# ENSE 623 Systems Validation and Verification

## *System Behavior Modeling and Validation with UPPAAL*

Mark Austin

E-mail: austin@isr.umd.edu

Institute for Systems Research, University of Maryland, College Park

# Table of Contents

# System Modeling with UPPAAL

UPPAAL is ...

**... a toolset for simulation, validation (via graphical simulation) and verification (via automatic model checking) of real time systems.**

modeled as ...

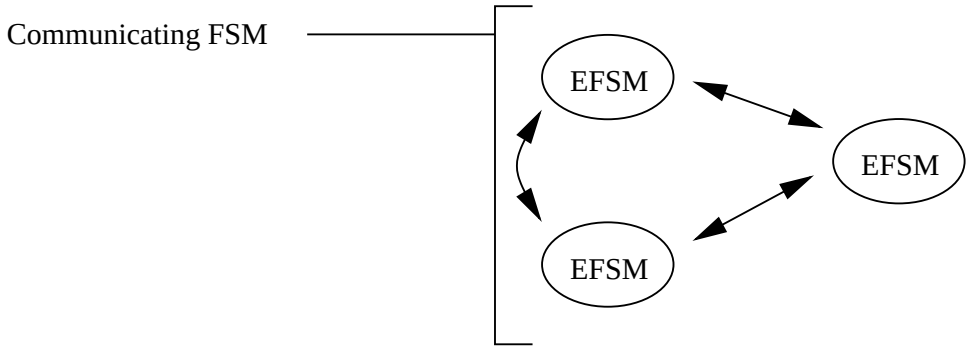
**... networks of timed automata based on temporal logic.**

UPPAAL is appropriate for systems that can be ...

**... modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks that communicate through channels and/or shared variables.**

# System Modeling with UPPAAL

## Networks of Communicating Extended Finite State Machines

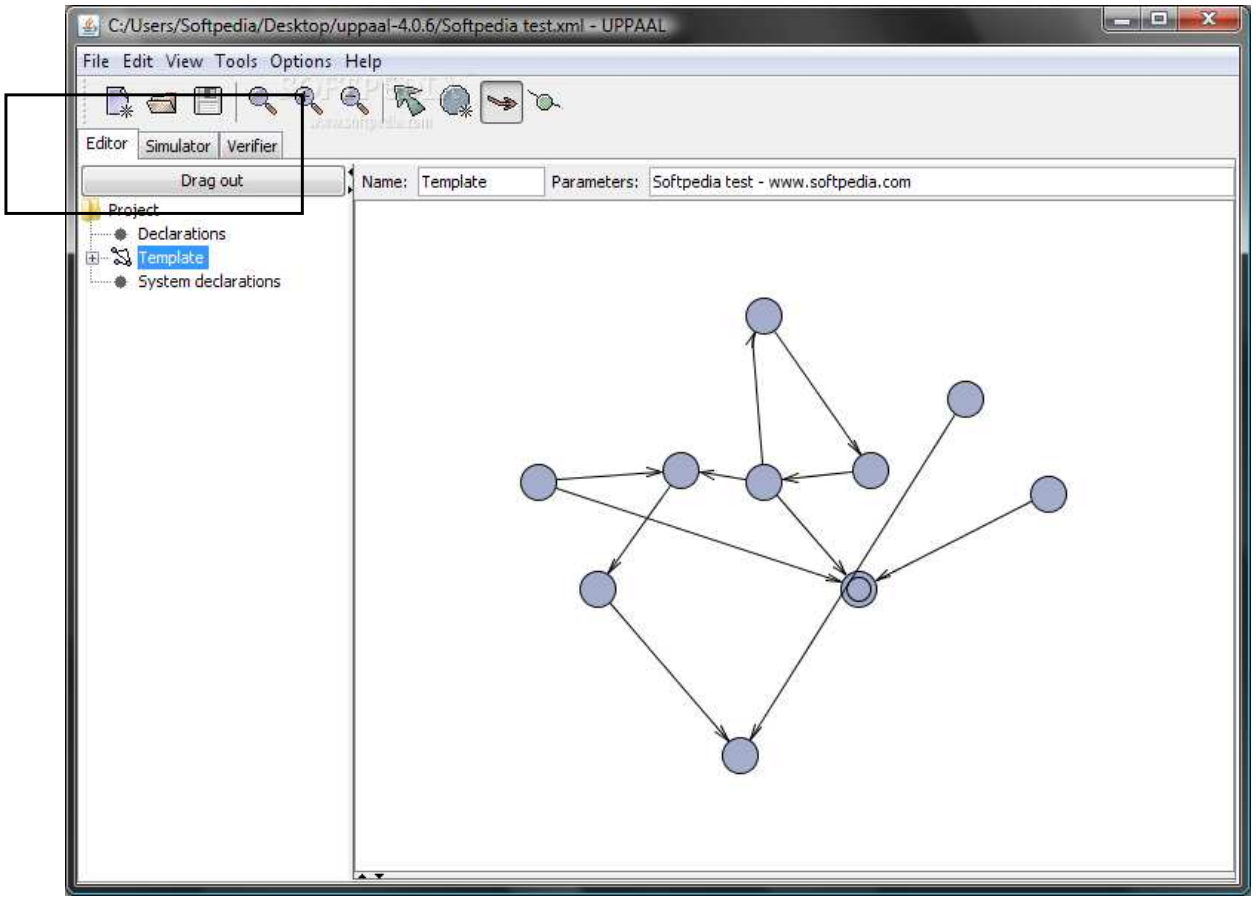



## UPPAAL Modeling Procedure

- Model a system using timed automata.
- Simulate the model.
- Verify properties on the model.

# System Modeling with UPPAAL

## UPPAAL Graphical User Interface



# System Modeling with UPPAAL

**Part 1. Description Language**

- The description language is a non-deterministic guarded command language with data types.
- System behavior can be described as networks of timed automata extended with variables.

**Part 2. Simulator**

- The simulator enables examination of possible dynamic executions/behaviors of a system.
- A particular simulation explores only a:

  **...particular execution trace (i.e. sequence of states of the system).**

  and, therefore, prividesan inexpensive means for fault detection.

# System Modeling with UPPAAL

**Part 3. Model Checker**

- The model checker covers exhaustive dynamic behavior of the system, covering all possible behaviors of the system. It assertains ...

  **... whether certain combinations of control nodes and constraints on clocks and integer variables are reachable from an initial configuration.**

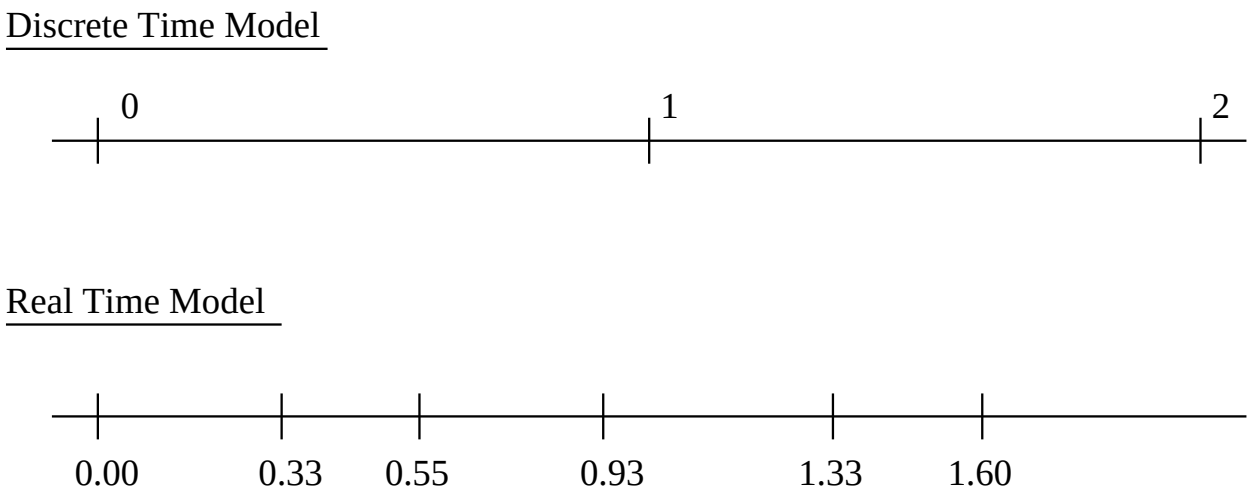
- Other properties such as bounded liveliness can be checked by...

  **...reasoning about the system in the context of testing automata or annotating the system description with debugging information and then checking reachability properties.**

- A diagnostic trace can be automatically reported explaining why the property is satisfied or not. This trace can be visualized by running it through the simulator.

# Dense Time Models

## Schematic of Discrete and Dense Time




## Mathematical Definitions (Furia et al., 2010)

- A discrete set consists of isolated points (e.g., the integers 0, 1, 2, ...)
- A dense set (ordered by $<$) is such that for every two points $t_1$, $t_2$, such that $t_1 < t_2$, there is always another point inbetween (i.e., $t_1 < t_3 < t_2$).

# Time Model

**UPPAAL Time Model**

- UPPAAL uses a ...

  **... dense-time model where a clock variable evaluates to a real number.**

- All the clocks progress synchronously.

- To avoid the need for dealing with an infinite number of clock values, ...

  **UPPAAL uses zones defined by sets of constraints on clock values.**

- When a system is in a particular state (location), we do not have a concrete value of time – instead, ...

  **... differences defined by region boundaries.**

# Timed Automata

## Definition of Timed Automata

A timed automaton is a stripped-down finite-state machine extended with clock variables.

### Timed Automata

**Definition**

A timed automaton $A$ over clocks $C$ and actions $Act$ is a tuple $(L, l_0, E, I)$, where:

- $L$ is a finite set of locations
- $l_0 \in L$ is the initial location
- $E \subseteq L \times \mathcal{B}(X) \times Act \times \mathcal{P}(C) \times L$ is the set of edges
- $I : L \longrightarrow \mathcal{B}(X)$ assigns to each location an invariant

Timed automata correspond to labeled transition systems (LTSs) extended with time.

# Timed Automata

**Timed Label Transition Systems**

- S is the set of states.
- $s_o \in$ S is the initial state.
- Act is the set of observable actions. $A_{\tau\delta}$ = Act $\cup \{\tau\} \cup \{\delta | \delta \in R_+\}$ is the action set with additional internal $\tau$ and delay $\delta$ actions.
- $\rightarrow \subseteq$ S $\times$ $A_{\tau\delta}$ $\times$ S is the transition relation satisfying the consistency relationships:
  - **Time Determinism**. Whenever $s \xrightarrow{\delta} s^{'}$ and $s \xrightarrow{\delta} s^{''}$, $s^{'} = s^{''}$.
  - **Time Addivity**. $\forall\ s, s^{'} \in$ S, $\exists\ s^{'} \in$ S, s $\xrightarrow{\delta_1} s^{'}$, $s^{'} \xrightarrow{\delta_2} s^{''}$, iff $s \xrightarrow{\delta_1+\delta_2} s^{''}$.
  - **Null Delay.** $\forall\ s, s^{'} \in$ S; $s \xrightarrow{0} s^{'}$ iff s = $s^{'}$.

# Timed Automata

**States and Transitions**

The system state is defined by:

- The locations of all automata,
- The clock constraints, and
- The values of the discrete variables.

In general transitions are not allowed to be carried out simultaneously.

Therefore the automata do not really operate in parallel, but rather in an interleaving way.

Its capabilities include checking invariant and reachability properties by exploring the state-space of a system.

# Nodes and Transitions

**Nodes**

- Nodes can have three special marks:

  **... initial, committed and urgent**.

- When a node is initial the automaton will start in this node – the initial node is marked by a double circle. Every automaton has only one initial node.

- When a node is **committed**, a transition from that node to the next node has to be taken immediately. No other transition in other automata can be taken in between.

- When a node is **urgent** it must take the next transition as soon as this is possible. It does not rule out other actions happening in between but when it is possible to take the transition it does rule out other possible actions at that point.

# Nodes and Transitions

**Transitions**

- The transitions between the nodes define the behaviour of the system.
- A transition can have three types of labels:

  **... a guard, a synchronisation and a reset.**

- Every transition will have at least one label.

**Variables**

- UPPAAL uses variables globally.
- The value of a variable will be the same in every automaton. Suppose, for example, that variable x is reset to n in an automaton p1. Then ...

  **... if there exists an automaton p2, x also acquires the value n for that automaton.**

# Syntax and Semantics

## Guards

- A guard is a condition which has to be met in order to take the transition.
- Formally guards are conjunctions of timing and data constraints of the form
- Guards take the form: x "H" n or x - y Ãć n where x and y are variables, n is a natural number in an integer constraint but an arbitrary integer in a timing constraint and "H" is an element of the set < , > , = , <=, >=, != .

## Reset Operations

- Reset operations set a clock's value back to zero.
- A reset only takes place after the guards (if any) of the transition is satisfied and a synchronisation is possible.

# Syntax and Semantics

**Channels and Synchronization**

- Channels serve the sole purpose of letting two automata communicate.

  **... Synchronization mechanisms correspond to handshaking.**

- A synchronisation takes place trough a channel and is always only between two automata.

- Two processes take a transition at the same time –

  **... one will have ”a!” the other will have ”a?” (the synchronization channel).**

- If the transition carries a synchronization annotation of the form a? (or a!) then some corresponding transition (also labeled by a! or a?) of some other timed automaton has to be taken simultaneously.

# Syntax and Semantics

Automaton elements may also be classified as being committed or urgent. Both annotations disallow the passage of time while the location is active.

**Committed**

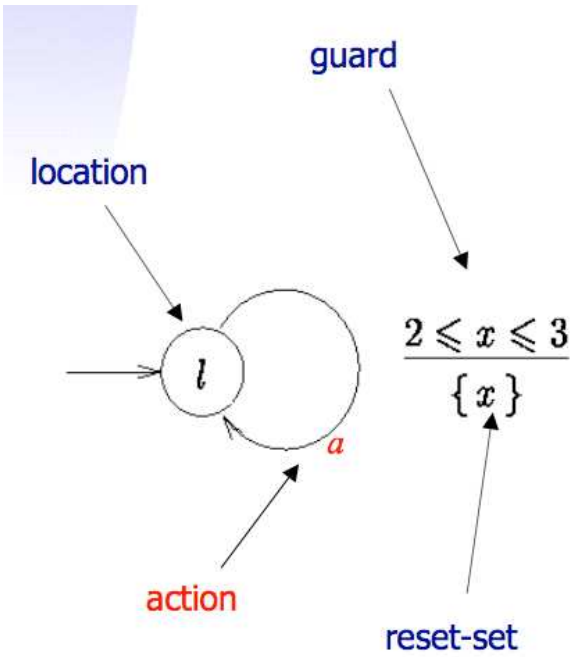
- Committed locations require the next system action to involve a transition whose source state is the committed location (see pg. 398 of Knapp, 2002.)
- Therefore, atomic transactions that involve more than a single transition can be modeled by labeling the intermediate transitions as committed.

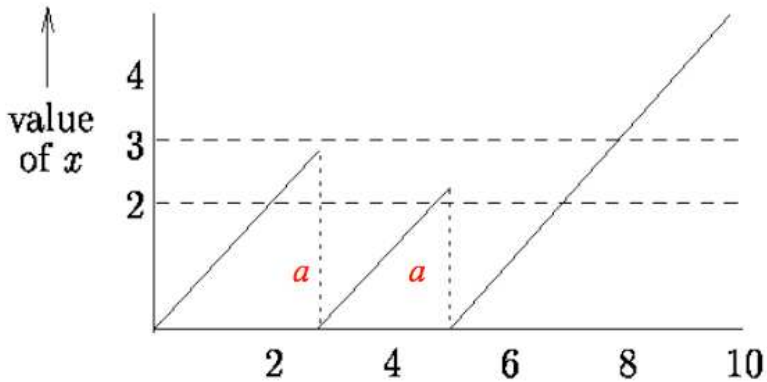
**Urgent**

- A channel can be declared urgent to disallow the passage of time as soon as synchronization of the channel is enabled (see pg. 398 of Knapp, 2002).

# Example 1. Time-Dependent Clock Behavior

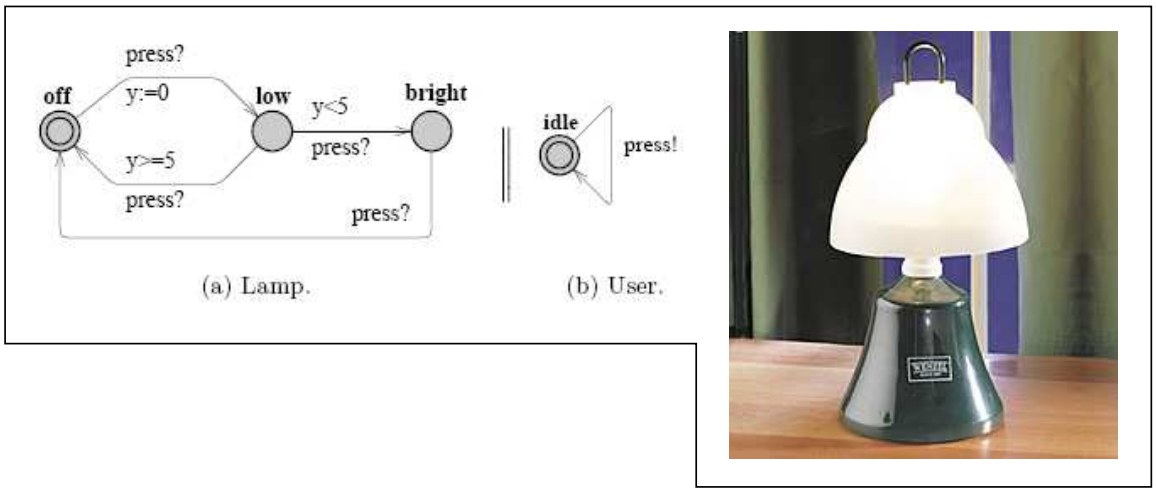
**Time-Dependent Clock Behavior**




**Interpretation:** If an action “a” occurs while the guard condition evaluates to true (i.e., the transition is enabled), then the clock “x” will be reset.

# Example 2. Operation of a Simple Lamp





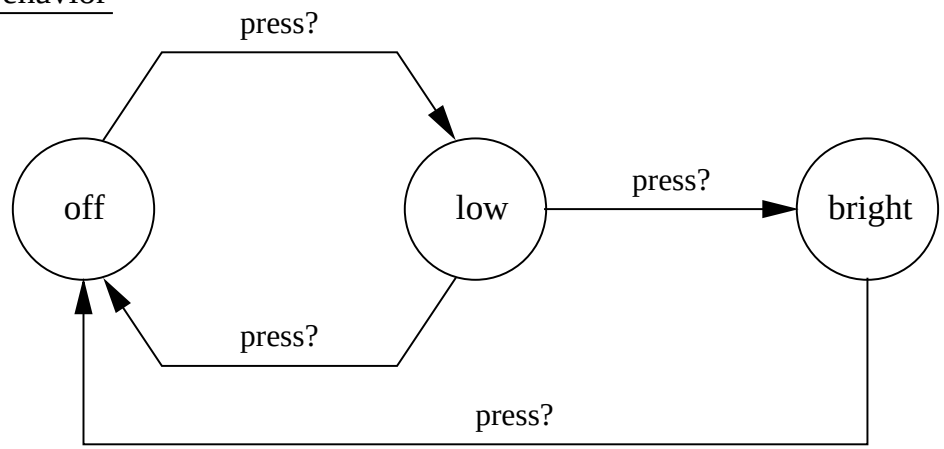

Operation of the lamp is defined by the following rules:

- If the user presses a button (i.e., synchronizes with press?), then the lamp is turned on.
- If the user presses the button again, the lamp is turned off.
- If the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright.

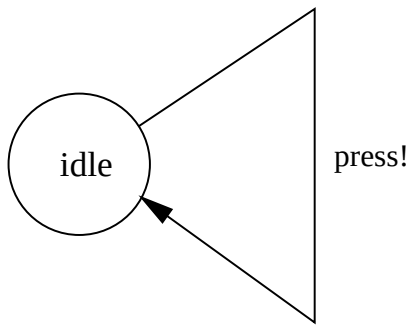
# Example 2. Operation of a Simple Lamp

## Action-Based View of Lamp and User Behavior

Lamp Behavior



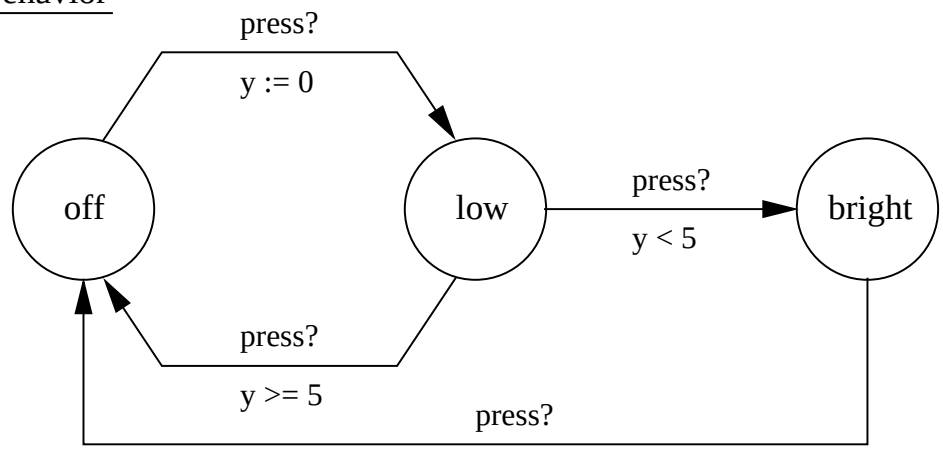

User Behavior



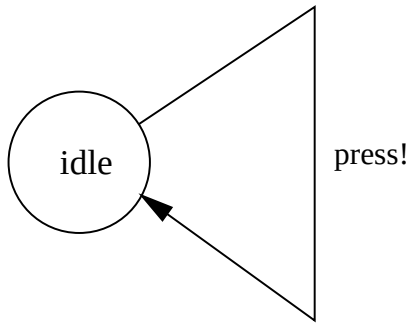
# Example 2. Operation of a Simple Lamp

## Add a Real-Valued Clock y to Behavior Model

Lamp Behavior




User Behavior



# Example 2. Operation of a Simple Lamp

**System Composition**

$$\text{System Behavior} = (\quad \text{Lamp Behavior} \quad \| \quad \text{User Behavior} \quad )$$

**Declarations** (...these are global).

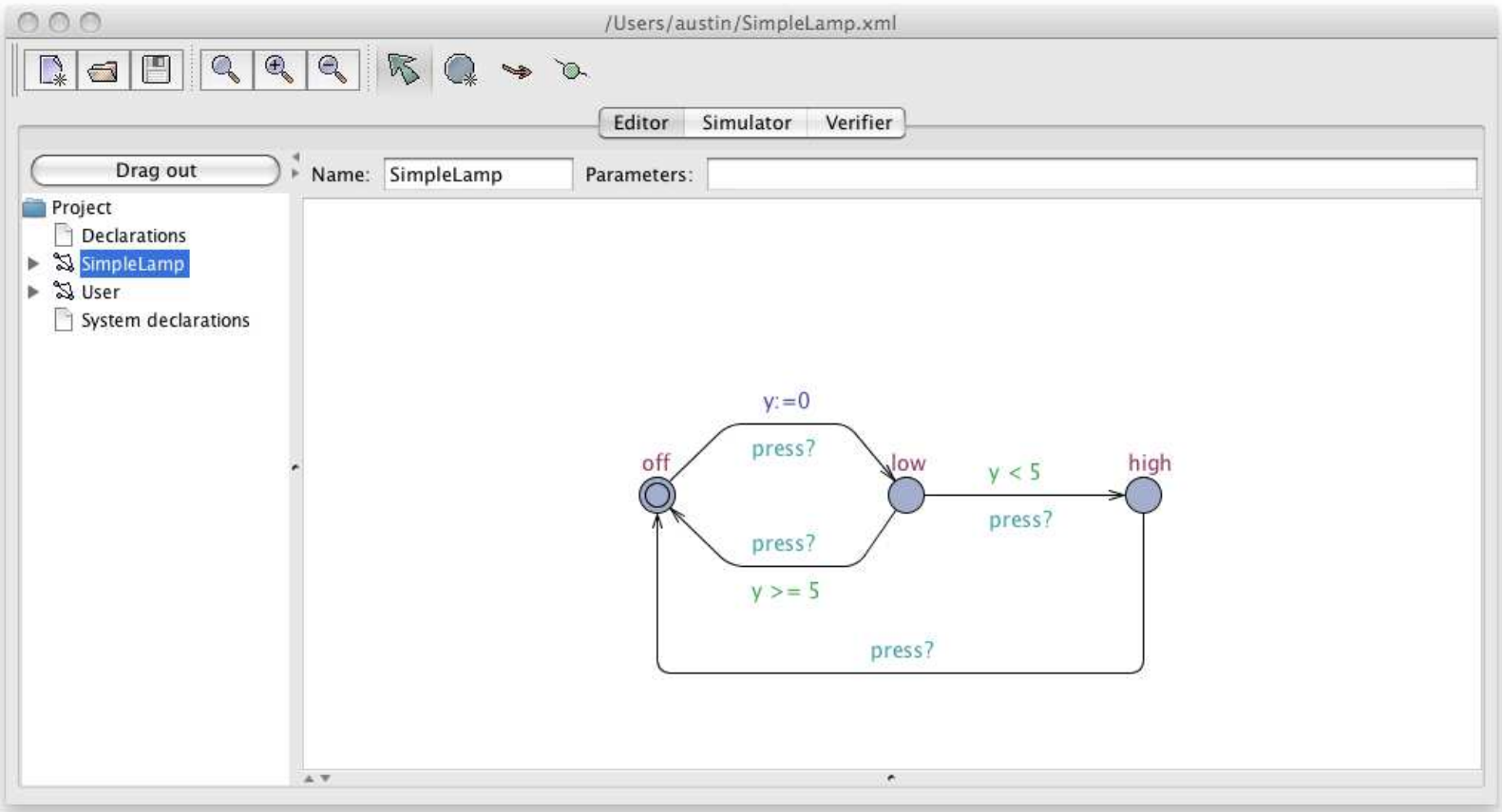
```
chan press;
clock y;
```

**System Declarations**

```
User1 = User();
Lamp1 = SimpleLamp();

// Compose processes into a system model....

system User1, Lamp1;
```

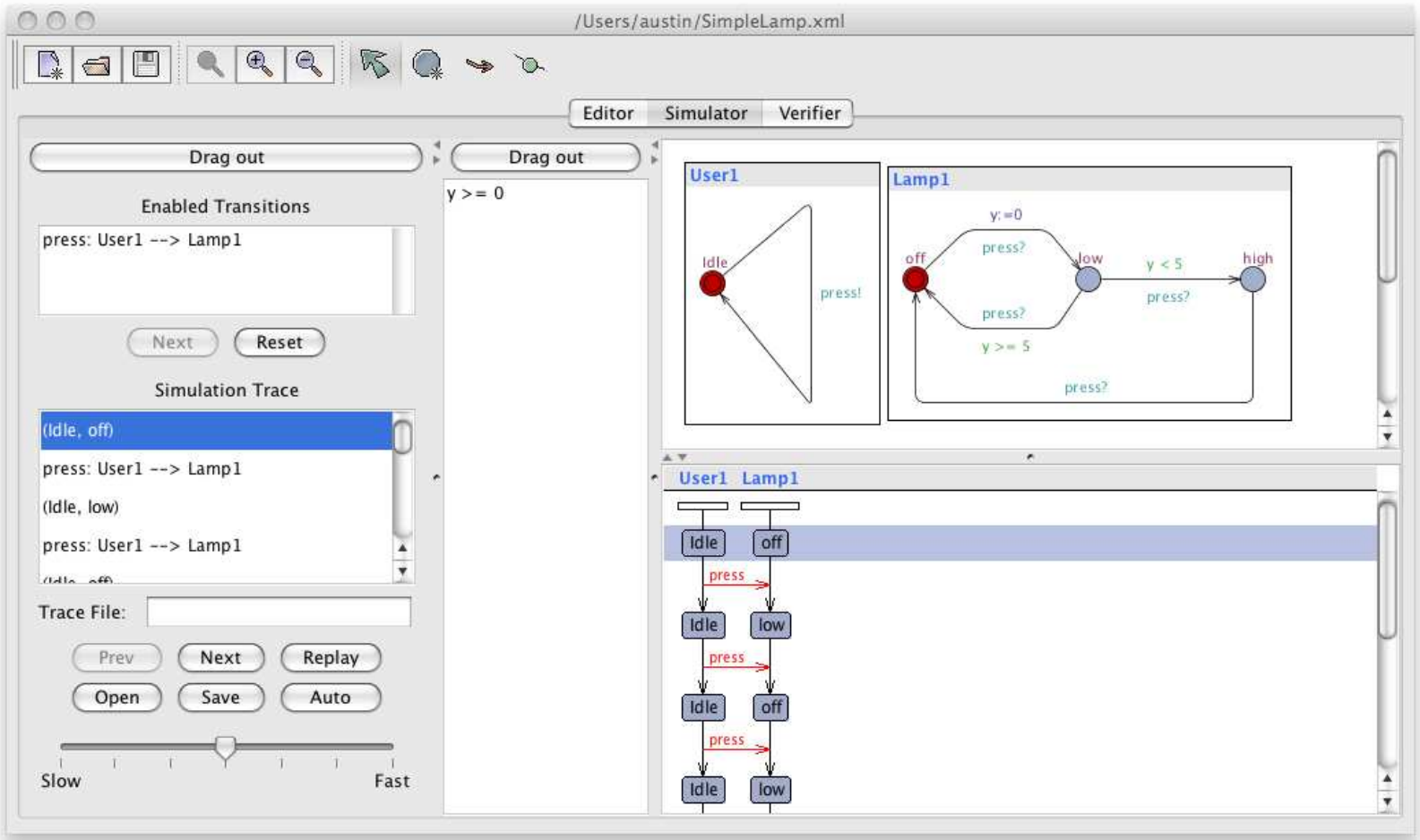
# Example 2. Operation of a Simple Lamp

## UPPAAL Editor



# Example 2. Operation of a Simple Lamp

## UPPAAL Simulator



# Example 2. Operation of a Simple Lamp

**A Few Obervations**

In our simplified model:

- While user behavior is defined in terms of sequences of press! events, there is nothing in user model to say how fast these clicks must occur.
- Thus, the user model really doesn't do a good job of ...

  **... driving the critical elements of the lamp behavior.**

We can solve this problem by ...

**... add a clock to the user behavior and an invariant to force timely sequences of press events.**

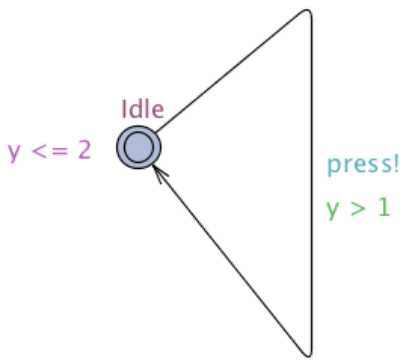
# Example 2+. Operation of a Simple Lamp

## Invariants

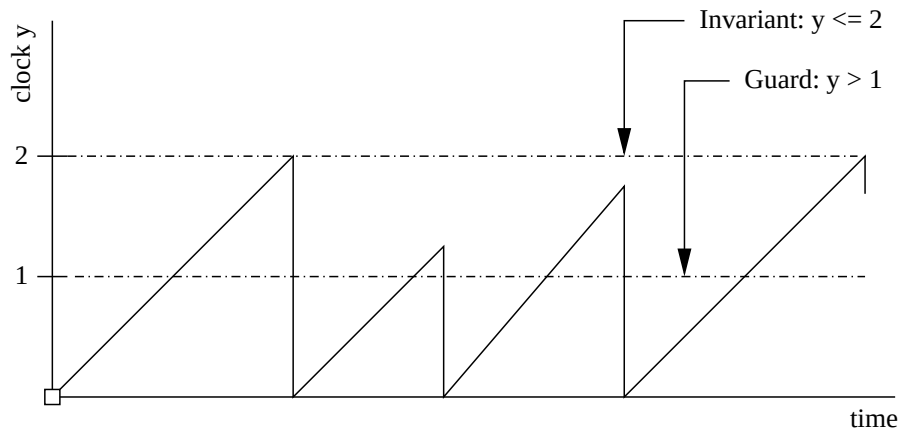
- An invariant is a progress condition.

## Adding an Invariant to the User Behavior

Modified User Behavior

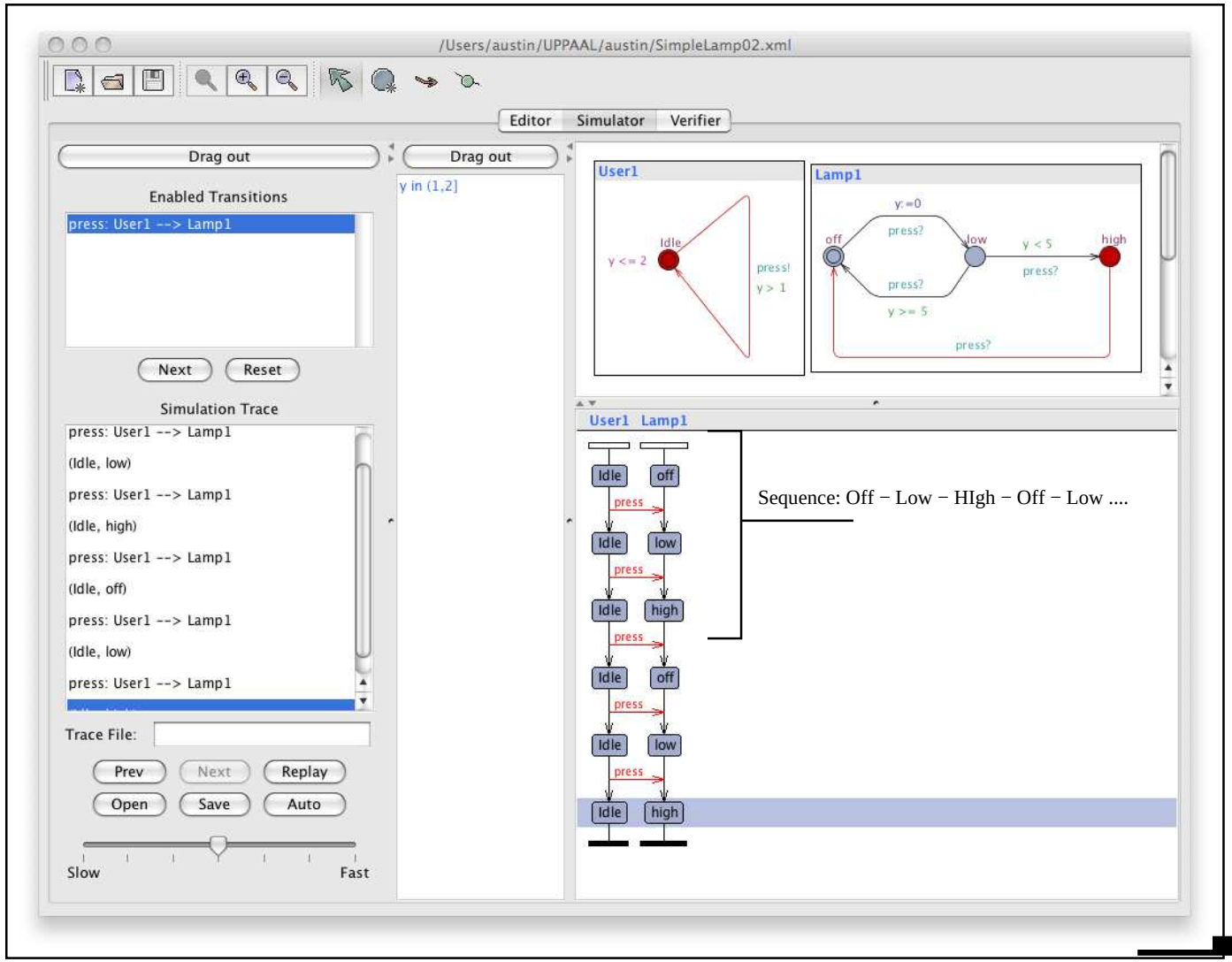



Time−Dependent Behavior




**Remark.** The guard condition activates the transition when clock $y > 1$. The invariant states that the system cannot stay in the idle state for more than 2 units of time.

# Example 2+. Operation of a Simple Lamp



# Simulation versus Verification in UPPAAL

**Simulation in UPPAAL**

- Allows for virtual interaction with the system.
- The simulator shows the states of compound automata and the values of variables.
- State transitions may be chosen either manually or randomly.

**Verification in UPPAAL**

- Accepts user formulated properties to be verified on a particular timed-automata model.
- Displays results of the verification: true or false.
- Provides an event trace example if the property proof requires one.

# Verification in UPPAAL

**Framework for Verification in UPPAAL**

Let p and q be state formulas (e.g., $y < 2$).

UPPAAL understands the following types of queries:

- $E <>$ p: there exists a path where p eventually holds true.
- $E$ [ ] p: there exists a path where p always holds.
- $A <>$ p: for all paths p will eventually hold.
- $A$ [ ] p: for all paths p always holds.
- $p \longrightarrow q$: whenever p holds q will eventually hold.

# Example 2. Verification of Lamp Behavior

**Behavior of Simple Lamp and Simple User Model**

- There exists a path where `Lamp1.high` eventually holds true?

  `Query:`

  ```
  E<> Lamp1.high
  ```

  `Status:`

  **Property is satisfied.**

- There exists a path where `Lampl.high` always holds?

  `Query:`

  ```
  E[] Lamp1.high
  ```

  `Status:`

  **Property is not satisfied.**

# Example 2. Verification of Lamp Behavior

**Behavior of Simple Lamp and Simple User Model**

- For all paths `Lamp1.high` will eventually hold true?

  `Query:`

  ```
  A<> Lamp1.high
  ```

  `Status:`

  **Property is not satisfied.**

- For all paths `Lamp1.high` always holds true?

  `Query:`

  ```
  A[] Lamp1.high
  ```

  `Status:`

  **Property is not satisfied.**

# Example 2-2+. Verification of Lamp Behavior

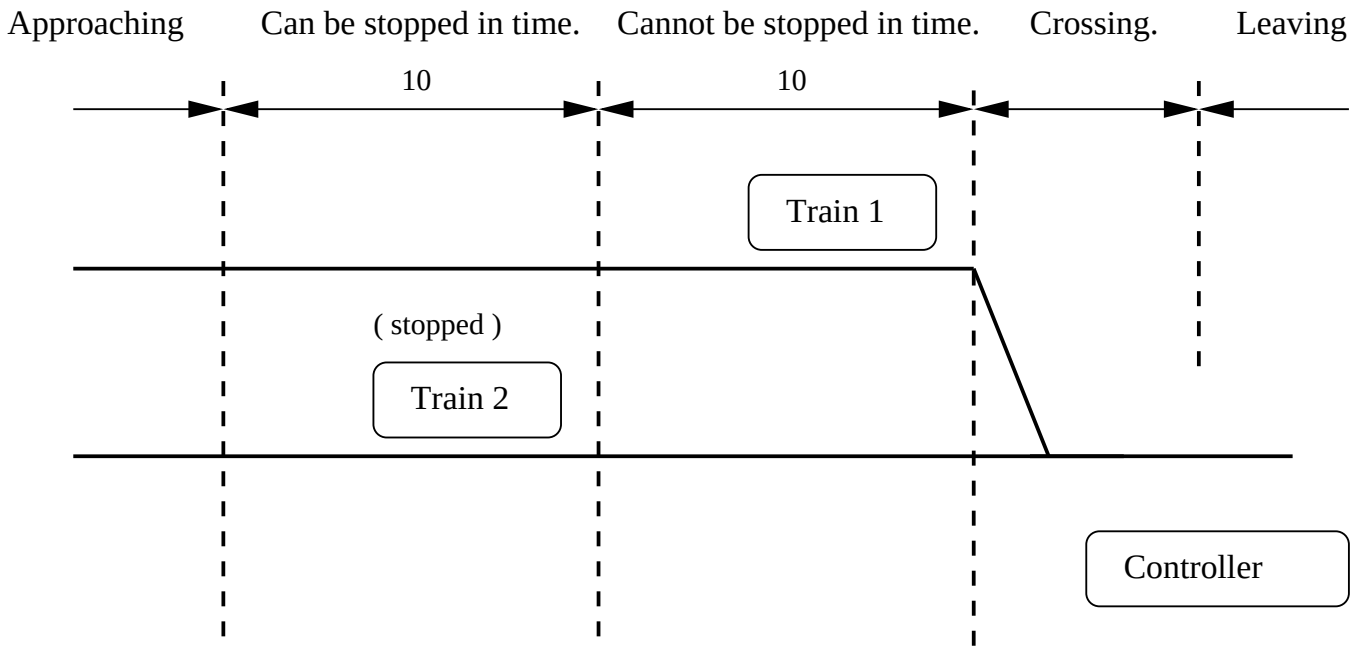
**Comparison of Simple Lamp Behavior with Basic and Enhanced User Models**

| Query | Basic User Model | Enhanced User Model |
|---|---|---|
| `E <> Lamp1.high` | Property is satisfied. | Property is satisfied. |
| `E [] Lamp1.high` | Property is not satisfied. | Property is not satisfied. |
| `A <> Lamp1.high` | Property is not satisfied. | Property is satisfied. |
| `A [] Lamp1.high` | Property is not satisfied. | Property is not satisfied. |
| `Lamp1.low --> Lamp1.high` | Property is not satisfied. | Property is satisfied. |

# Example 3. Train Crossing Problem

**Problem Statement**

A railway control system controls access to a bridge for several trains.




The bridge is a critical shared resource that may be accessed by only one train at a time.

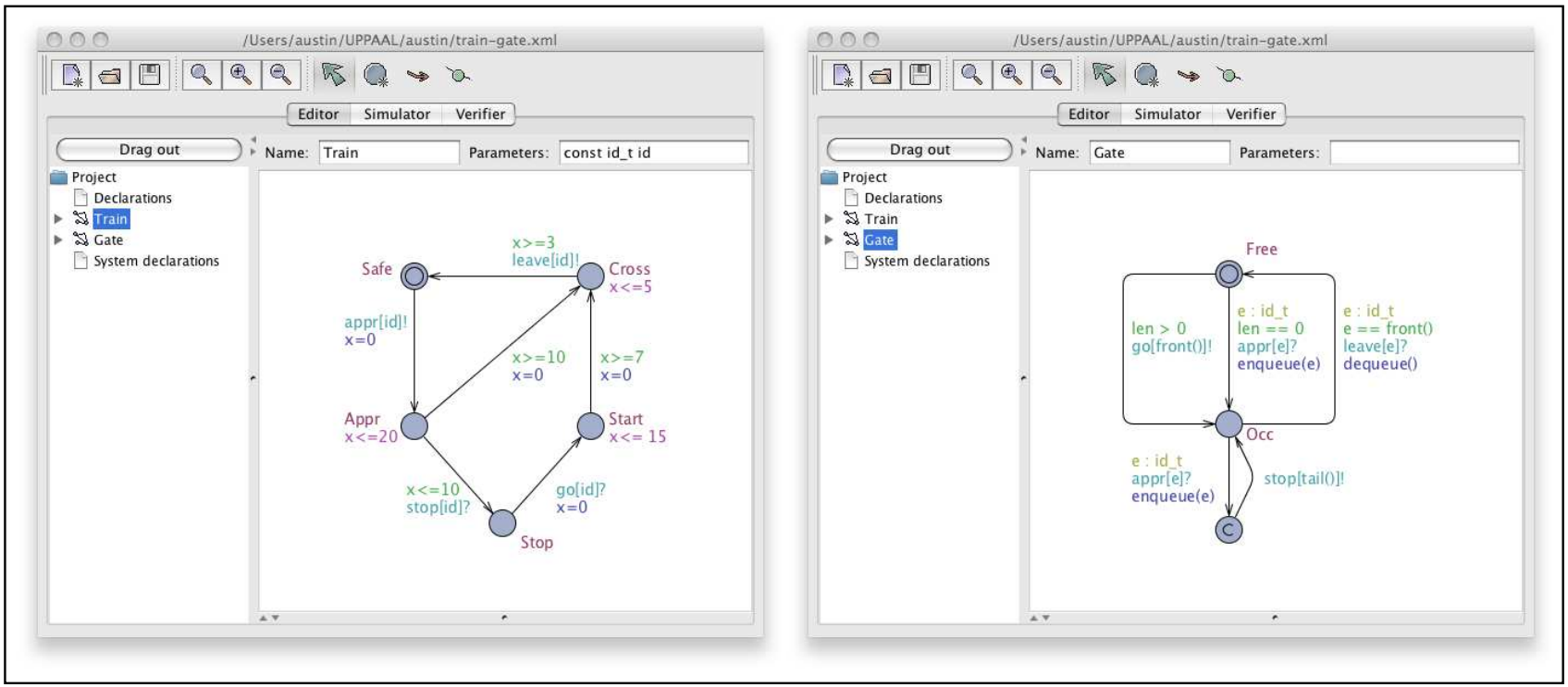
# Example 3. Train Crossing Problem

**Key Modeling Parameters** (adapted from Behrmann et al.)

- The system is defined by a number of trains (in this case 2) plus a controller.
- A train cannot be stopped instantly. Restarting a train also takes time.
- An approaching train sends an `appr!` signal to the controller.
- Then, it has 10 units of time to receive a stop signal. This allows the train to stop safely before the bridge.
- After these 10 units it takes another 10 units of time to reach the bridge if it is not stopped.
- A stopped train resumes its course when the controller sends a `go!` signal after a previous train has left the bridge and sent a `leave!` signal.

# Example 3. Train Crossing Problem

## UPPAAL Models




**Note.** The location `Appr` has the invariant $0 \leq 20$, meaning that the location must be left within 20 time units. The outgoing transitions are guarded by the constraints $x \geq 10$ and $x \leq 10$.

# Example 3. Train Crossing Problem

**More Points to Note**

- At exactly x = 10, both transitions are enabled. This enables us to take care of any race conditions, should they exist.
- If the train can be stopped (i.e., $x \leq 10$), then the transtion to the location `Stop` is taken, otherwise the train goes to location `Cross`.
- The transition to `Stop` is also guarded by the transition e == id, and is synchronized with `stop?`
- When the controller decides to stop a train, it decides which one (sets e) and synchronizes with `stop!`
- The location `Stop` has no invariant. As such, a train may be stopped for an unlimited amount of time waiting for the sychronization go?

# Example 3. Train Crossing Problem

**Declarations**

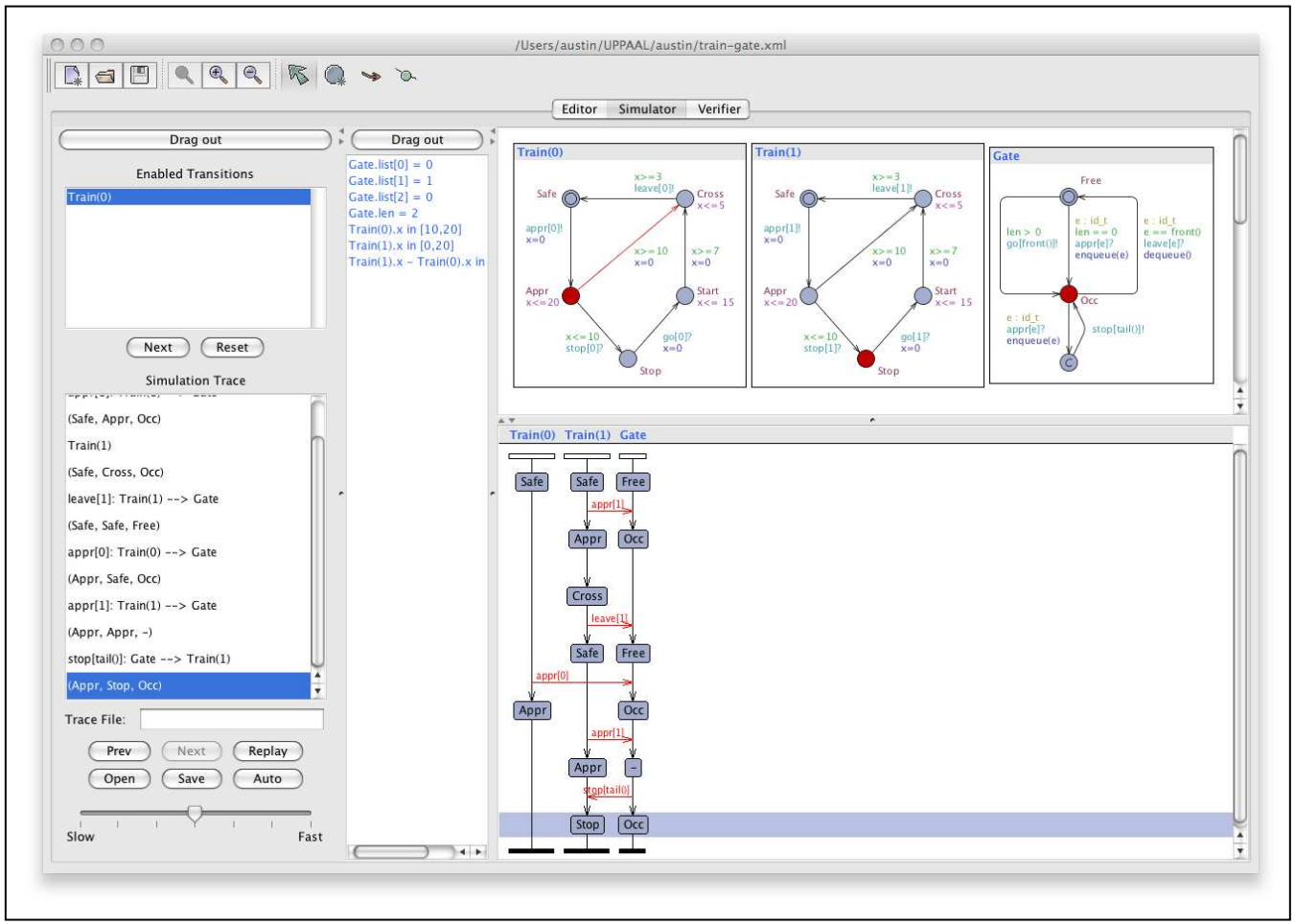
```
const int N = 2;            // # trains
typedef int[0,N-1] id_t;

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

**Reference.** Adapted from:

- Yi W., Petterson P., and Daniels M., “Automatic Verification of Real-Time Communicating Systems by Constraint Solving,” In Proceedings of the 7th International Conference on Formal Description Techniques, pages 223-238, North-Holland. 1994.

# Example 3. Train Crossing Problem

## UPPAAL Simulator



# Verification of Train Crossing Properties

**Verification of Behavior**

- Gate can receive (and store in queue) msg's from approaching trains.

  Query:

  ```
  E<> Gate.Occ
  ```

  Status:

  **Property is satisfied.**

- Trains 0 and 1 can reach crossing.

  Query:

  ```
  E<> Train(0).Cross
  E<> Train(1).Cross
  ```

  Status:

  **Property is satisfied.**

  **Property is satisfied.**

# Verification of Train Crossing Properties

**Verification of Behavior** (cont'd)

- Train 0 can be crossing bridge while Train 1 is waiting to cross.

  Query:

  ```
  E<> Train(0).Cross and Train(1).Stop
  ```

  Status:

  **Property is satisfied.**

- Train 0 can cross bridge while the other trains are waiting to cross.

  Query

  ```
  E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)
  ```

  Status:

  **Property is satisfied.**

# Verification of Train Crossing Properties

**Verification of Behavior** (cont'd)

- There is never more than one train crossing the bridge (at any time instance)

```
A[] forall (i : id_t)
    forall (j : id_t) Train(i).Cross && Train(j).Cross imply i == j
```

- There can never be N elements in the queue (thus the array will not overflow).

```
A[] Gate.list[N] == 0
```

- Whenever a train approaches the bridge, it will eventually cross.

```
Train(0).Appr --> Train(0).Cross\
Train(1).Appr --> Train(1).Cross
```

- The system is deadlock-free.

```
A[] not deadlock
```

**All properties are satisfied.**

# References


- Behrmann G., David A., and Larsen K.G., “A Tutorial on UPPAAL,” Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science, 2004, Volume 3185/2004.
- Mikucionis M. and Sasnaunskaite E., “On-the-fly Testing using UPPAAL,” MS Thesis, Department of Computer Science, Aauborg University, Denmark, June 2003.
- Furia C.A., Mandrioli D., Morzenti A., and Rossi M., Modeling Time in Computing: A Taxonomy and Comparative Survey, ACM Computing Surveys Paper, Vol. 42, No. 2, February 2010.
- UPPAAL 4.0: Small Tutorial, November 2009.