

# Homework 2 is out

- Due on Apr. 9<sup>th</sup> 11:59pm
- Two parts
  - Risk Management
  - UPPAAL

# Lecture 10: UPPAAL Tutorial

Partially referenced Prof. Insup Lee's course at UPenn

# UPPAAL??!!

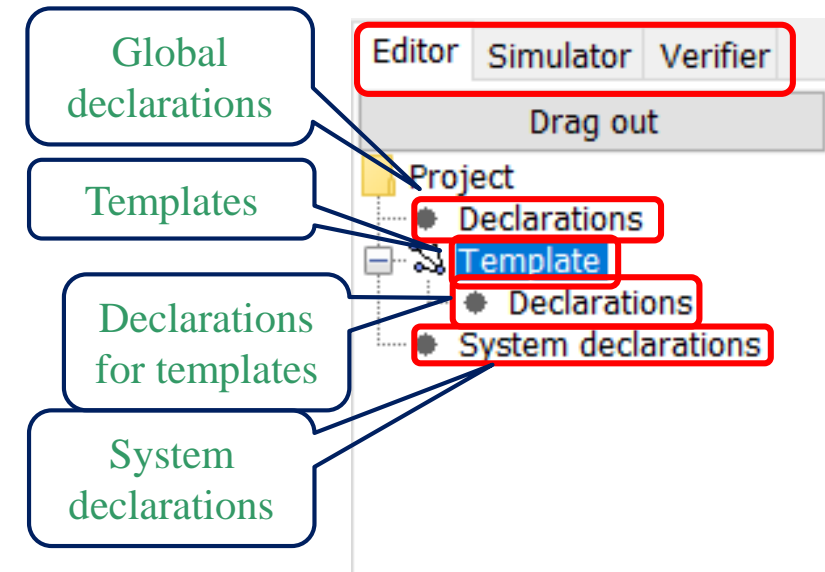
- Model checking tool for Timed-automata
- Developed by Uppsala University and Aalborg University
- **SWE**den + **DEN**mark = SWEDEN
  - REJECTED
- swe**DEN** + den**MARK** = DENMARK
  - REJECTED
- **UPP**sala + **AAL**borg = UPPAAL
  - ACCEPTED

# UPPAAL Tool Parts

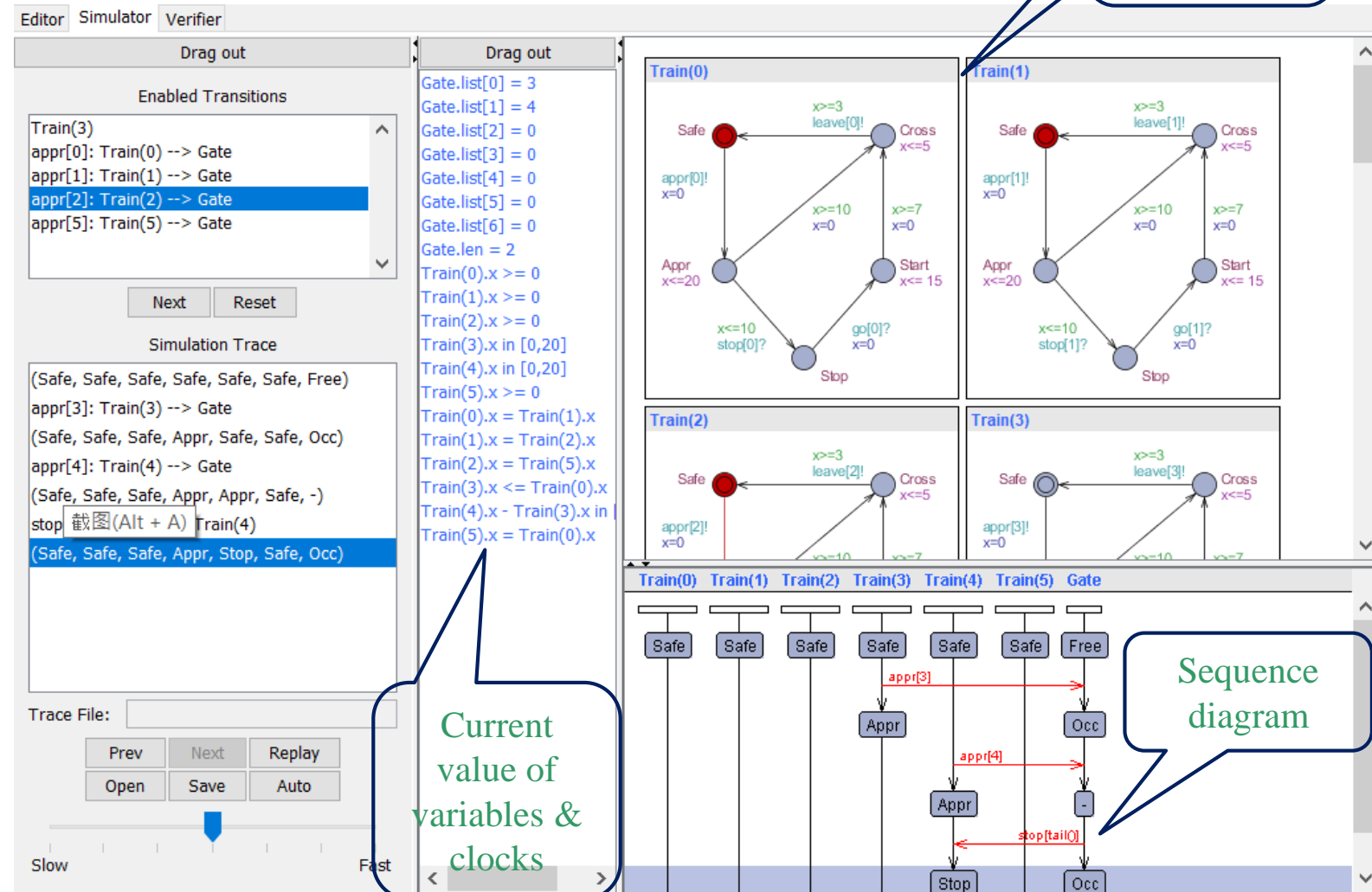
- Graphical user interface (GUI)
  - Used for modeling, simulation, and verification. Uses the verification server for simulation and verification.
- Verification server
  - Used for simulation and verification. In simulation, it is used to compute successor states.
- A command line tool
  - A stand-alone verifier, appropriate for e.g. batch verifications.

# UPPAAL GUI: Editor

- Global declarations
  - Accessible to all system processes
- Templates
  - Can be parameterized
- Declarations for templates
  - Only accessible locally
- System declarations
  - Declare processes and system composition

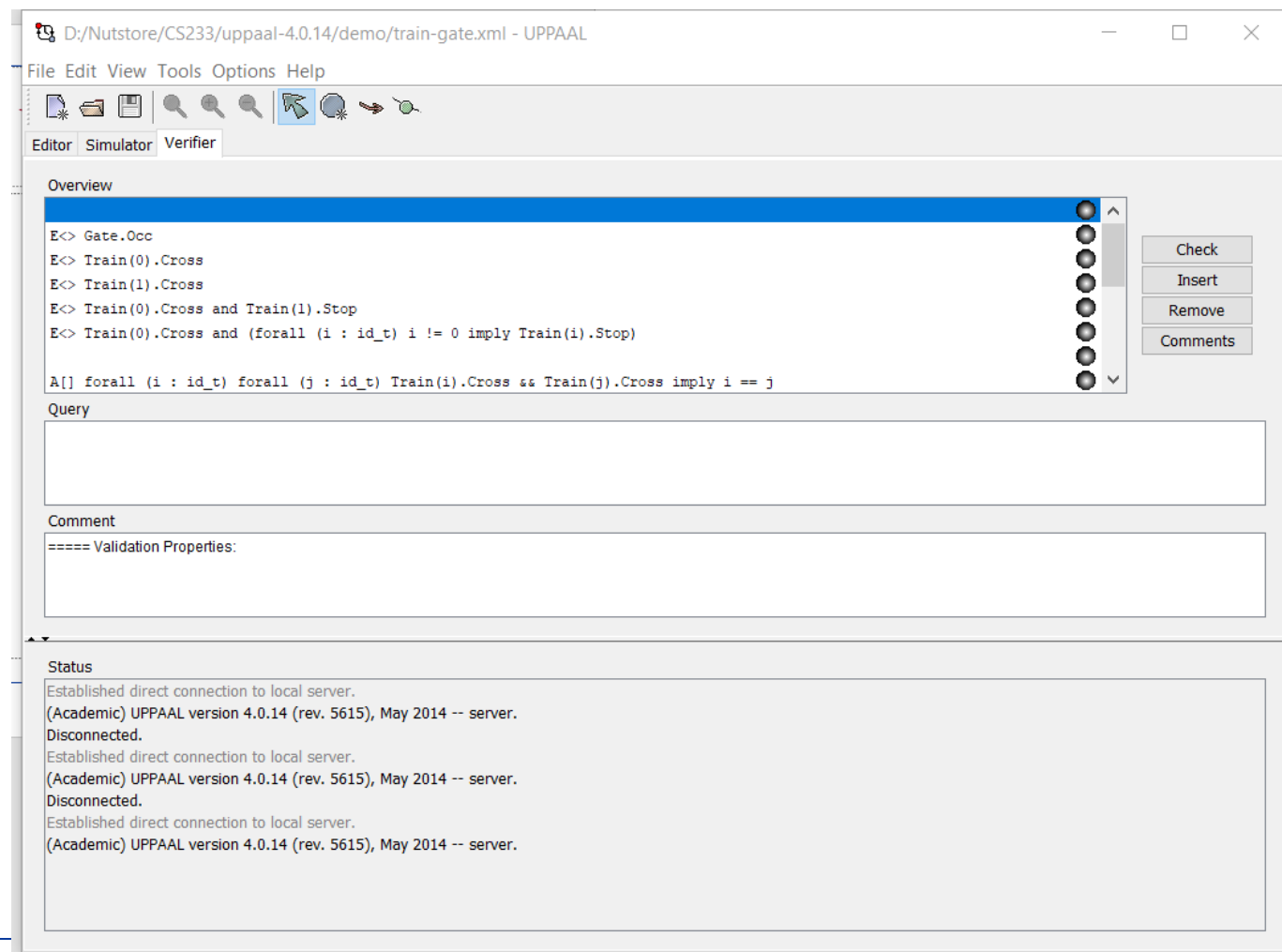
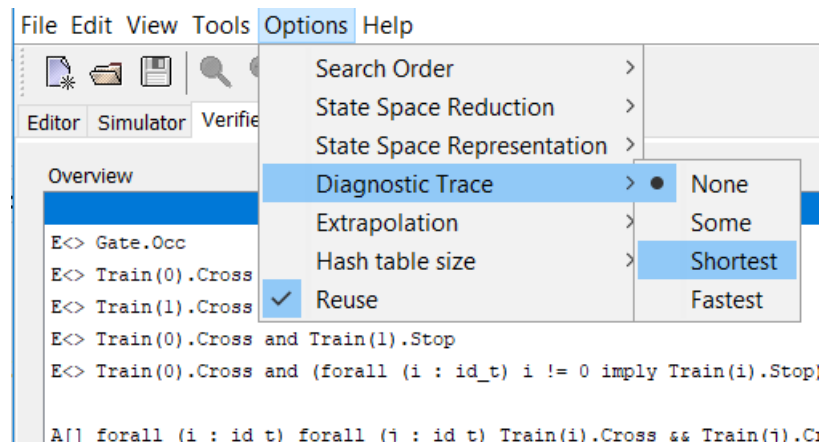


# UPPAAL GUI: Simulator



# UPPAAL GUI: Verifier

- Turn on diagnostic trace to view counter-examples in the simulator



# UPPAAL Syntax

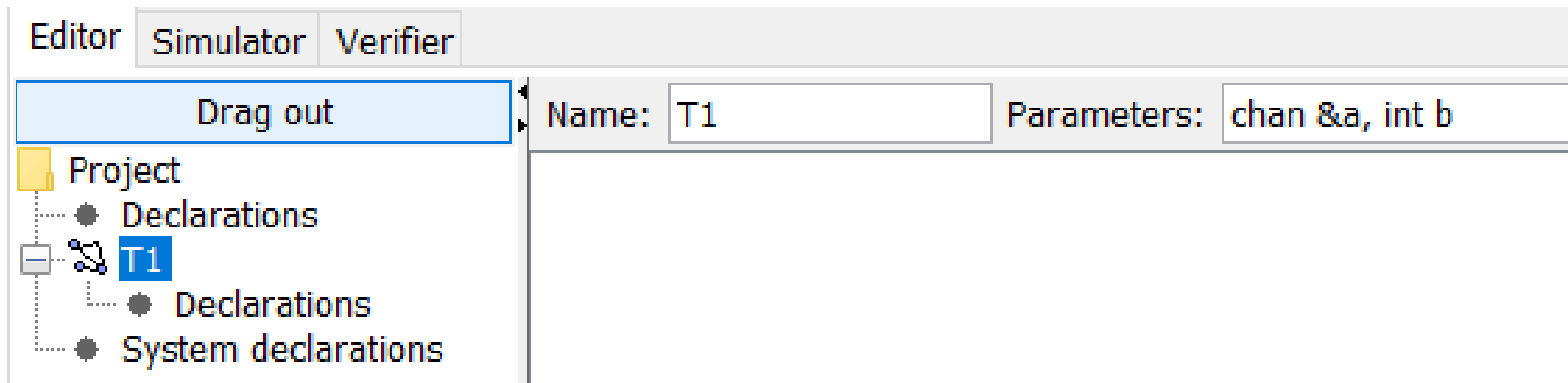
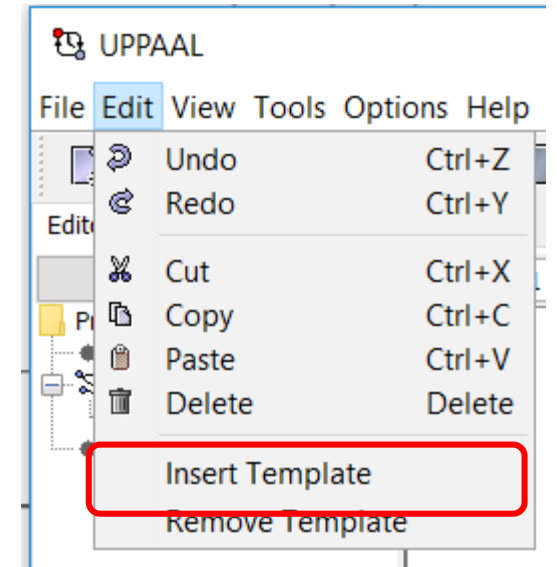


# UPPAAL Syntax

- Global declarations
  - Clocks:
    - clock  $x_1, \dots, x_n$ ;
  - Data variables
    - int  $n_1, \dots$ ; integer with default domain
    - Int[ $l, u$ ]  $n_1, \dots$ ; integer with domain defined by  $[l, u]$
    - Int  $n_1[m], \dots$ ; array with elements  $n_1[0]$  to  $n_1[m-1]$
  - Channels
    - Chan  $a, \dots$ ;
    - Urgent chan  $b \dots$ ;
    - Broadcast chan  $c \dots$ ;
  - Constants
    - Const int  $c_1 = n_1$ ;

# UPPAAL Syntax (cont.)

- Template
  - Names should be unique
  - Just like classes in UML, can be instantiated in system declaration
  - Channel declaration has a “&” in front



# UPPAAL Syntax (cont.)

- System declaration

The screenshot displays the UPPAAL IDE interface. At the top, there are tabs for 'Editor', 'Simulator', and 'Verifier'. Below these, a 'Drag out' button is visible. To the left, a project tree shows a 'Project' folder containing 'Declarations', 'T1' (highlighted), 'Declarations', and 'System declarations'. To the right, the 'Name' field is set to 'T1' and the 'Parameters' field is set to 'chan &a, int b'.




Below this, another 'Drag out' button is shown. To its left, the project tree shows 'Project' containing 'Declarations', 'T1', 'Declarations', and 'System declarations' (highlighted). To the right, the code editor shows the following UPPAAL code:

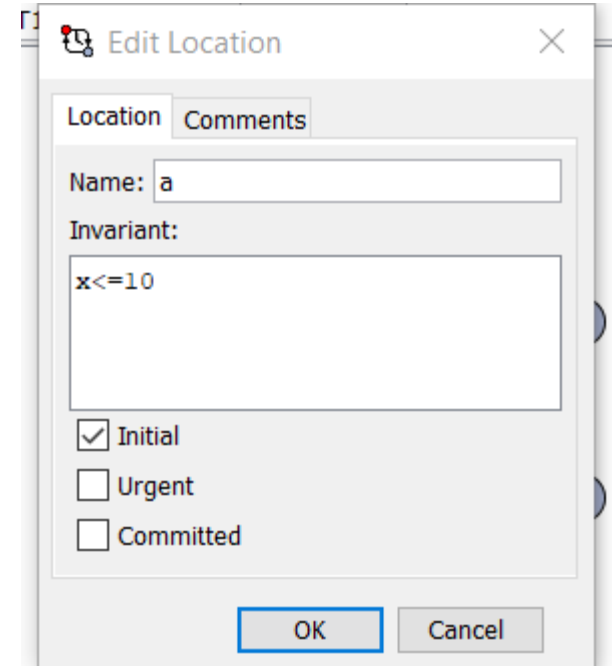
```
// Can declare additional global stuff
chan a;
// Place template instantiations here.
P1 = T1(a,10);
P2 = T1(a,20);

// List one or more processes to be composed into a system.
system P1,P2;
```

A callout box points to the instantiation lines, stating: "Can instantiate multiple processes from the same template".

# UPPAAL Syntax: Locations

- Initial State 
  - Only one per template
- Urgent State 
- Committed State 
- Invariant
  - Conditions that need to be satisfied when in state a



Dialog box titled "Edit Location" showing fields for Name (a), Invariant (x <= 10), and checkboxes for Initial (checked), Urgent, and Committed. Buttons for OK and Cancel are at the bottom.

# UPPAAL Syntax: Edges

- Select
  - Defines multiple parameterized transitions
- Guard
  - Condition under which the edge is enabled
- Sync
  - `a!` for sending message
  - `a?` for receiving message
- Update
  - Actions taken during the transition

The screenshot shows the 'Edit Edge' dialog box in UPPAAL. It has two tabs: 'Edge' and 'Comments'. The 'Edge' tab is selected, displaying four input fields for defining a transition:

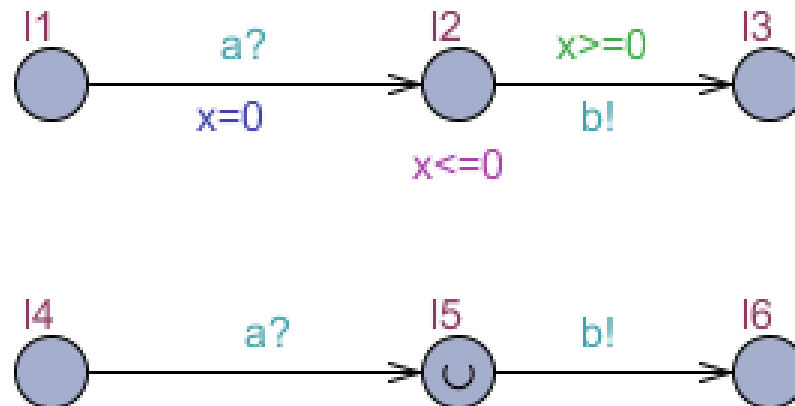
- Select:** `e : int[1,5]`
- Guard:** `x>5`
- Sync:** `a[e]?`
- Update:** `b=e`

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

# UPPAAL Semantics

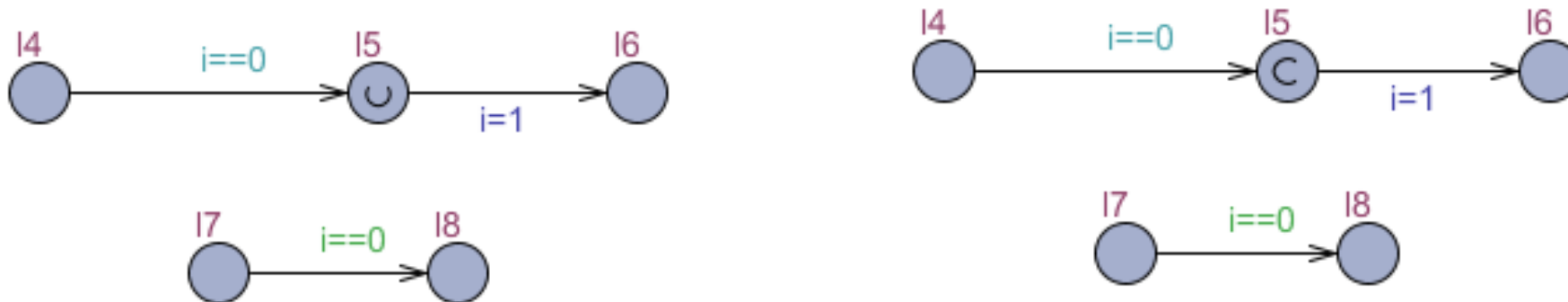
# Urgent Location

- No time pass in an urgent location
- The two automata is equivalent
- Save a clock thus reduce state space



# Committed Location

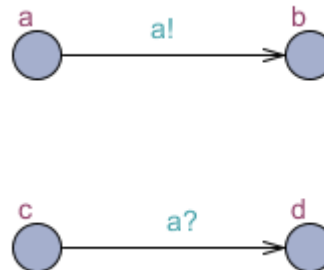
- Urgent location still allows interleaving
  - 17 -> 18 can happen before 15->16, although no time has passed
- Committed states are stronger than urgent locations
- If multiple committed states reached at the same time, the transitions will interleave
- Reduce interleaving thus reduce complexity





# Urgent channels

- Urgent channel definition
  - Urgent chan a;
- a! sent as soon as location a and c are reached
- No clock guard is allowed on the transitions with urgent channel components

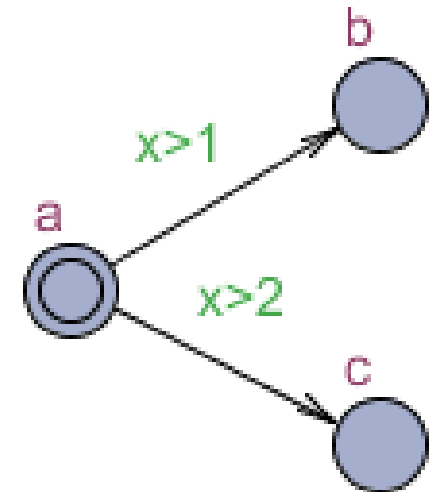


# Broadcast channels

- Synchronize multiple processes
- If receiving channel  $a?$  is enabled, the transition must be taken
- Can send without any receivers ready

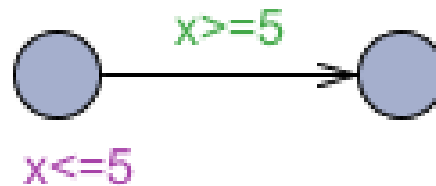
# Non-determinism

- Transitions with guard evaluates true are only enabled
- Used to model variabilities of system environment
- Location a does not have an invariant, thus can stay forever
  - $x \in [0,1]$ : location a
  - $x \in (1,2]$ : location a or b
  - $x \in (2, \infty]$ : location a or b or c



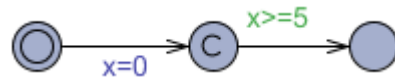
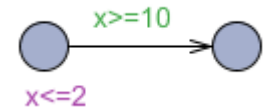
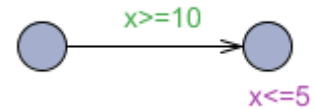
# Want to do something at a particular time?

- The transition will take place when  $x==5$



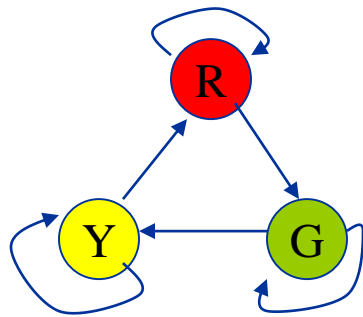
# Deadlock

- No enabled transitions
- Common deadlock scenarios
  - Cannot enter a state
  - State invariant about to expire and no enabled transition available
  - Committed state does not have enabled outgoing transition

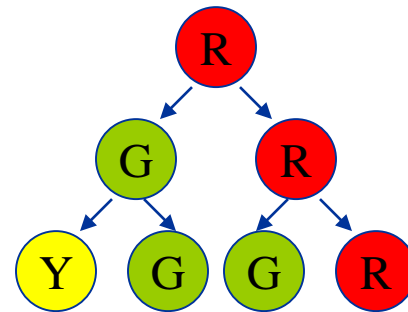


# Computational Tree Logic (CTL)

- CTL is a logic used to express properties for model checking
- CTL is useful because there is an efficient technique to check it
- A **temporal logic** is a logic which can express aspects of time
- CTL makes statements about the **computational tree** of a state machine



Traffic light FSM

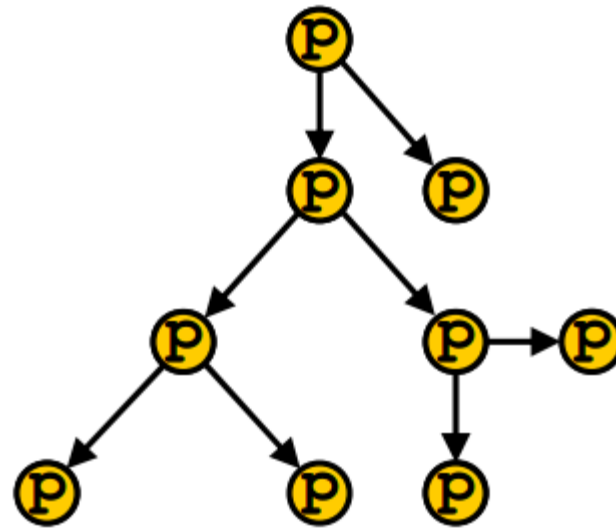


Computational tree for FSM

# Temporal Computational Tree Logic (TCTL)

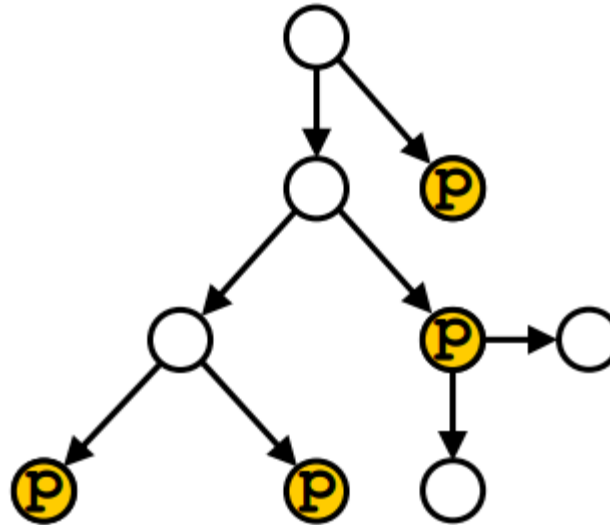
## Properties

- $A[] p$  “Always globally  $p$ ”
- For each (all) execution path  $p$  holds for all the states of the path



# Properties

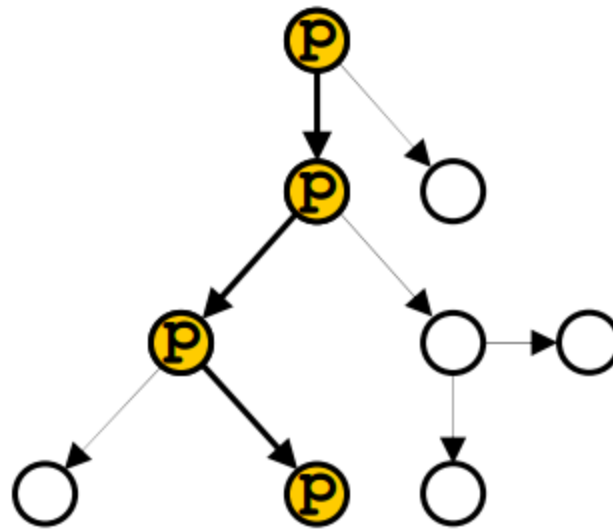
- $A \leftrightarrow p$  “Always eventually  $p$ ”
- For each (all) execution path  $p$  holds for at least one state of the path





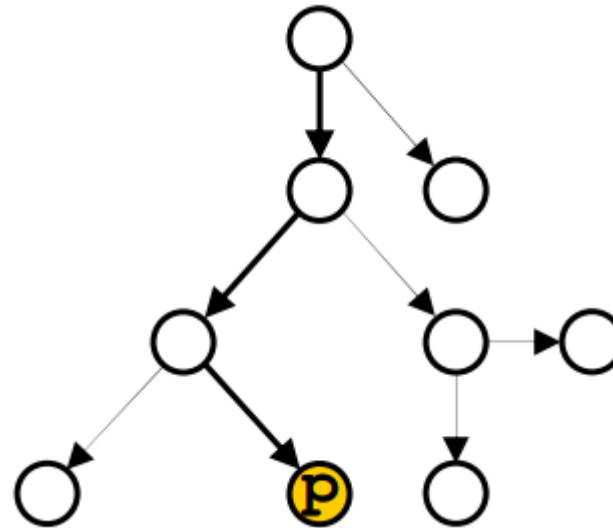
# Properties

- **E[] p** “Exists globally p” meaning there is an execution path in which p holds for all the states of the path



# Properties

- **$E \leftrightarrow p$**  “Exists eventually p” meaning there is an execution path in which p eventually (in some state of the path) holds

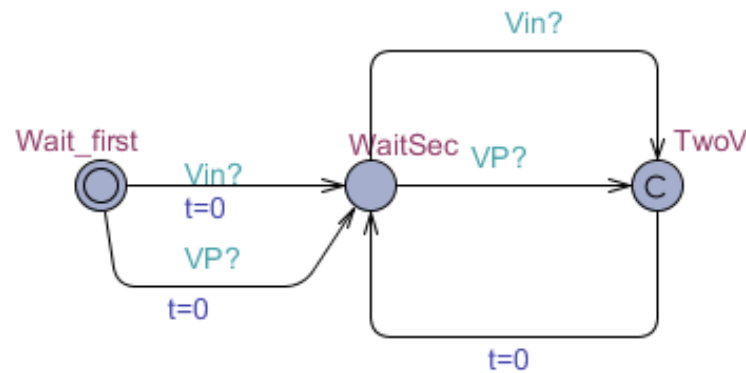


# UPPAAL TCTL Properties

- $A[]p$ ,  $A\langle\rangle p$ ,  $E\langle\rangle p$ ,  $E[]p$ ,  $p\text{--}\rightarrow p'$  ( $p$  imply  $p'$ )
- $p$  can be
  - Location of a process:  $a.l$
  - Data guard
  - Clock guard
  - $p$  and  $p'$
  - $p$  or  $p'$
  - Not  $p$
  - $p$  imply  $p'$

# Monitors

- Sometimes we need assistance to express our requirements
- The maximum interval between two ventricular events ( $V_{in}, V_P$ ) should be no larger than 1000ms
- $A[]$  ( $PM.TwoV \text{ imply } PM.t \leq 1000$ )



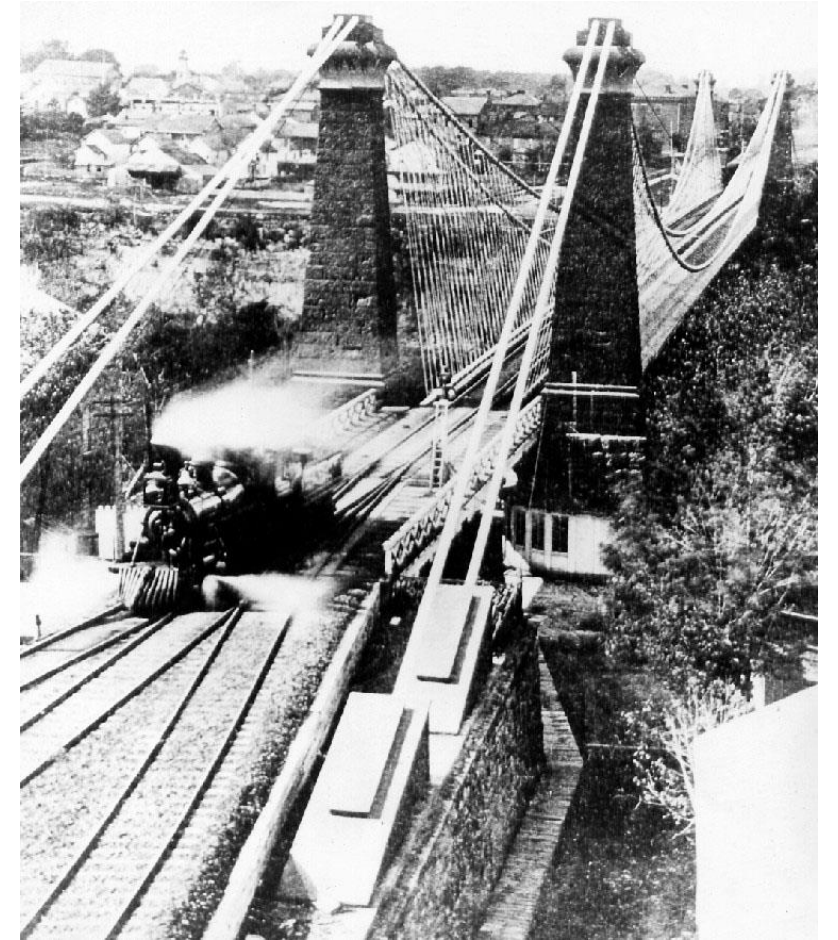
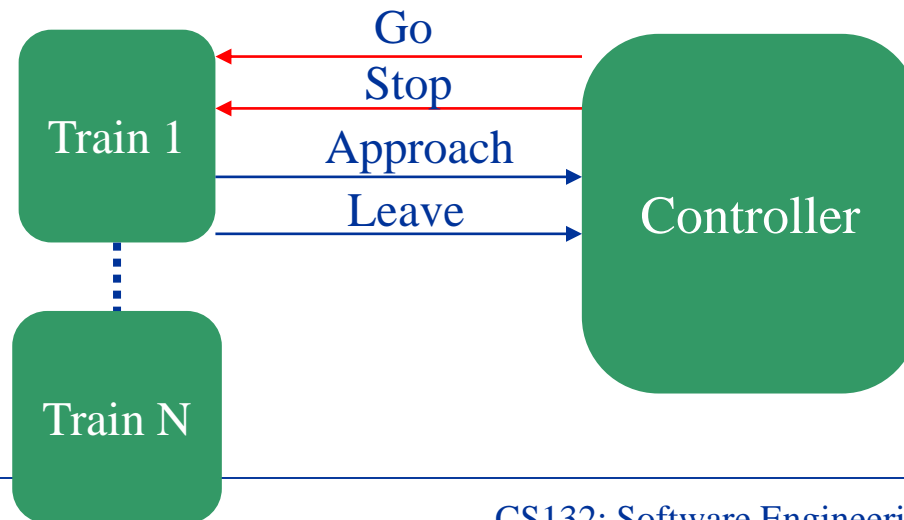
# UPPAAL Examples

# Reference

- Downlaod
  - [www.uppaal.org](http://www.uppaal.org)
- Tutorials
  - On the same webpage
  - Recommended:
    - UPPAAL 4.0: Small Tutorial.
    - Uppaal SMC Tutorial

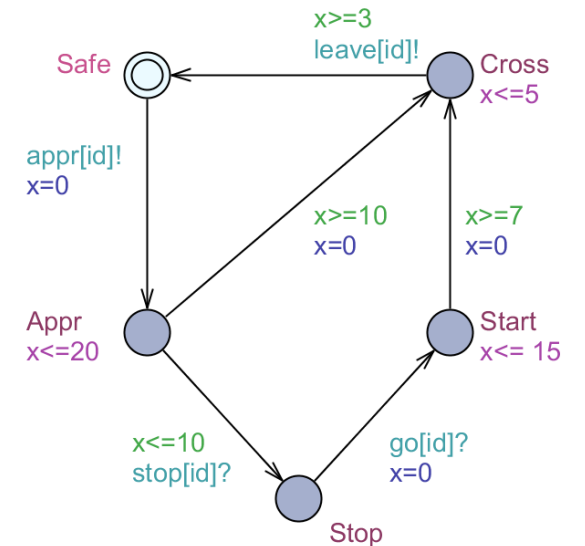
# Example: Train-Gate

- Niagara Falls Suspension Bridge
- One passage, multiple entries
- Design a software controller that makes sure:
  - Every train arrives the bridge eventually crosses
  - Only one train on the bridge at the same time



# Modeling Trains (Environment)

- Each train has an id
- Each train can approach the gate at any time
- Approaching takes 10-20 sec
- The gate controller can stop a train within 10 sec after its approaching, otherwise the train will cross
- After receive a GO signal, the train will start within 7-15 sec
- Crossing takes 3-5 sec





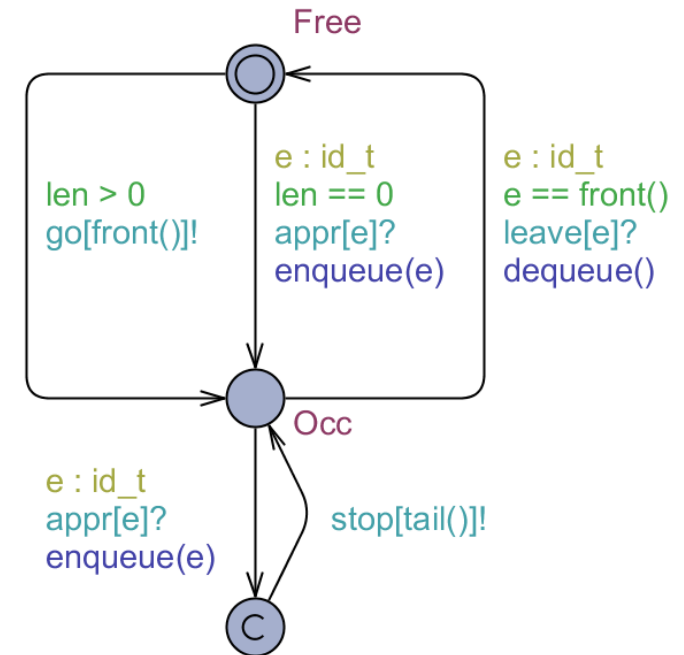
# Example: Train-Gate (cont.)

- Gate controller maintains a queue
- If queue empty and a train approaches, gate stay occupied
- If the gate is occupied and a train approaches, stop the last one in queue
- If the train at the front of the queue leaves, remove it from the queue
- If the gate is free and there are trains in queue, let the front one go

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

// Put an element at the end of the queue
void enqueue(id_t element)
{
    list[len++] = element;
}
```

```
// Remove the front element of the queue
void dequeue()
{
    int i = 0;
    len -= 1;
    while (i < len)
    {
        list[i] = list[i + 1];
        i++;
    }
    list[i] = 0;
}
```



```
// Returns the front element of the queue
id_t front()
{
    return list[0];
}

// Returns the last element of the queue
id_t tail()
{
    return list[len - 1];
}
```

# Example: Train-Gate (cont.)

- Train 0 can eventually cross
  - $E \triangleleft \text{Train}(0).\text{Cross}$
- Train 0 can be crossing bridge while Train 1 is waiting to cross
  - $E \triangleleft \text{Train}(0).\text{Cross}$  and  $\text{Train}(1).\text{Stop}$
- Train 0 can cross bridge while the other trains are waiting to cross
  - $E \triangleleft \text{Train}(0).\text{Cross}$  and  $(\text{forall } (i:\text{id}-t) \ i \neq 0 \text{ imply } \text{Train}(i).\text{Stop})$
- There can never be N elements in the queue
  - $A[] \text{ Gate.list}[N] == 0$
- There is never more than one train crossing the bridge
  - $A[] \text{ forall } (i:\text{id}-t) \text{ forall } (j:\text{id}-t) \text{ Train}(i).\text{Cross} \ \&\& \ \text{Train}(j).\text{Cross} \text{ imply } i == j$
- Whenever a train approaches the bridge, it will eventually cross
  - $\text{Train}(1).\text{Appr} \rightarrow \text{Train}(1).\text{Cross}$
- The system is deadlock-free
  - $A[]$  not deadlock

