

# Formal methods.

## Introduction to UPPAAL

**Vitaliy Mezhuyev**

---

---

# Introduction

- UPPAAL is an integrated environment for modeling, simulation and verification of real time systems, modeled as network of timed automata.
  - UPPAAL was developed jointly by **Uppsala** University and **Aalborg** University.
  - The first prototype of UPPAAL named TAB, was developed at Uppsala Univ. in 1993 by Wang Yi et al.
  - In 1995, Aalborg University was joined the development and then TAB was renamed into UPPAAL with UPP standing for Uppsala and AAL for Aalborg.
-

# Introduction

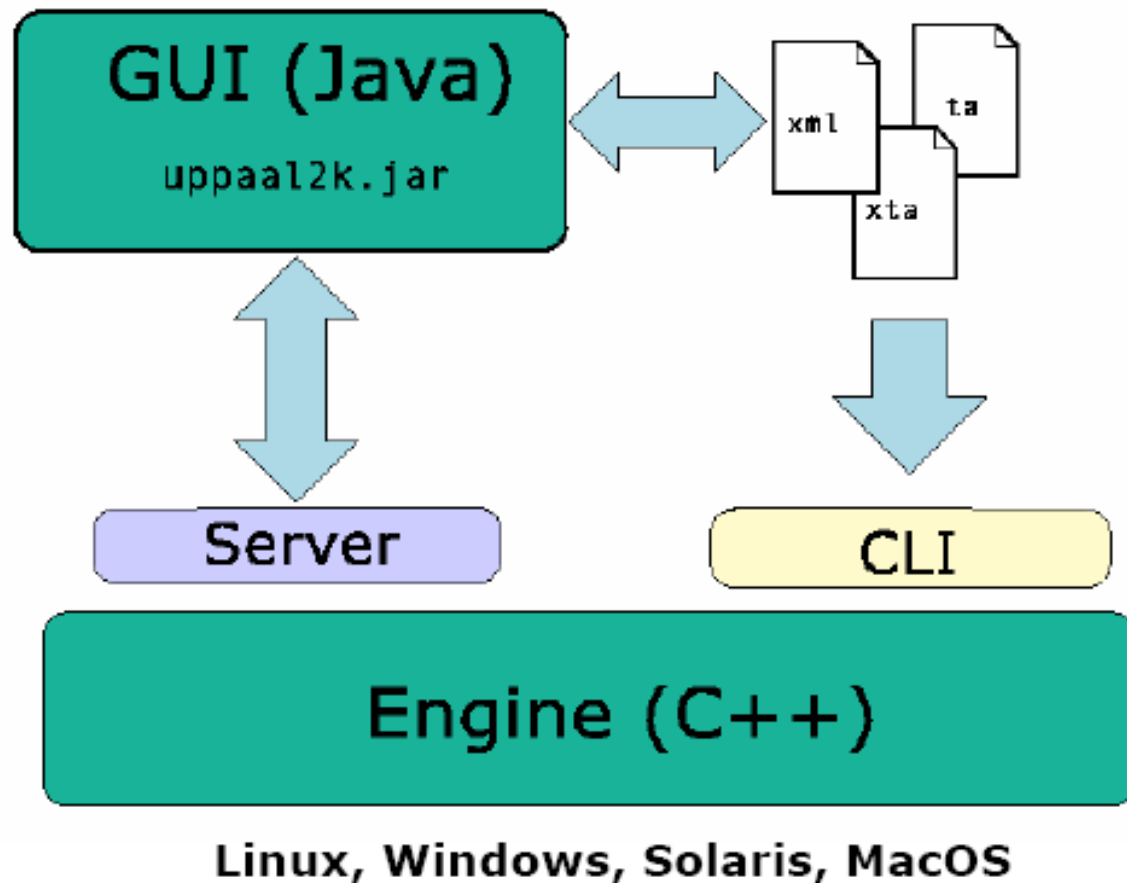
- UPPAAL gives a modeling language to describe the system behavior as networks of automata extended with clock and data variables.
- It models a system as a collection of processes (automations), control structures (for, if, etc.) and clocks, communicating through channels and shared variables.
- It used to make a simulation of the system and check if there is an error in the model.
- It allows to check both **invariants** and **reachability** properties by exploring the state space of a system.

---

# UPPAAL's Features

- A graphical interface allowing to define networks of timed automata by drawing.
  - An automatic compilation of the graphical definition into a textual format, used by the model-checker, thus supporting the important principle “what you see is what your verify”.
  - In case if verification of a particular real-time system fails, a diagnostic trace is automatically reported by UPPAAL.
-

# UPPAAL Architecture

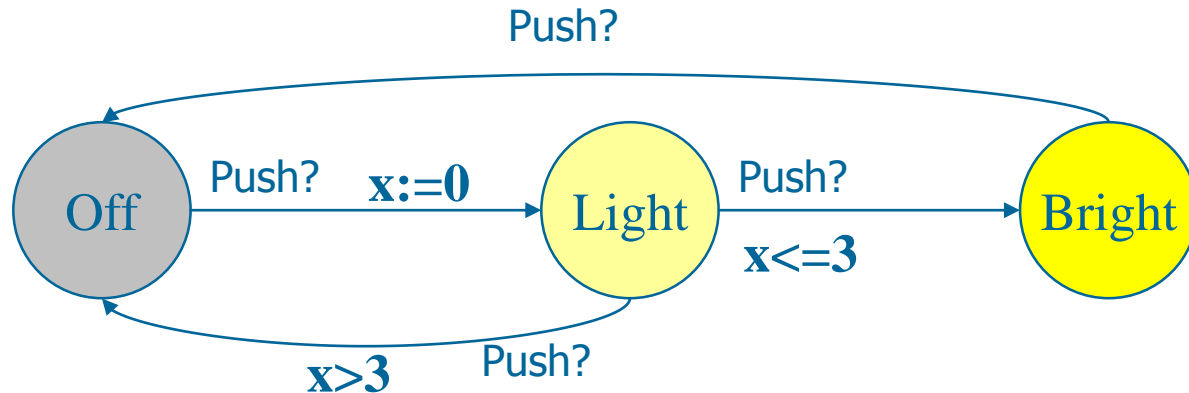


---

# UPPAAL Components

- **UPPAAL** consists of three main parts:
    - ▣ a **description language**,
    - ▣ a **simulator**, and
    - ▣ a **model checker**.
  - The **description language** is a non-deterministic guarded command language with data types. It is used to describe a system as a *network of timed automata* in graphical or textual format.
  - The **simulator** enables examination of *possible* dynamic executions of a system during modeling.
  - The **model checker** exhaustively checks *all* possible states.
-

# Example: Intelligent Light Control

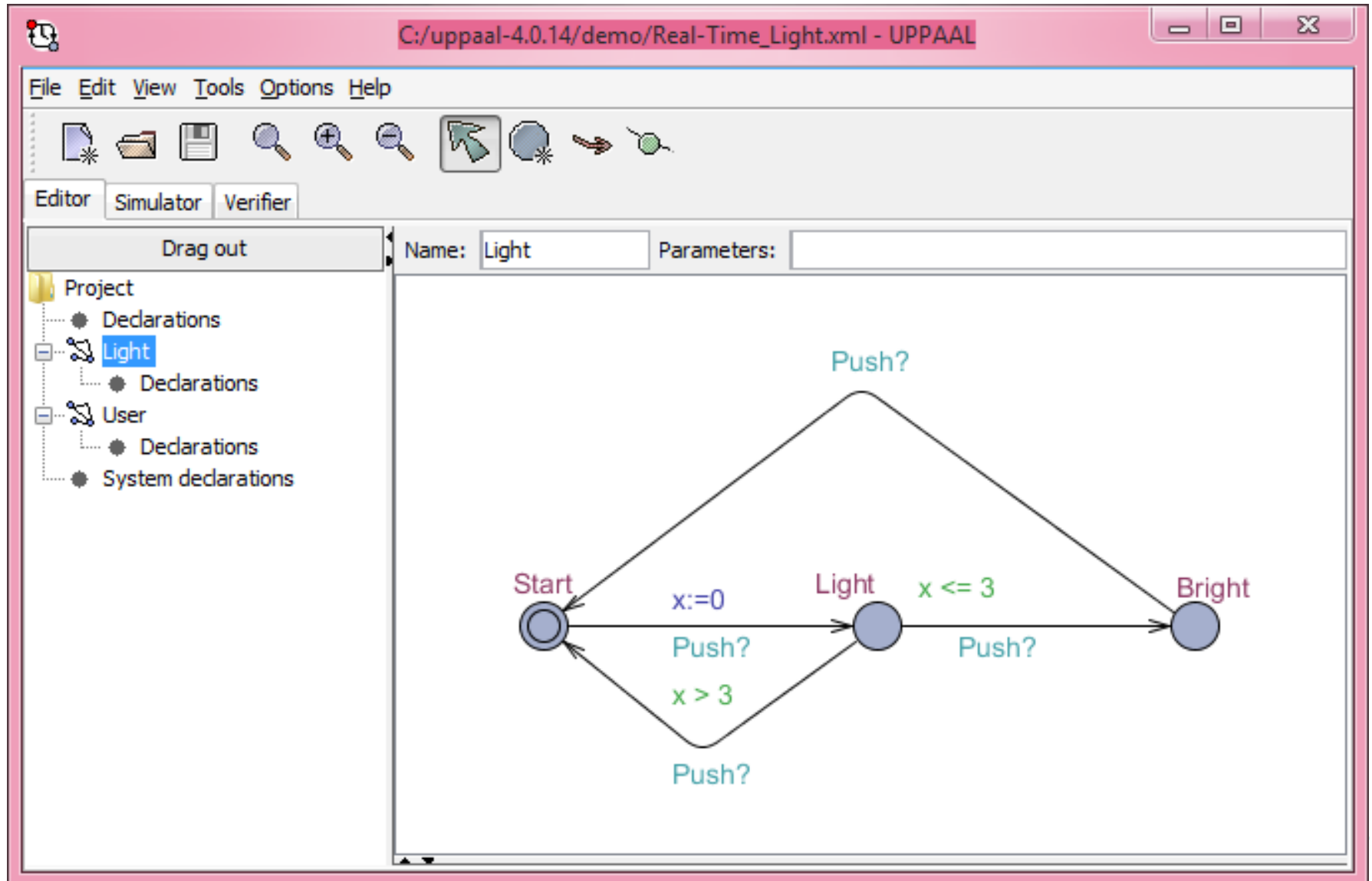


## Requirements:

- If a user presses the light control, then it lights;
- If a user **quickly** presses the light control, then the light should get brighter;
- if the user **slowly** presses the light control, the light should turn off.

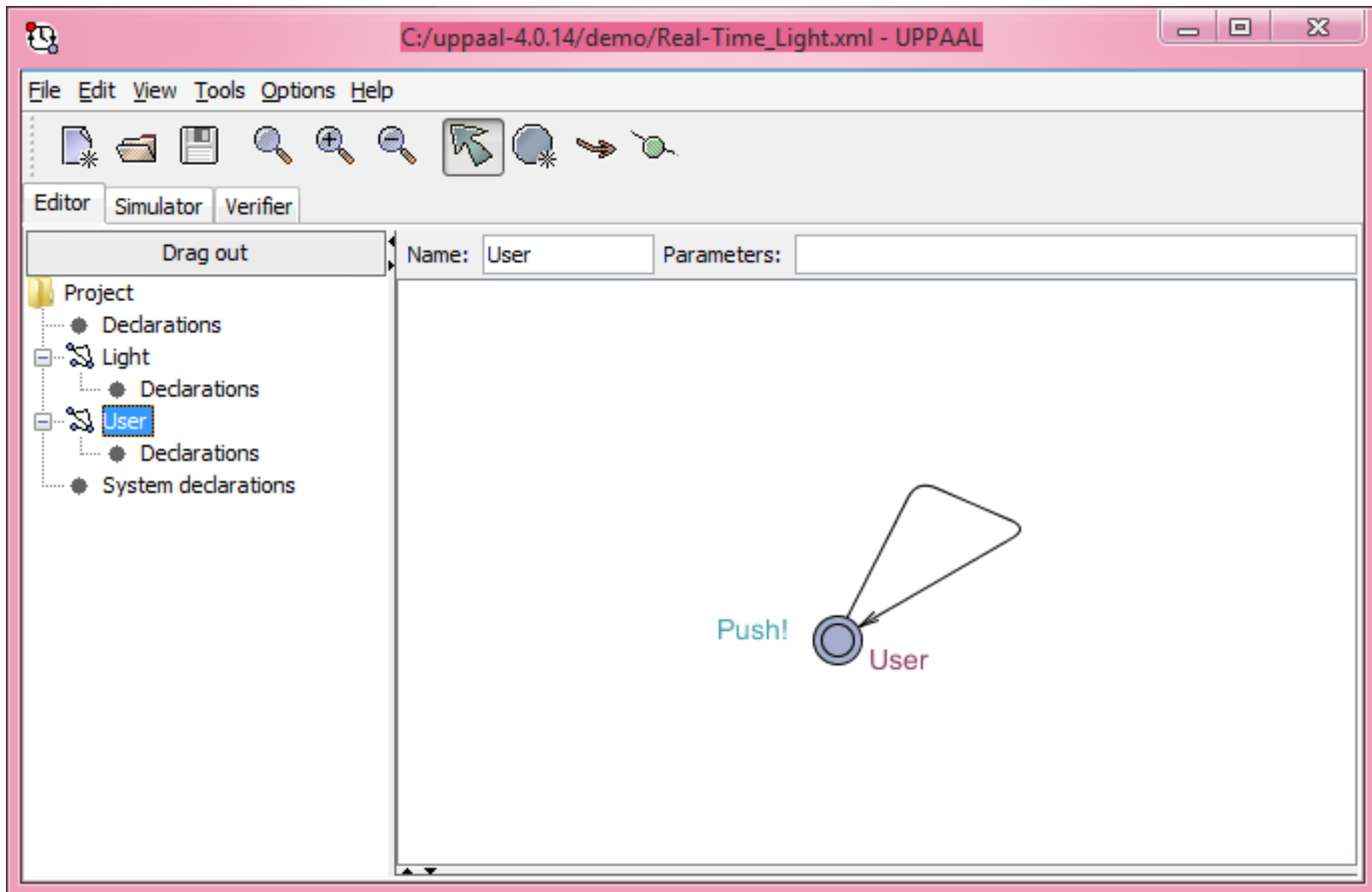
**Solution:** Add a real-valued clock,  $x$ .

# UPPAAL - Editor (Modeller)





# UPPAAL - Editor (Modeller)



# UPPAAL - Simulator

C:/uppaal-4.0.14/demo/Real-Time\_Light.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

Push: User1 --> Light1

Next Reset

Simulation Trace

(Start, User)  
Push: User1 --> Light1  
(Light, User)  
Push: User1 --> Light1  
(Start, User)

Trace File:

Prev Next Replay  
Open Save Auto

Slow Fast

Drag out

$x > 3$

Light1

Start Light Bright

Push?

$x := 0$

$x \leq 3$

Push?

Push?

User1

Push!

User

Light1 User1

Start User

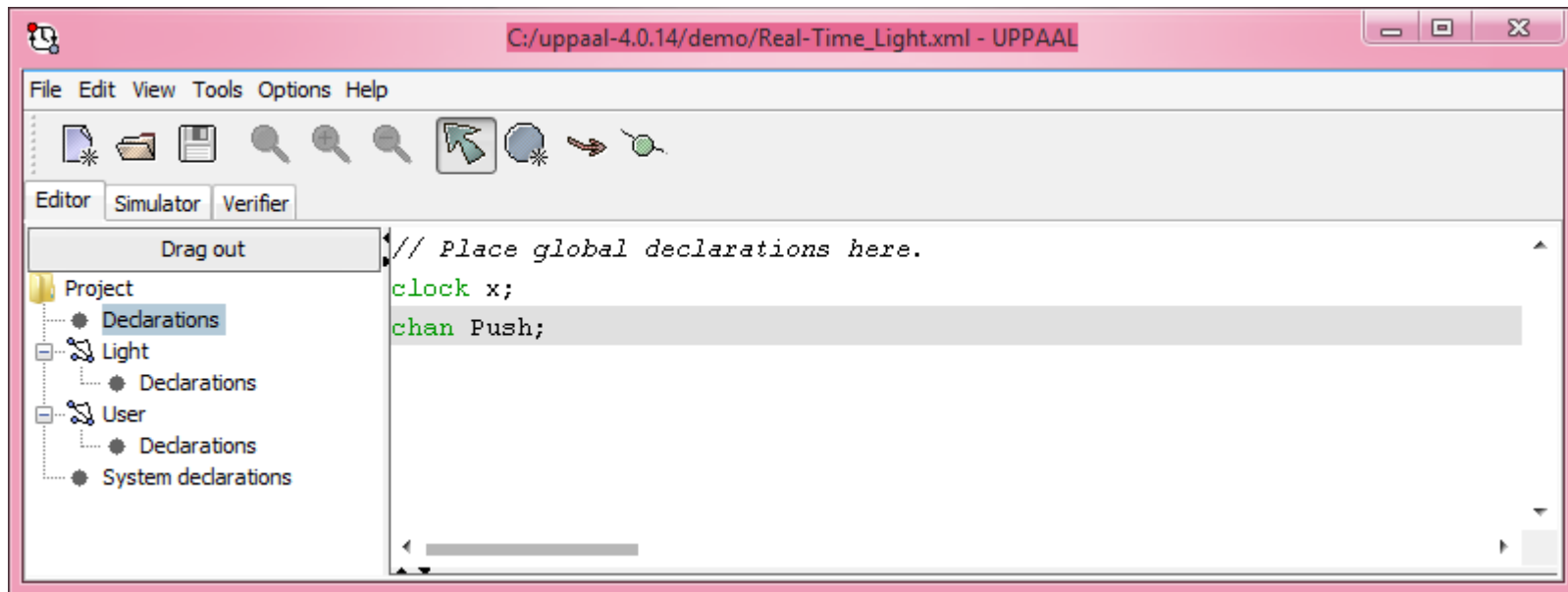
Light User

Start User

Push

Push

# UPPAAL – Global Declarations

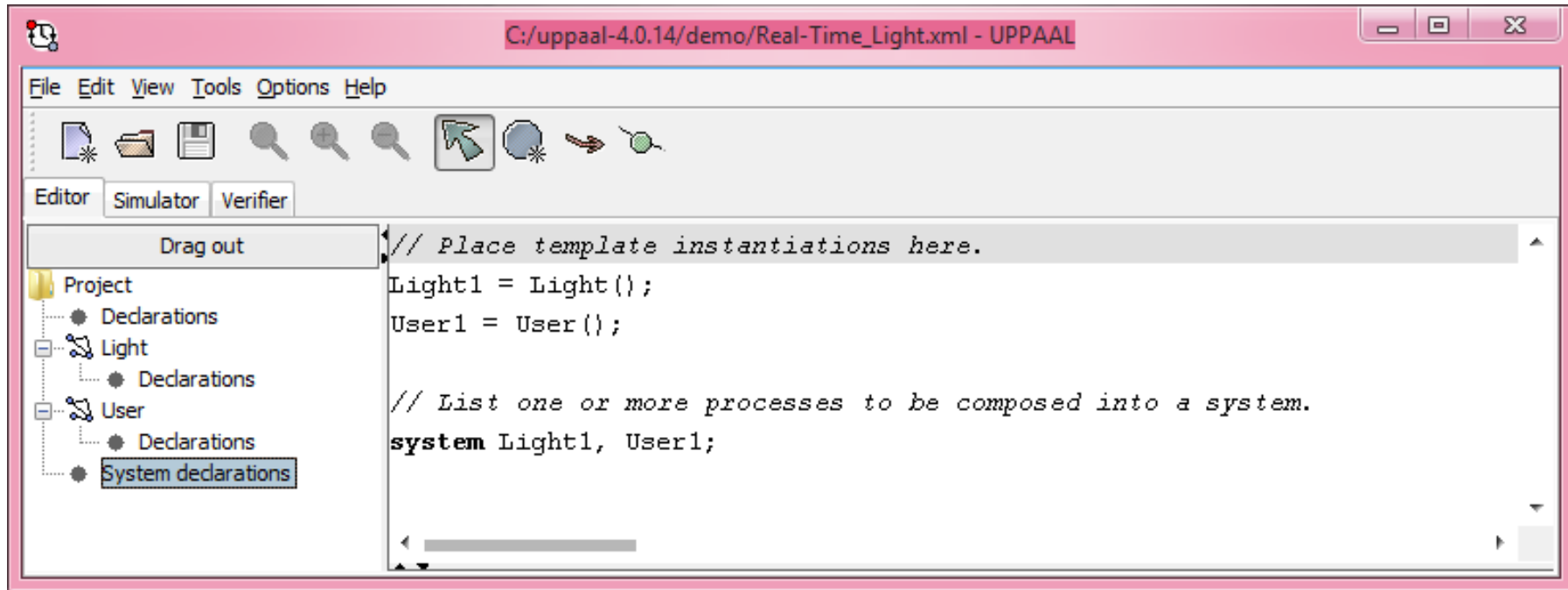


Global variables:

**clock** x;

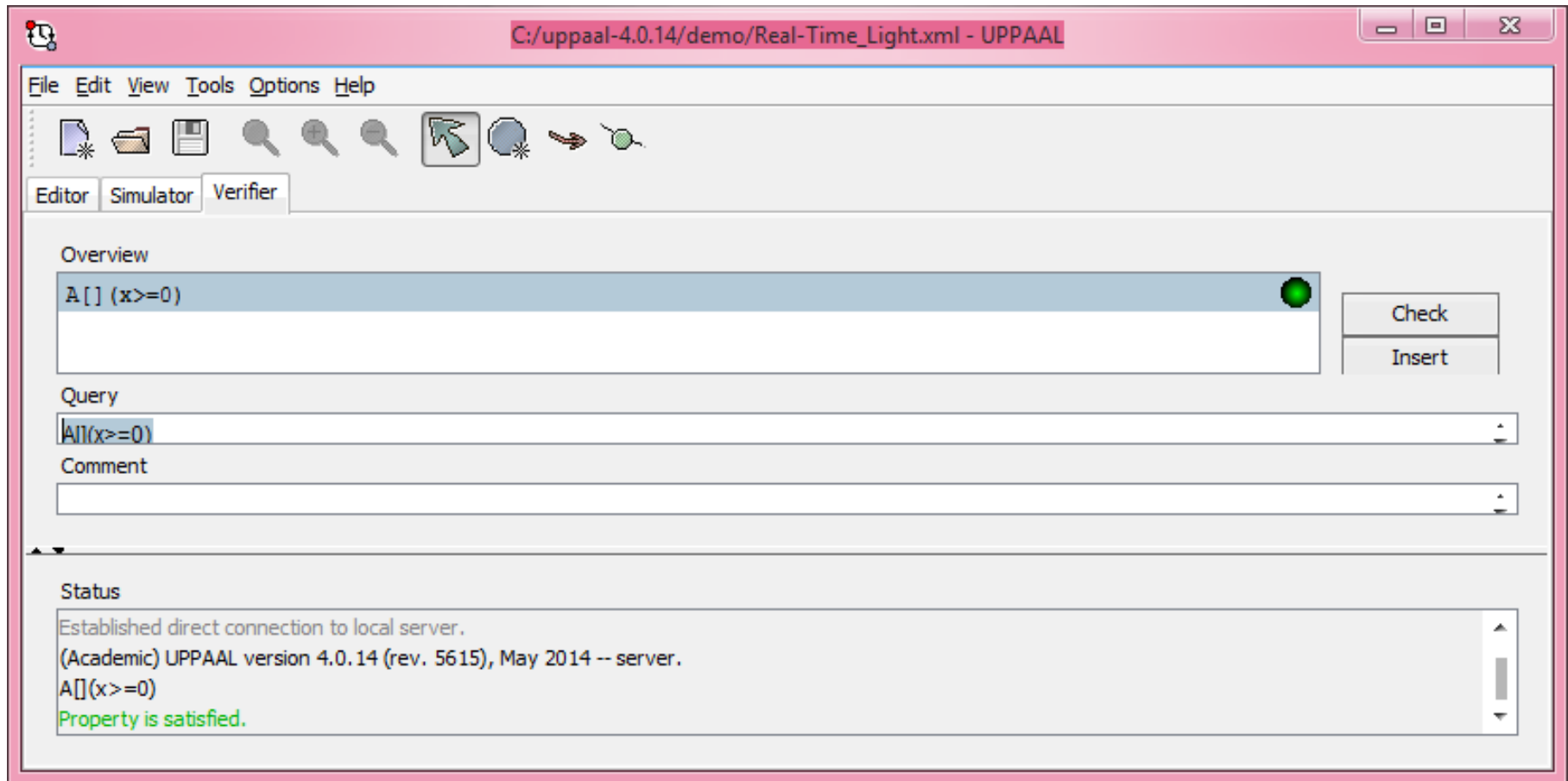
**chan** Puch;

# UPPAAL – Processes and System Definition



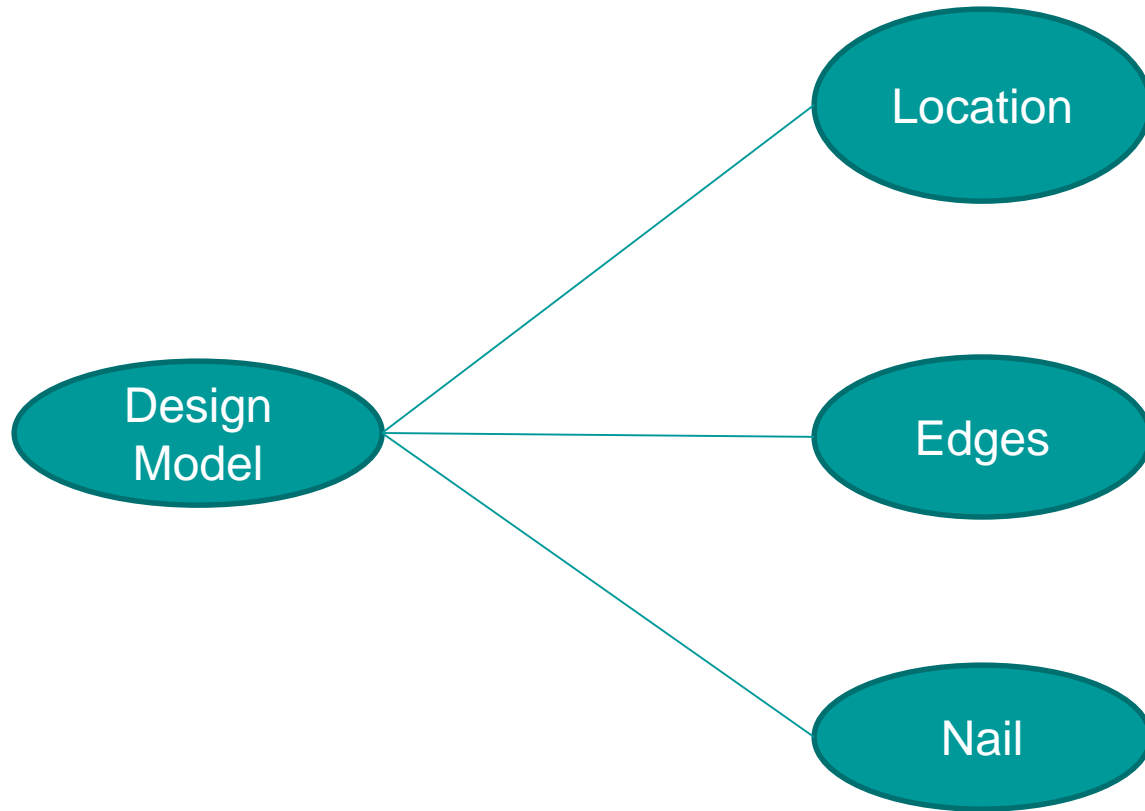
**Light** and **User** are templates (types) of processes (*automations*).  
Light1 and User1 are instances of these templates (processes).  
System composed of Light1 and User1 processes

# UPPAAL – Model checker (Verifier)



Example of the property to be checked: `A[](x>=0)`

# UPPALL notation for a model design



---

# Locations (The nodes of an UPPAAL process)

## 1. Initial Locations

The beginning of the process. Each process must have exactly one initial location. The initial location is marked by a double circle.

## 2. Urgent Locations

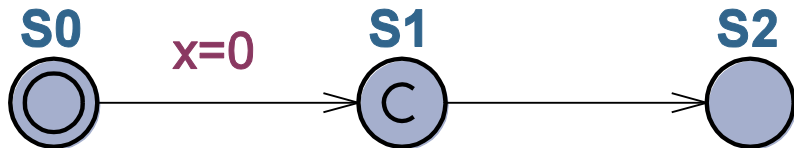
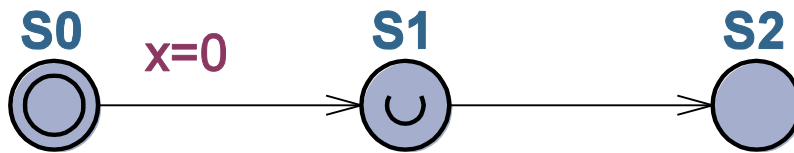
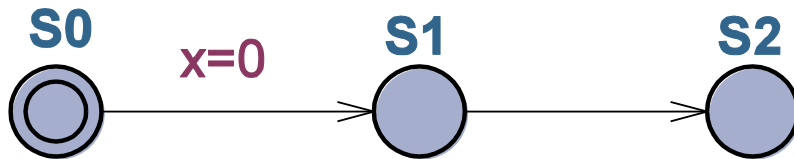
Urgent locations freeze time. The process makes a transition via urgent node without any delay.

## 3. Committed Locations

Like urgent locations, committed locations freeze time. Furthermore, if any process is in a committed location, the next transition must be done from one of the committed locations.

---

# Example





# Locations (nodes)

Locations can have an optional name. It serves as an identifier allowing you to refer to the location from the language. The name must be a valid identifier.

Conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks.

The screenshot shows a 'Location' dialog box. The 'Name' field is set to 'Start'. The 'Invariant' field contains the expression  $x \leq 2$ . The 'Initial' checkbox is checked, while 'Urgent' and 'Committed' are unchecked. The dialog also features 'OK' and 'Cancel' buttons at the bottom.

Exactly one per Template

Freeze time; *i.e.* time is not allowed to pass when a process is in an urgent location.

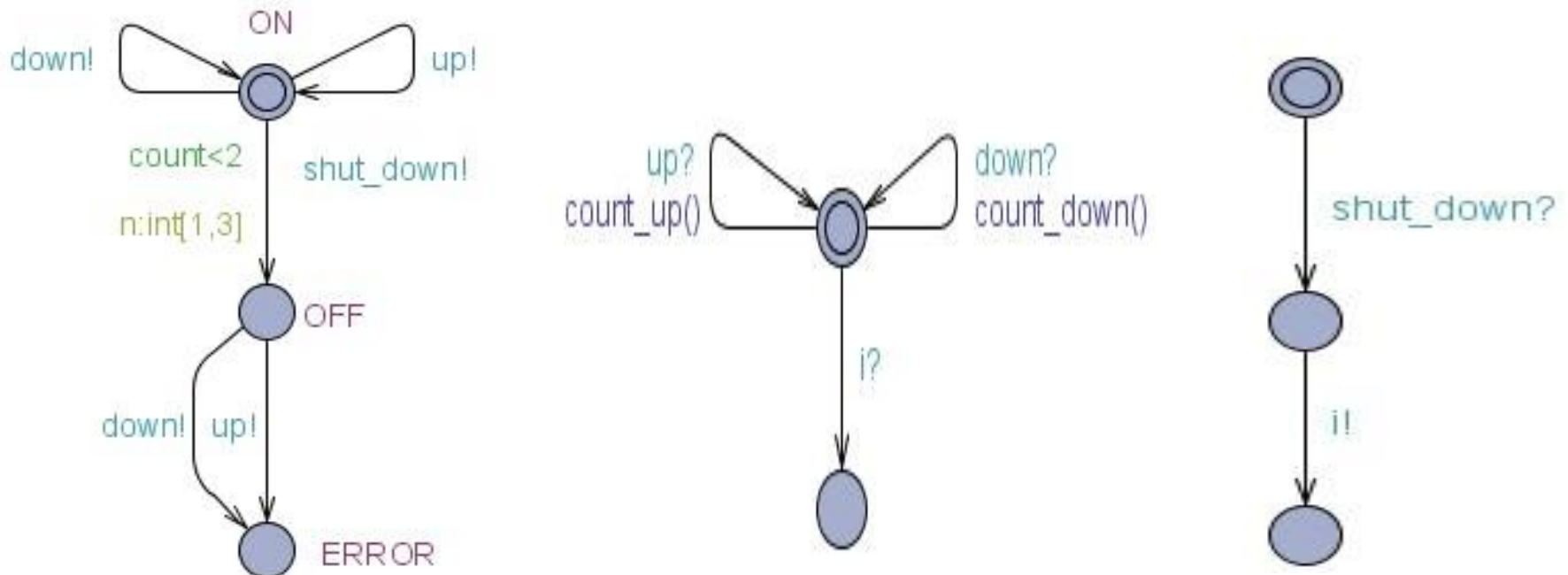
Like urgent locations, committed locations freeze time. Furthermore, if any process is in a committed location, the next transition must involve an edge from one of the committed locations.

# Invariant

- Express condition at location on the value of clocks and integer variables that must be satisfied for the transition to be taken.
- Its evaluates as Boolean value
- Only clock variables, integer variables, constants can be used (or arrays of such)

# Edges

Line between two control nodes (locations).  
Edges are annotated with **selections**, **guards**, **synchronizations** and **updates**.



# Edges – Cont.

## 1. Selections

Selections non-deterministically bind a given identifier to a value in a given range.

## 2. Guards

- ✓ Express condition at edge that must be satisfied for the transition to be taken.
- ✓ It should be correct and side-effect free (i.e. no assignments allowed) Boolean expression
- ✓ Only clock variables, integer variables, constants are allowed (or arrays of such)

---

# Edges - Cont

## 3. Synchronisations

Synchronisation means that two processes make transition (change location) in one step. Synchronisation is done via channels.

To synchronize two processes, the edges should be labeled with the *channel variable* that has been declared before (e.g. **chan** Puch), followed by “!” for one of them and “?” for the other.

## 4. Updates

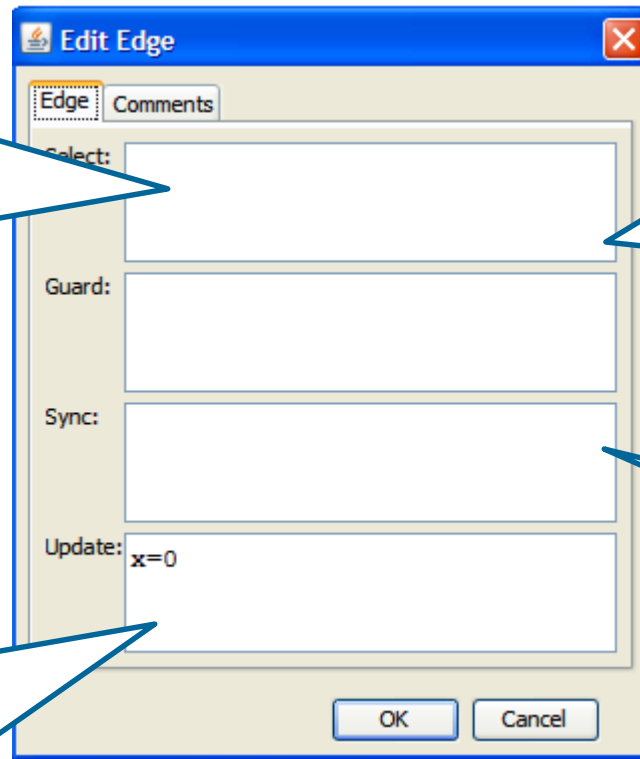
When executed, the update (i.e. assignement) expression of the edge is evaluated. The side effect of this expression changes the state of the system (e.g.  $x = 3$ ;

---

# Edges

Bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding.

When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.



The image shows a software dialog box titled "Edit Edge". It has a blue title bar with a close button (X) in the top right corner. Inside the dialog, there are two tabs: "Edge" (selected) and "Comments". The "Edge" tab contains four input fields with labels to their left: "Select:", "Guard:", "Sync:", and "Update:". The "Update:" field contains the text "x=0". At the bottom of the dialog, there are two buttons: "OK" and "Cancel".

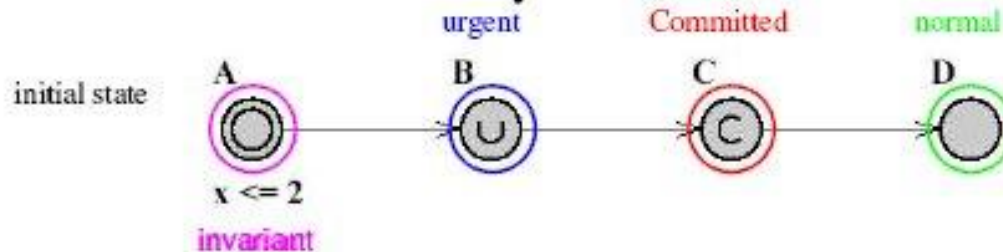
An transition is enabled in a state if and only if the guard is true.

Put here the name of common synchronization channel to synchronize processes.

# UPPAAL Locations (Nodes)

Types of locations:

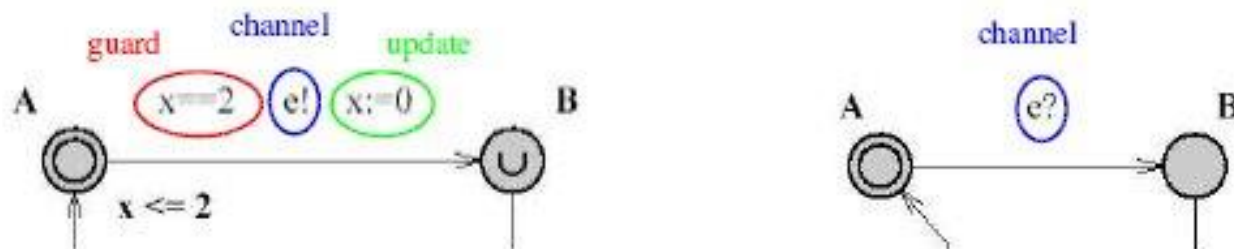
- **normal**
- **normal with invariants**  
invariants are progress conditions expressing constraints on the clock values in order for control to remain in a particular state
- **urgent**  
time may not pass, but interleavings are allowed
- **committed**  
has to be left immediately



# UPPAAL Transitions

The transitions of the automata can be labeled with three different types of labels:

- **guard**  
expressing a condition on the values of clocks and integer variables that must be satisfied in order for the transition to be enabled
- **synchronization channel**  
hand-shaking synchronization, urgent, broadcast
- **update**  
number of clock resets and assignments to integer variables

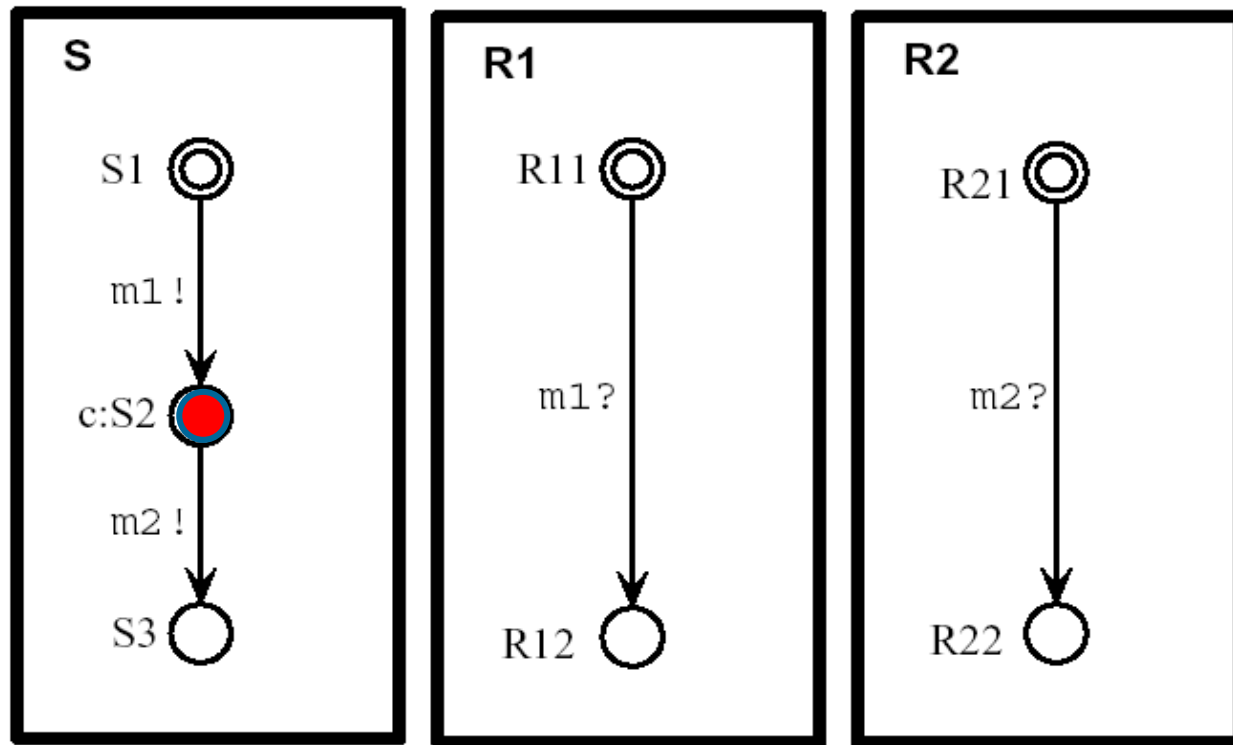




# Broadcast channel

- Broadcast channels allow 1-to-many synchronisations.
- An edge with synchronisation label **e!** emits a broadcast on the channel **e** and that any enabled edge with synchronisation label **e?** will synchronise with the emitting process.

# Committed Locations and Broadcast



A **committed location** must be left immediately.

A broadcast can be represented by two transitions with a committed state between sends.

# Variable

Generally, there are 4 variable types in Uppaal

- Integer
  - Bool
  - Clock
  - Channel
- Same types with C and C++
- New types in Uppaal

# Integer

- The range is -32768....32767

# Boolean

- There are 2 values of Bool Variable:
  - True
  - False

---

# Clocks

- Used to schedule processes.
  - Declared with the keyword **clock**.
-

# Channel

- Used to synchronize two processes.
- Declared with the keyword **chan** (e.g. **chan Push**).
- Done by annotating edges in the model with special synchronization labels (e.g. **Push!** and **Push?**).
- There is special type of chan – **urgent channel**

# Urgent Channels

- When a channel is declared as an Urgent Channel, synchronization via that channel has priority over normal channels and the transition must be taken without delay.
- Declared with the keyword **urgent chan**.
- No clock guard allowed on transitions with urgent actions.
- Invariants and data-variable guards are allowed.



# Declaration in Uppaal

The syntax used for declarations in UPPAAL is similar to the syntax used in the C programming language.

## Integer

- **int num1, num2;**  
two integer variables “num1” and “num2” with default domain.
- **int[0,100] a;**  
an integer variable “a” with the range 0 to 100.
- **int a[2][3];**  
a multidimensional integer array.
- **int[0,5] b=0;**  
an integer variable with the range 0 to 5 initialized to 0.

---

# Declaration in Uppaal - Cont

## Boolean

- **bool yes = true;**  
a boolean variable “yes” initialize to true.
- **bool b[8], c[4];**  
two boolean arrays b and c, with 8 and 4 elements respectively.

## Const

- **const int a = 1;**  
constant “a” with value 1 of type integer.
  - **const bool No = false;**  
constant “No” with value false of type boolean.
-

---

# Declaration in Uppaal - Cont

## Clock

- **clock x, y;**  
two clocks x and y.

## Channel

- **chan d;**  
a channel.
- **urgent chan a, b ,c;**  
urgent channel.

# Propositional logic in UPPAAL

- and
- or
- not
- imply

There is special keyword – **deadlock**

**Note.** UPPAAL is case sensitive.

---

# Linear Temporal Logic (LTL)

- LTL formulae are used to specify temporal properties.
- LTL includes both propositional logic and temporal operators:
  - **[ ]P** = always P
  - **<>P** = eventually P
  - **P U Q** = P is true until Q becomes true

# LTL in UPPAAL

- **E** - exists a path ( “**E**” in UPPAAL).
- **A** - for all paths ( “**A**” in UPPAAL).
- **[]** – all states in a path
- **<>** - some states in a path
- **p --> q** - leads to

The following combination are supported:

- **A[ ], A<>, E<>, E[ ]**

# Examples

// Possible, that automation **Obs** is in the state **Idle** and **x>3**  
**E<> Obs.Idle and x>3**

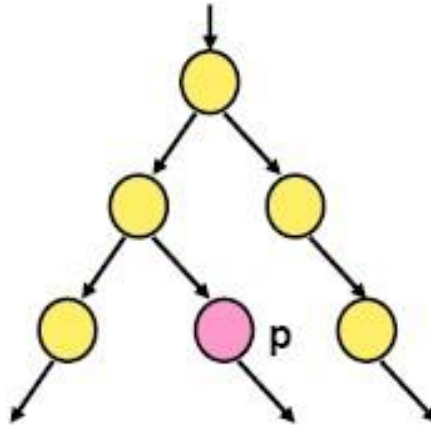
// In all states **x>=0**  
**A[](x>=0)**

// If **x>3** we never come to the state **Obs.Taken**  
**A[] x>3 imply not Obs.Taken**

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

# $E \langle \rangle p$ – “p Reachable”

$E \langle \rangle p$  – it is possible to reach a state in which  $p$  is satisfied.

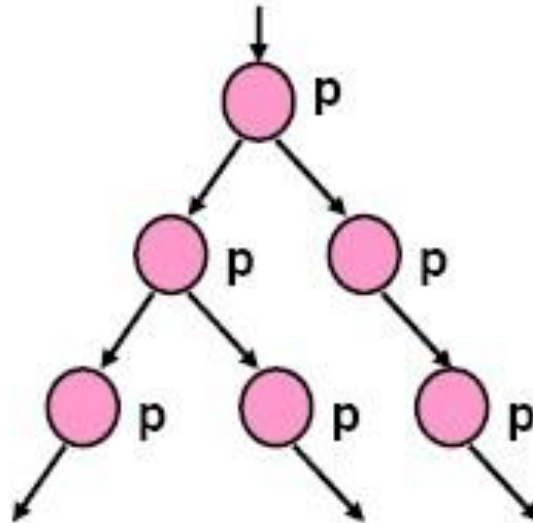


$p$  is true in (at least) one reachable state.



$A[] p$  – “Invariantly  $p$ ”

$A[] p$  –  $p$  holds invariantly.

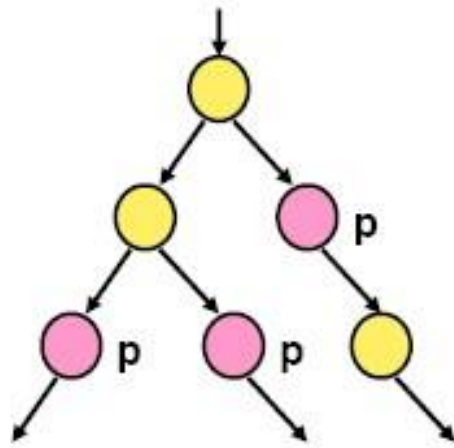


$P$  is true in all reachable states.

# $A \leftrightarrow p$ – “Inevitable $p$ ”

$A \leftrightarrow p$  –  $p$  will inevitably become true

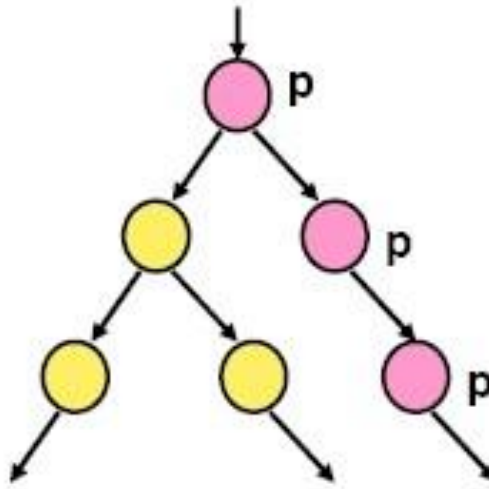
- The automaton is guaranteed to eventually reach a state in which  $p$  is true.



$P$  is true in some state of all paths.

$E[] p$  – “Potentially Always  $p$ ”

$E[] p$  –  $p$  is potentially always true.



There exists a path in which  $p$  is true in all states.

# Verifying Properties

- $E<>p$ : there exists a path where p eventually true. (**Possibly / Reachable**)
- $A[]p$ : for all paths p always true. (**Invariantly**)
- $A<>p$ : for all paths p will eventually hold. (**Inevitable**)
- $E[]p$ : there exists a path where p always hold. (**Potentially Always**)
- $p \rightarrow q$ : whenever p holds q will eventually hold. (**Leads To**)

# A simple program

**Int** x

**Process P**

```
do
  :: x < 2000 → x := x + 1
od
```

**Process Q**

```
do
  :: x > 0 → x := x - 1
od
```

**Process R**

```
do
  :: x = 2000 → x := 0
od
```

**fork P; fork Q; fork R**

**What are possible values for x?**

**Questions/Properties:**

$E \diamond (x > 1000)$

$E \diamond (x > 2000)$

$A[] (x \leq 2000)$

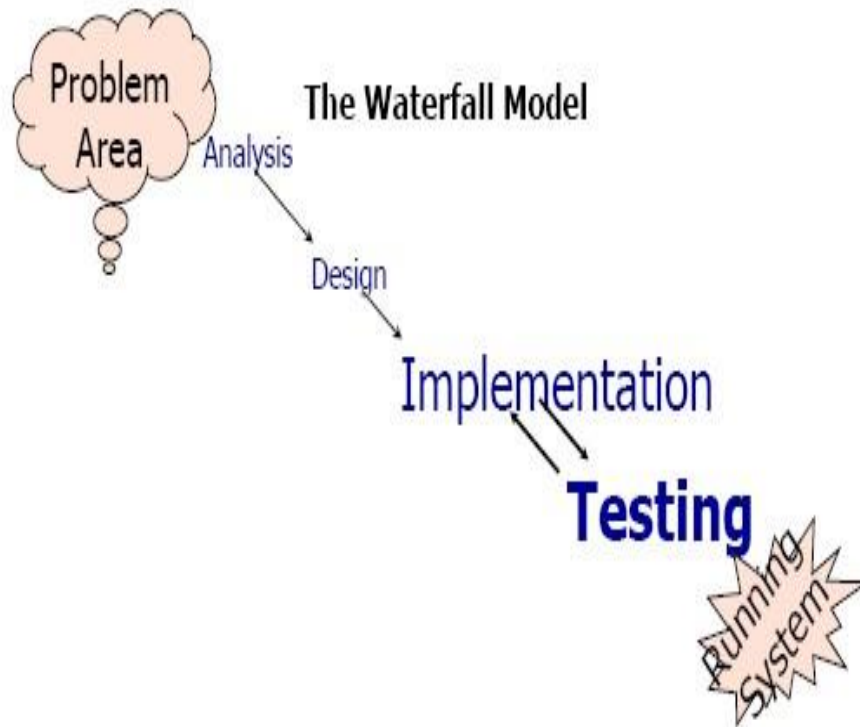
$E \diamond (x < 0)$

**Possible**  $A[] (x \geq 0)$

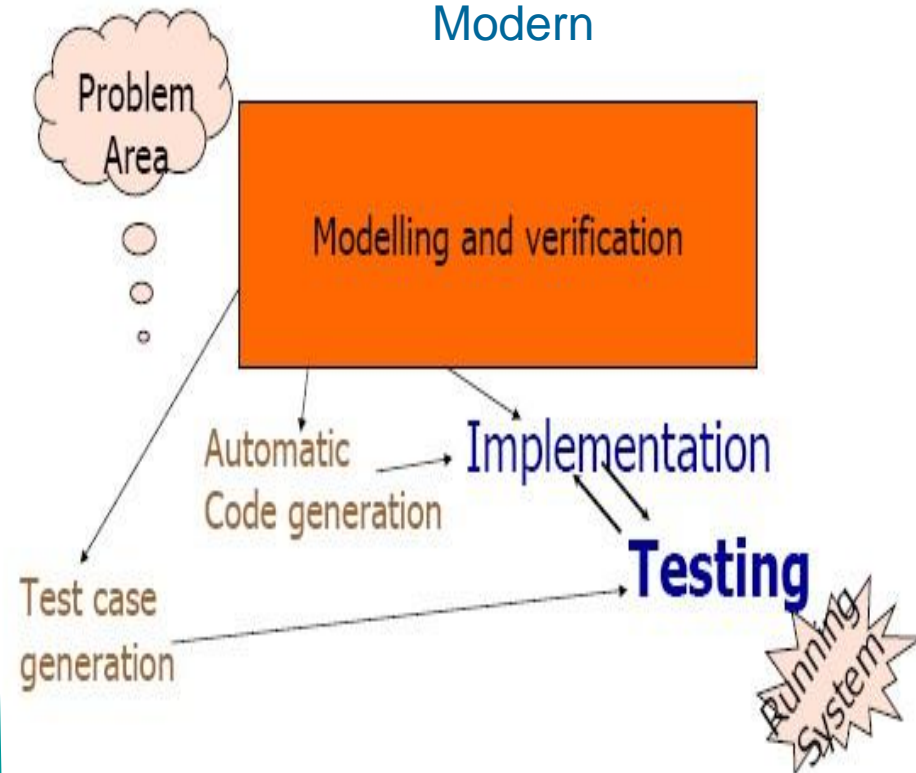
**Always**

# Different Software Development

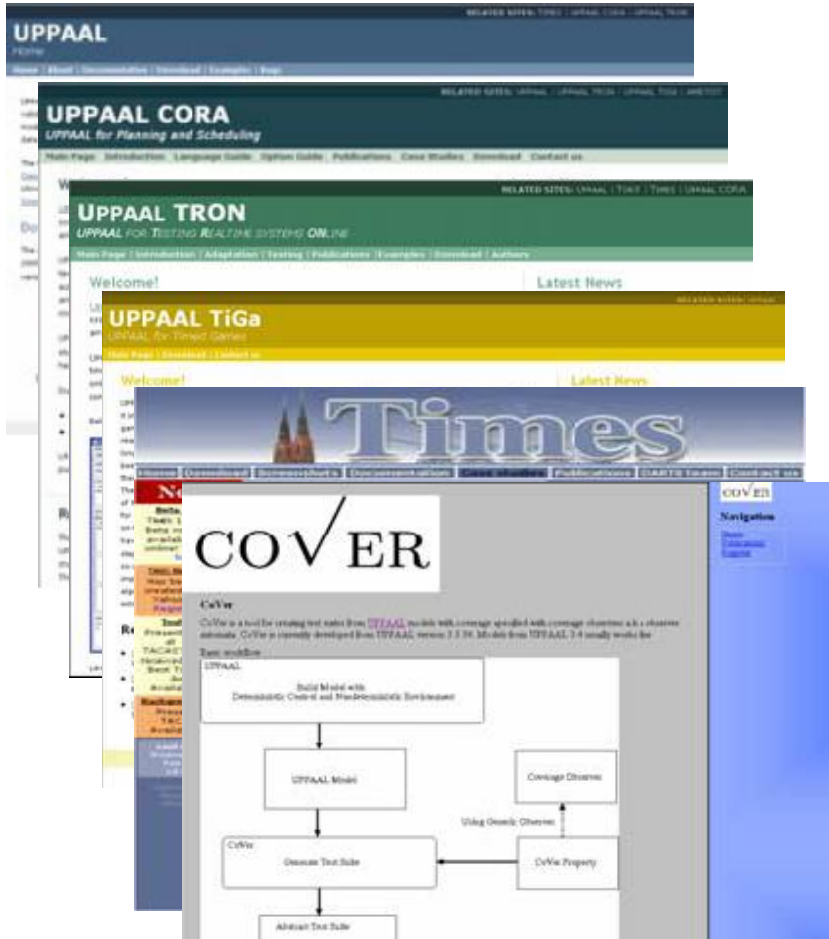
## Traditional



## Modern



# UPPAAL Family



- “Classic”: real-time verification
- Cora : real-time scheduling
- Tron: online real-time testing
- TiGa: timed game
- Times: schedulability analysis
- CoVer: test case generation

---

# Terima Kasih

---