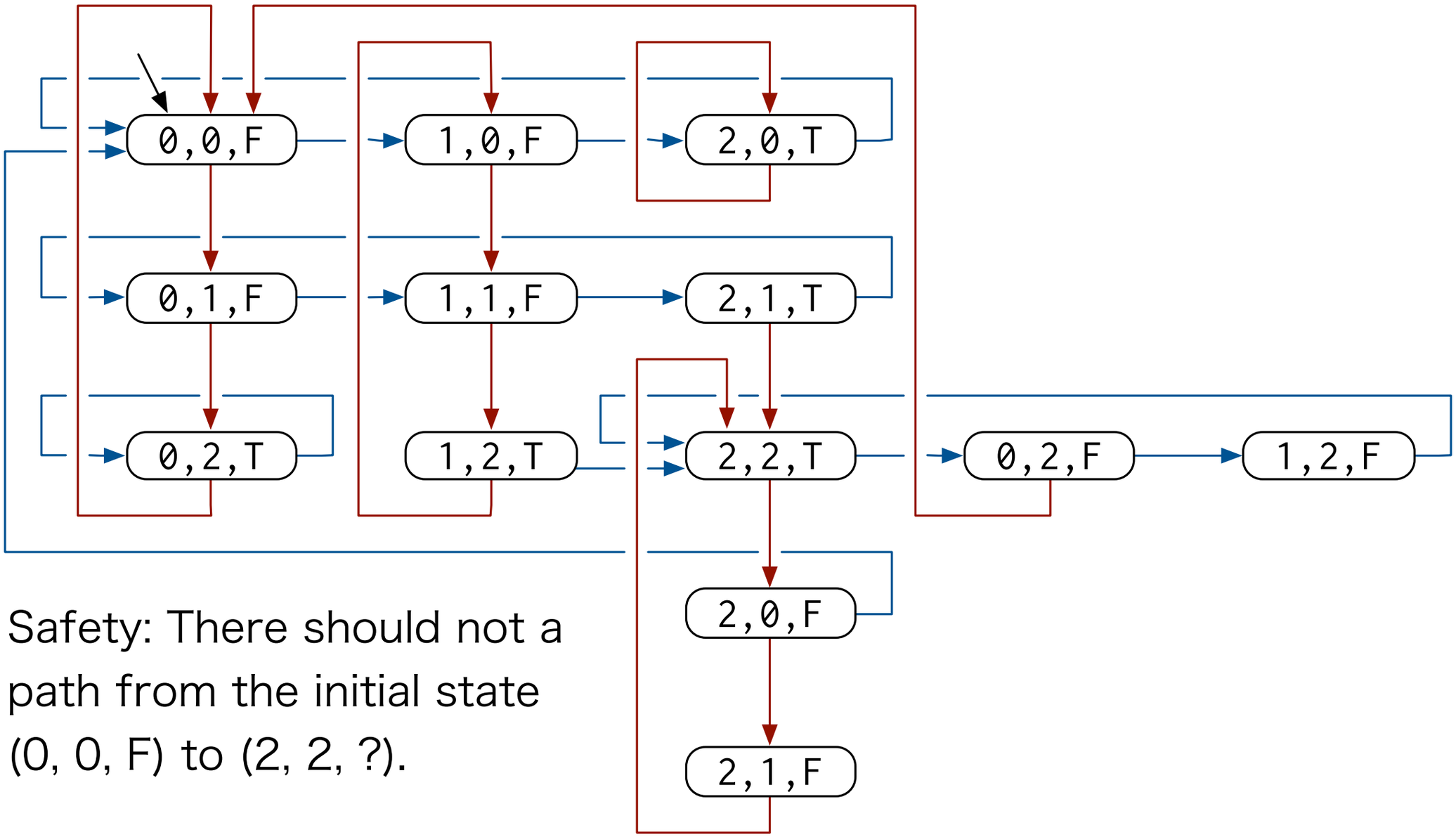
Modeling Threads (2)



Reduces the models by removing NC and CS since they don't participate in the mutual exclusion algorithm.

Safety: We should not see p2 and q2 at the same time.

Composing Models



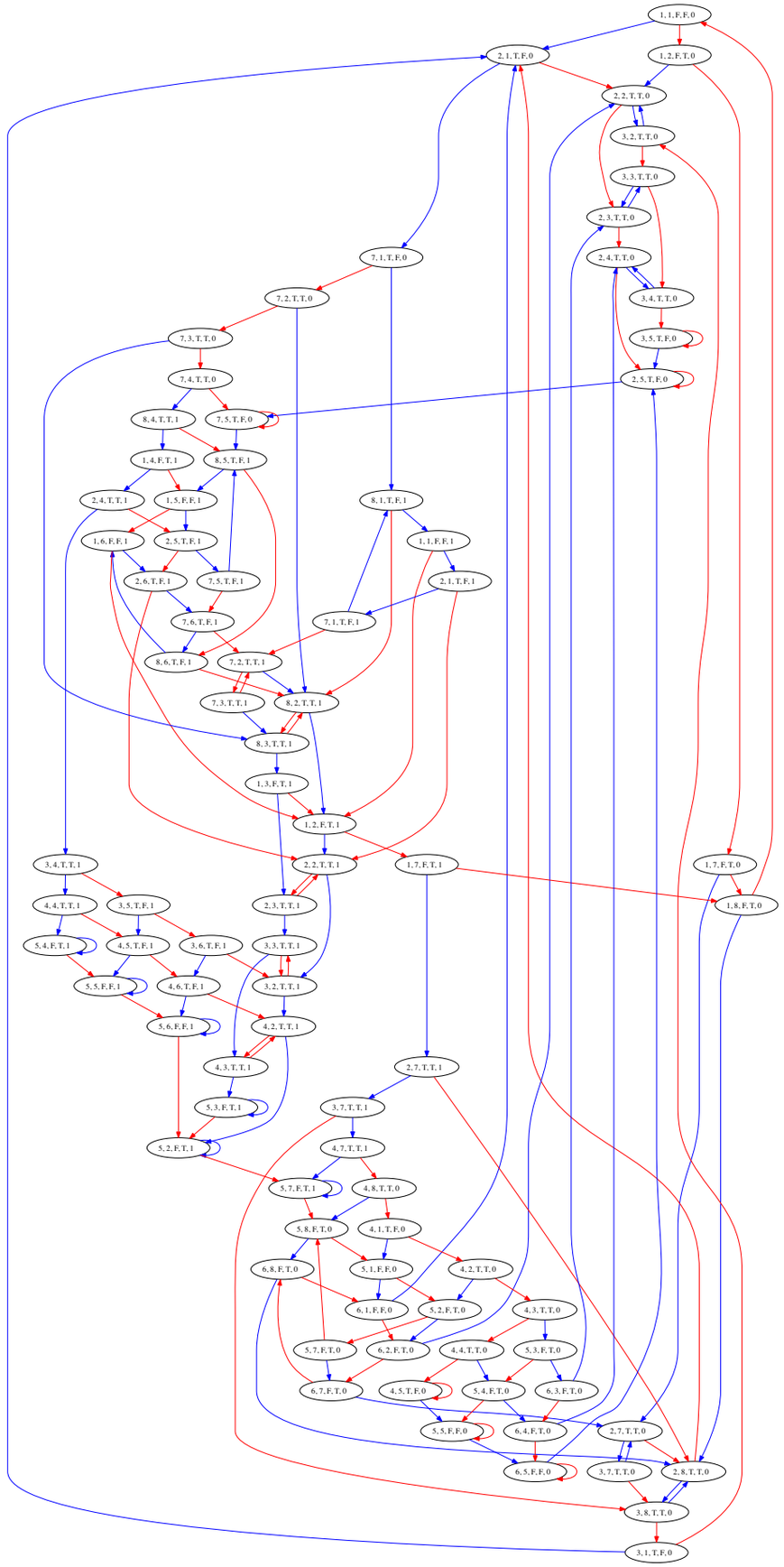
Safety: There should not a path from the initial state (0, 0, F) to (2, 2, ?).

Model Checking Dekker's Algorithm

```
mtype = { TurnP, TurnQ };
mtype turn = TurnP;
bool wantp = false, wantq = false;
int ncs = 0;

active proctype P () {
  do :: wantp = true;
        do :: wantq ->
              if :: (turn == TurnQ) ->
                    wantp = false;
                    turn == TurnP;
                    wantp = true;
                 :: else -> skip;
              fi
           :: else -> break;
        od;
progress:
        ncs++;
        assert(ncs == 1);
        ncs--;
        turn = TurnQ;
        wantp = false;
  od
}
```

Dekker's Algorithm in Promela



Using Atomic Instructions

```
// shared variable
bool in_use = false;
```

```
// behavior of i-th thread
void P() {
  while (true) {
    NC
    while (XCHG(&in_use, true));
    CS
    in_use = false;
  }
}
```

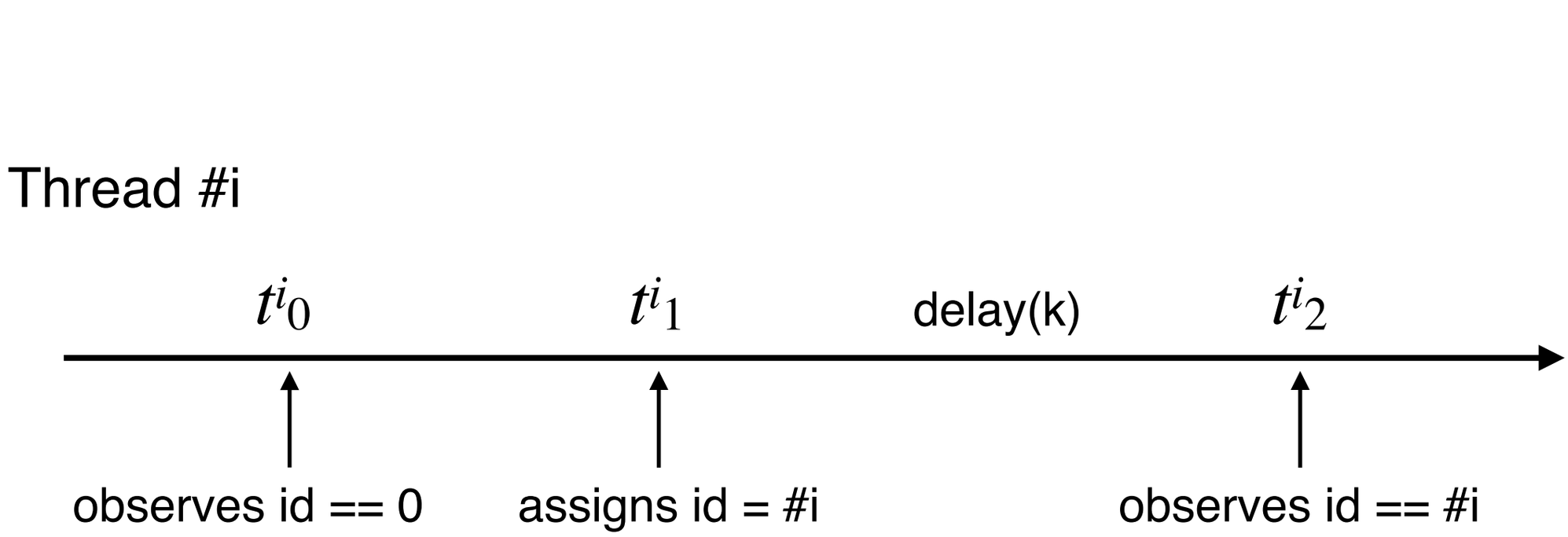
```
atomic bool XCHG(bool *var, bool new) {
  bool old = *var;
  *var = new;
  return old;
}
```

Fischer's Mutual Exclusion Algorithm

```
// shared variable
int id = 0;
```

```
// behavior of i-th thread
const int k = ...:                    // delay time
void P() {
  while (true) {
    NC                                // noncritical section
    do {
      while (id != 0);
      id = pid;                       // pid != 0
      delay(k);
    } while (id != pid);
    CS                                // critical section
    id = 0;
  }
}
```

Fischer's Mutual Exclusion Algorithm

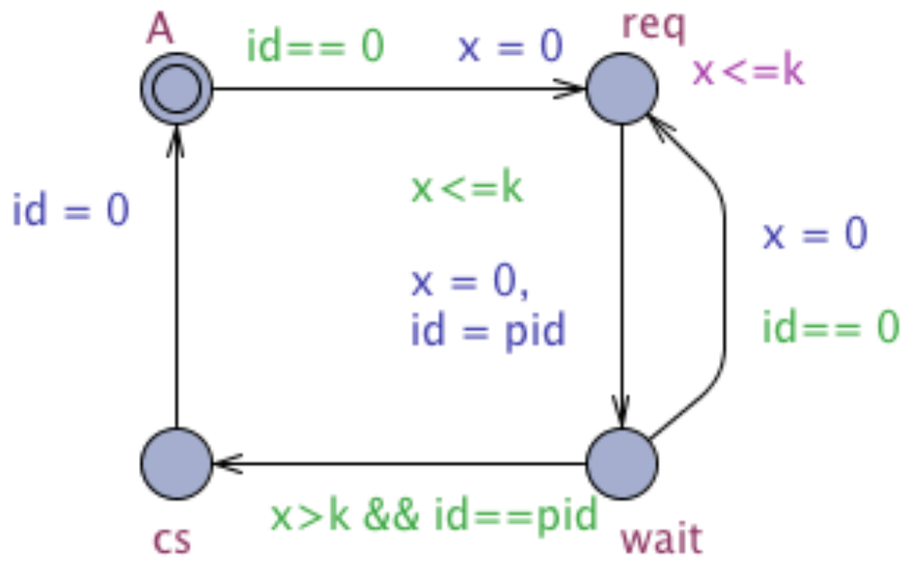


- Assumption: for each thread #i
  - $t^i_1 - t^i_2 \leq \Delta$
  - $t^i_2 - t^i_1 > \Delta$

- If k is larger than or equal to Δ, the algorithm satisfies the mutual exclusiveness and deadlock-freeness.

Fischer's Mutual Exclusion Algorithm

Process: P




```
clock x;
const int k = 2;
```

System Declarations

```
typedef int[1,6] id_t;
int id;

system P;
```

Fischer's Algorithm: Properties

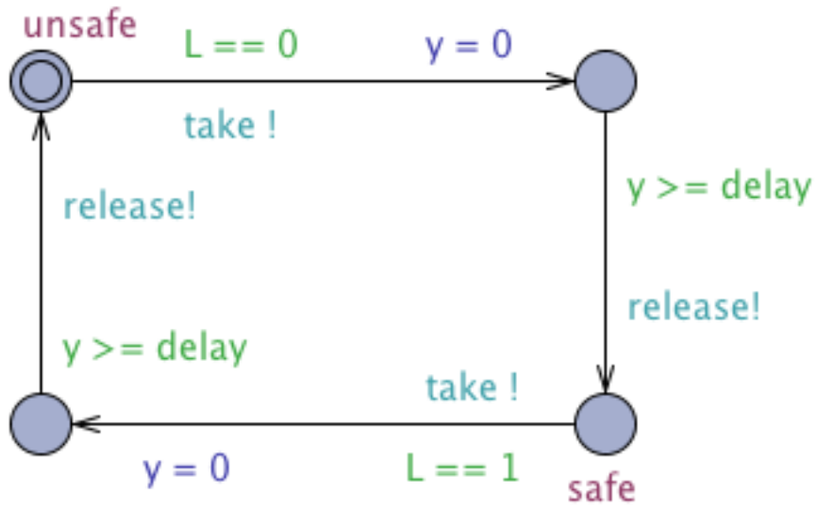
- Mutex
  - `A[] forall (i:id_t) forall (j:id_t) P(i).cs && P(j).cs imply i == j`
- Deadlock Freeness
  - `A[] not deadlock`
- Whenever a process requests access to the CS, it eventually enter the wait state
  - `P(1).req --> P(1).wait`

Four Vikings

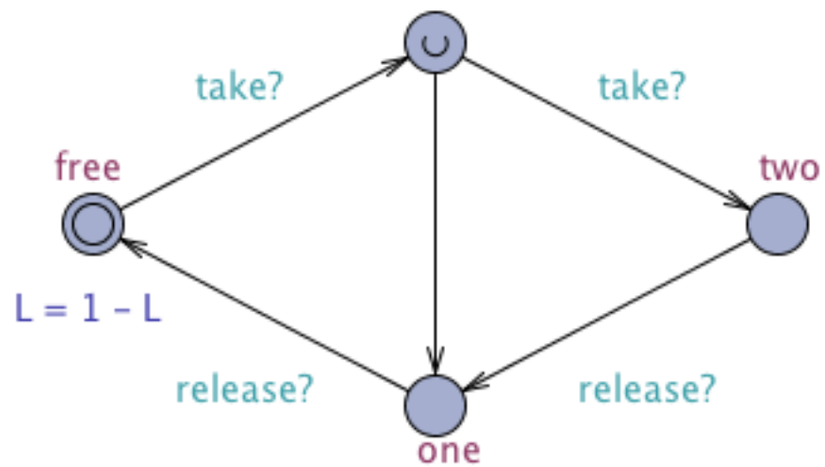
- Four vikings are about to cross a damaged bridge in the middle of the night. The bridge can only carry two of the vikings at the time and to find way over the bridge the vikings need to bring a torch. The vikings need 5, 10, 20 and 25 minutes (one-way) respectively to cross the bridge.
- Does a schedule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

Four Vikings: Definitions

Viking



Torch



Declarations

```
chan take, release;
int[0,1] L;
clock time;
```

```
clock y;
```

System Declarations

```
Viking1 = Viking(5);
Viking2 = Viking(10);
Viking3 = Viking(20);
Viking4 = Viking(25);
system Viking1, Viking2, Viking3, Viking4, Torch;
```

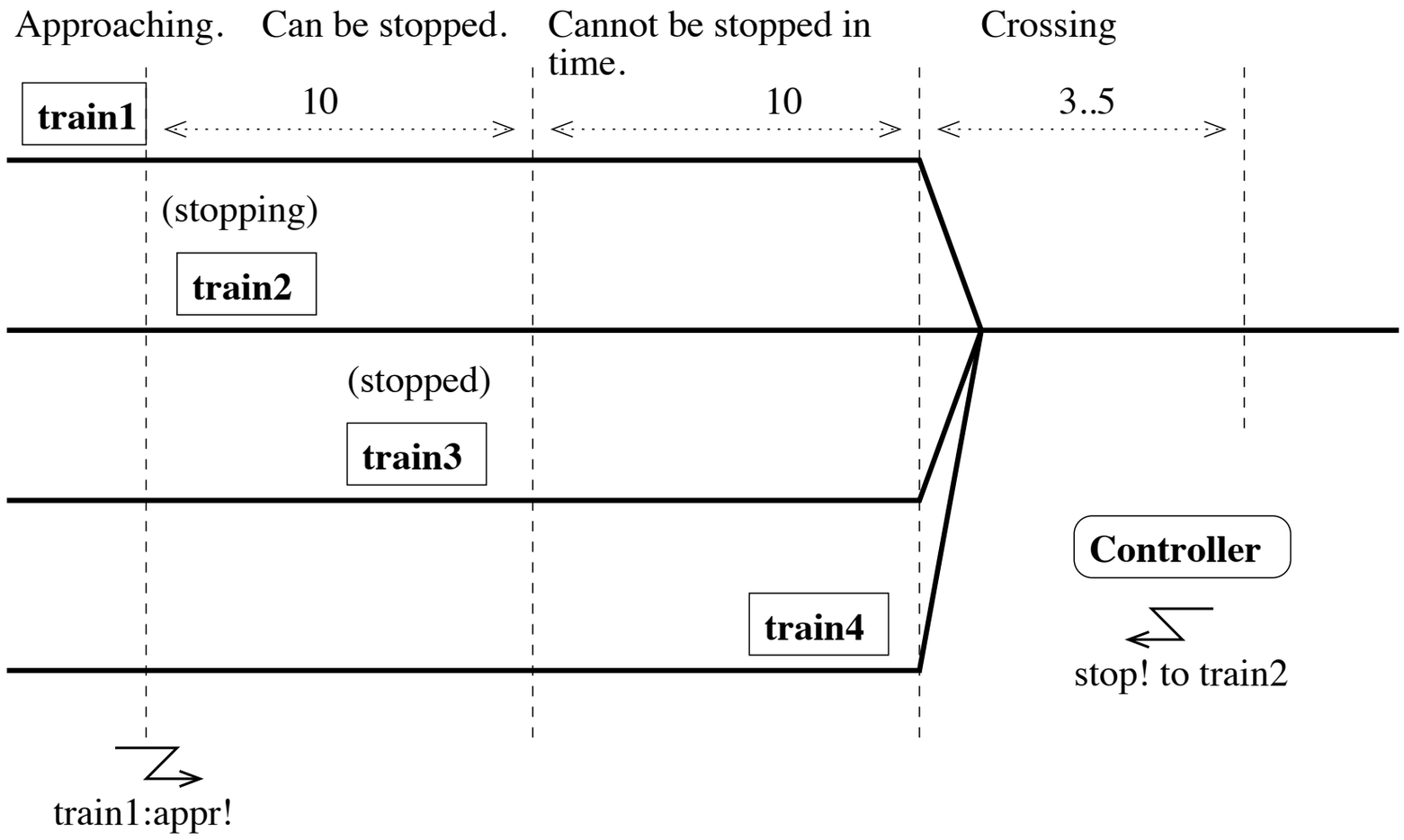
Four Vikings: Properties

- Deadlock Freeness
  - `A[] not deadlock`
- Each viking can cross the bridge
  - `E<> Vikingk.safe`
- Slowest viking needs 25 minutes
  - `A[] not (Viking4.safe and time < 25)`
  - `E<> Viking4.safe imply time >= 25`
- All vikings can cross the bridge within 60 min.
  - `E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe and time <= 60`

Train Gate

- A railway control system which controls access to a bridge for several trains.
  - The bridge is a critical shared resource that may be accessed only by one train at a time.
  - The system is defined as a number of trains (assume 4 for this example) and a controller.
  - A train can not be stopped instantly and restarting also takes time. Therefore, there are timing constraints on the trains before entering the bridge.

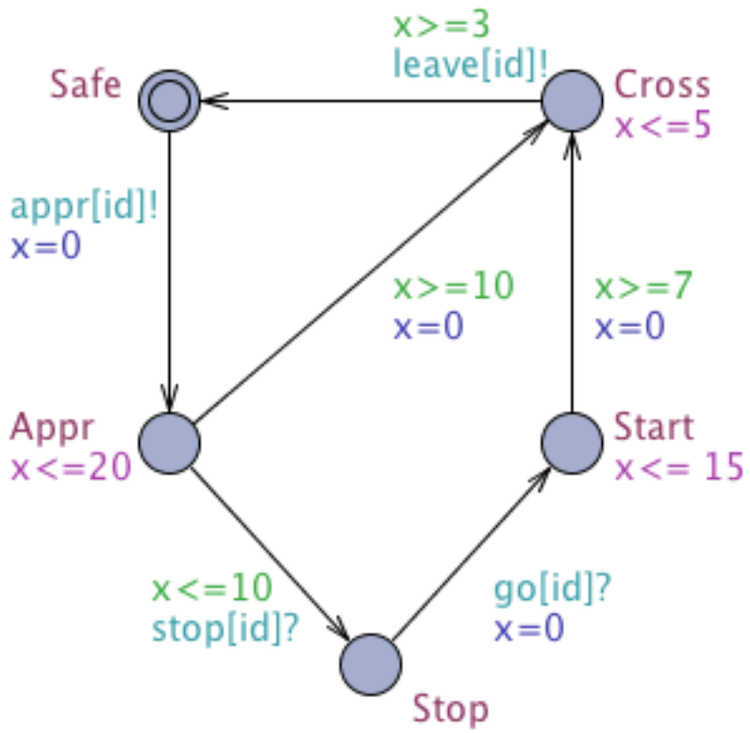
Train Gate: Example Scenario



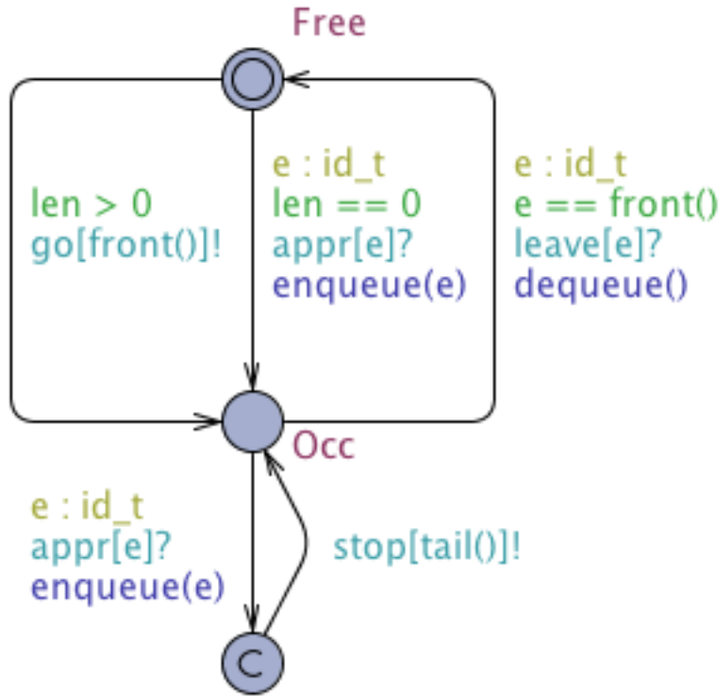
Train4 is about to cross the bridge, train3 is stopped, train2 was ordered to stop and is stopping. Train1 is approaching and sends an appr! signal to the controller that sends back a stop! signal. The different sections have timing constraints (10, 10, between 3 and 5).

Train Gate: Train and Gate

Train



Gate




```
clock x;
```

Train Gate: Local Declarations of Gate

```
id_t list[N + 1];
int[0,N] len;

void enqueue(id_t element) {
    list[len++] = element;
}

void dequeue() {
    int i = 0;
    len -= 1;
    while (i < len) {
        list[i] = list[i + 1];
        i++;
    }
    list[i] = 0;
}

id_t front() { retuen list[0]; }
id_t tail() { return list[len - 1]; }
```

Train Gate: System

Declarations

```
const int N = 6;
typedef int[0,N-1] id_t;

chan appr[N], stop[N], leave[N];
urgent chan go[N];
```

System Declarations

```
system Train, Gate;
```

Train Gate: Properties

- Gate can receive messages
  - `E<> Gate.Occ`
- Each train can reach crossing
  - `E<> Train(k).Cross`
- Train k can cross the bridge while Train l is waiting (k != l)
  - `E<> Train(k).Cross and Train(l).stop`
- Train k can cross the bridge while other trains are waiting
  - `E<> Train(k).Cross and (forall (i:id_t) i != k imply Train(i).Stop)`

Summary

- Modeling Real-Time Systems
  - Timed Automata
    - Clock
    - Guards, Location Invariants
    - Network of Timed Automata
    - TCTL
  - UPPAAL
    - Model Checking Timed Automata
    - Editor / Simulator / Verifier
    - Urgent Locations, Committed Locations
    - Urgent Channels
    - Examples