UPPAAL tutorial

- What's inside UPPAAL
- The UPPAAL input languages
  (i.e. TA and TCTL in UPPAAL)

- Train Gate Example;
- A[] Gate.list[N] == 0

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

- ➔ What does it mean????

UPPAAL

- *Uppaal* is a toolbox for modeling, simulating and verifying real-time systems
- It is jointly developed by **Upp**sala University (Sweden) and **Aal**borg University (Denmark)
- The tool is designed for real-time systems that can be modeled as networks of *timed automata*
- 3 components:
  - System editor: graphical user interface to build a system
  - Simulation: step by step movement through system
  - Verification: model checker evaluates questions in temporal logic

Timed Automata with Invariants

- urgent action channels,
- urgent and committed locations,
- data-variables (with bounded domains),
- arrays of data-variables,
- constants,
- guards and assignments over data-variables and arrays…,
- templates with local clocks, data-variables, and constants.
- C subset

Declarations in UPPAAL (1/4)

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

- **Clocks**:
  - **Syntax:**
  - `clock x1, …, xn ;`
  - **Example:**
  - `clock x, y;` **Declares two clocks: x and y.**

Declarations in UPPAAL (2/4)

- **Data variables**
  - Syntax:

▪ `int n1, … ;`	Integer with "default" domain.
▪ `int[l,u] n1, … ;`	Integer with domain "l" to "u".
▪ `int n1[m], … ;`	Integer array w. elements n1[0] to n1[m-1].

  - Example;
  - `int a, b;`
  - `int[0,1] a, b[5][6];`

Declarations in UPPAAL (3/4)

- Actions (or channels):
  - Syntax:

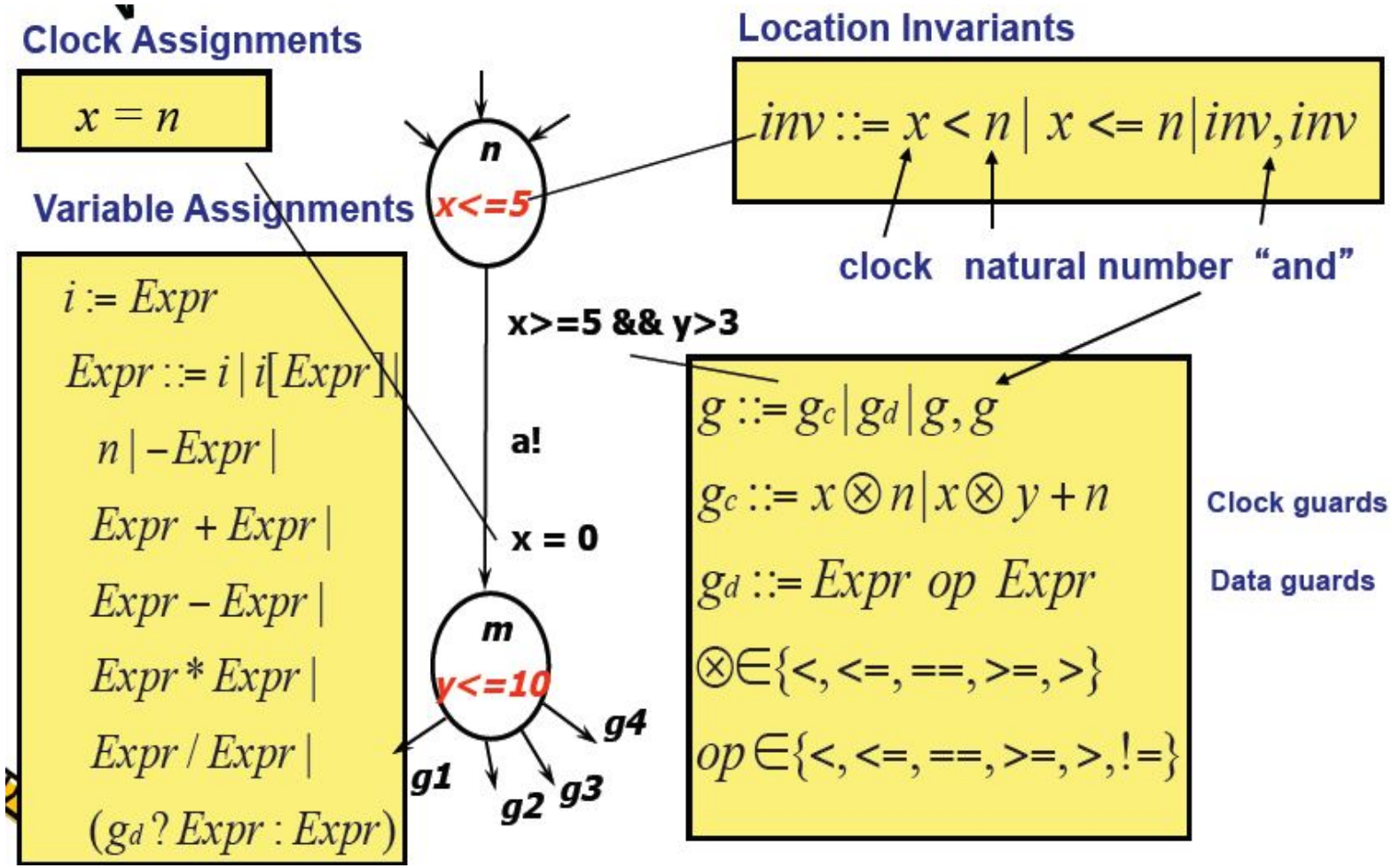
▪ `chan a, … ;`	Ordinary channels.
▪ `urgent chan b, … ;`	Urgent actions (see later)

  - Example:
  - `chan a, b;`
  - `urgent chan c;`

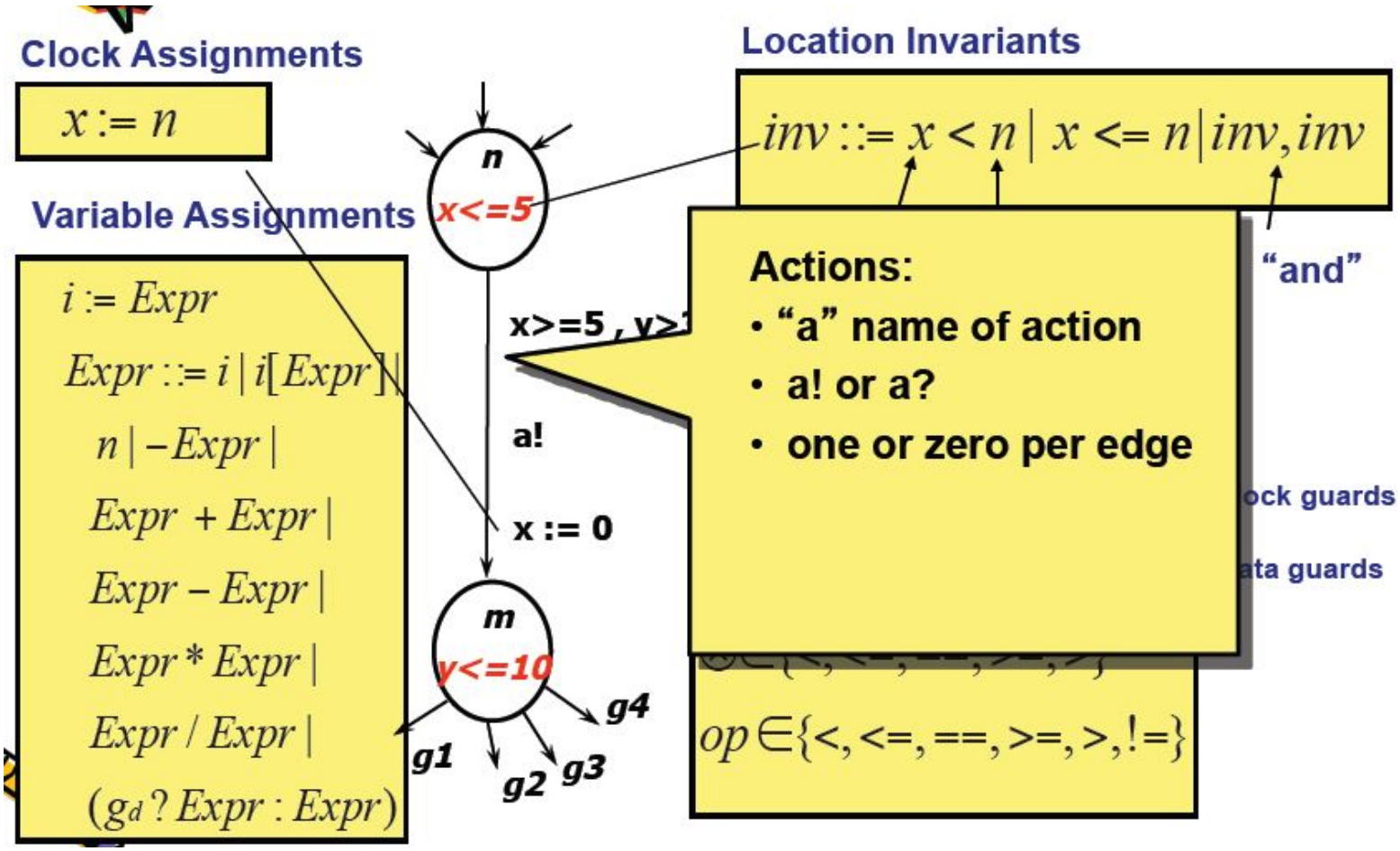
Declarations in UPPAAL (4/4)

- Constants
  - **Syntax:**
  - `const int c1 = n1;`
  - **Example:**
  - `const int[0,1] YES = 1;`
  - `const bool NO = false;`

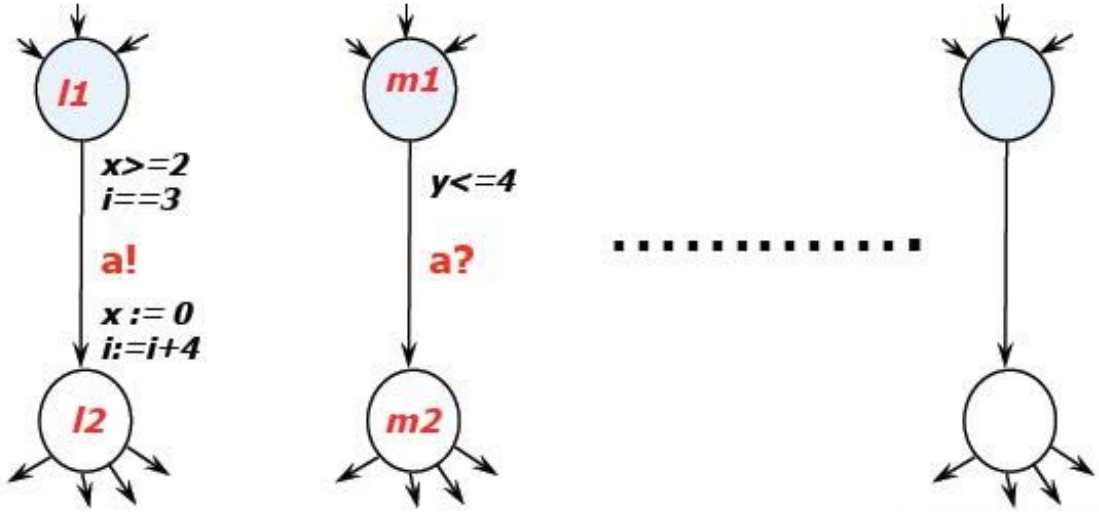
TA in UPPAAL


Clock Assignments
$x = n$
Variable Assignments
$i := Expr$
$Expr ::= i \mid i[Expr] \mid$
$n \mid -Expr \mid$
$Expr + Expr \mid$
$Expr - Expr \mid$
$Expr * Expr \mid$
$Expr / Expr \mid$
$(g_d ? Expr : Expr)$
n
x<=5
x>=5 && y>3
a!
x = 0
m
y<=10
g1
g2
g3
g4
Location Invariants
$inv ::= x < n \mid x <= n \mid inv, inv$
clock
natural number
"and"
$g ::= g_c \mid g_d \mid g, g$
$g_c ::= x \otimes n \mid x \otimes y + n$
$g_d ::= Expr \ op \ Expr$
$\otimes \in \{<, <=, ==, >=, >\}$
$op \in \{<, <=, ==, >=, >, !=\}$
Clock guards
Data guards

TA in UPPAAL


Clock Assignments
$x := n$
Variable Assignments
$i := Expr$
$Expr ::= i \mid i[Expr] \mid$
$n \mid -Expr \mid$
$Expr + Expr \mid$
$Expr - Expr \mid$
$Expr * Expr \mid$
$Expr / Expr \mid$
$(g_d ? Expr : Expr)$
Location Invariants
$inv ::= x < n \mid x <= n \mid inv, inv$
"and"
n
x<=5
x>=5 , y>1
a!
x := 0
m
y<=10
g1
g2
g3
g4
Actions:
• "a" name of action
• a! or a?
• one or zero per edge
ock guards
ata guards
$op \in \{<, <=, ==, >=, >, !=\}$

Networks of TA



Two-way synchronization on **complementary** actions.

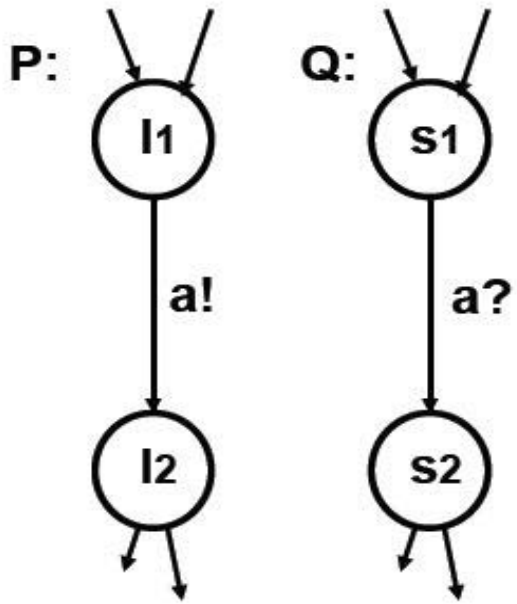
Closed Systems!

Broadcasting Synchronization

- Declared like
  `broadcast chan a, b, c[2];`
- If a is a broadcast channel:
  - `a!` = Emmision of broadcast
  - `a?` = Reception of broadcast

- A set of edges in different processes can synchronize if one is emitting and the others are receiving on the same b.c. channel.
- A process can always emit.
- Receivers *must* synchronize if they can.
- No blocking.

Urgent Channels: Example 1



- Suppose the two edges in automata P and Q should be taken as soon as possible
- I.e. as soon as both automata are ready (simultaneously in locations l_1 and s_1).
- How to model with invariants if either one may reach l_1 or s_1 first?
- **Solution**: declare action "a" as urgent.

Urgent Channels

```
urgent chan hurry;
```

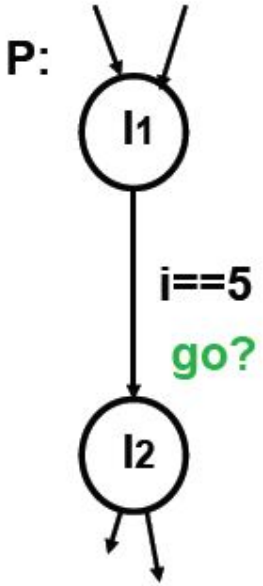
Informal Semantics:

- There will be <u>no delay</u> if transition with urgent action can be taken.

Restrictions:

- <u>No clock guard</u> allowed on transitions with urgent actions.
- <u>Invariants</u> and <u>data-variable guards</u> are allowed.

Urgent channel: Example 2



- Assume i is a data variable.
- We want P to take the transition from l1 to l2 as soon as i==5.
- **Solution**: P can be forced to take transition if we add another automaton:

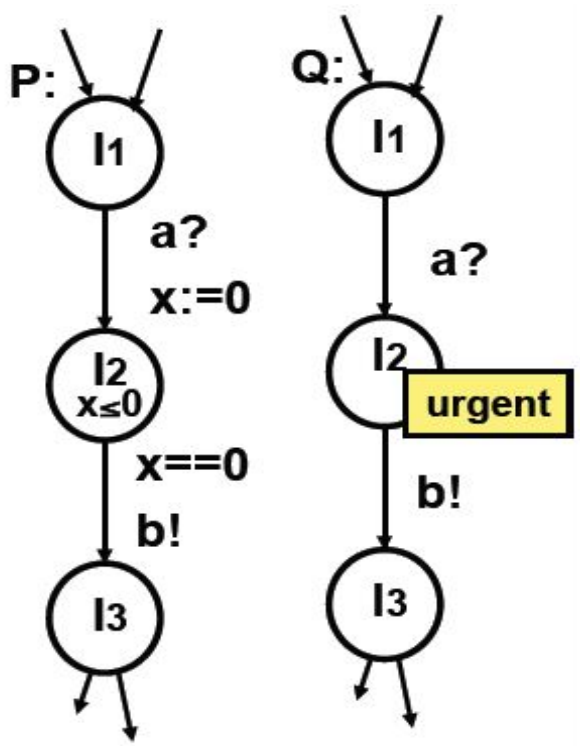
where "go" is an urgent channel, and we add "go?" to transition l1→l2 in automaton P.

Urgent location: Example

- Assume that we model a simple media M:

that receives packages on channel a and immediately sends them on channel b.

- P models the media using clock x.
- Q models the media using **urgent location**.
- P and Q have the same behavior.



Urgent location

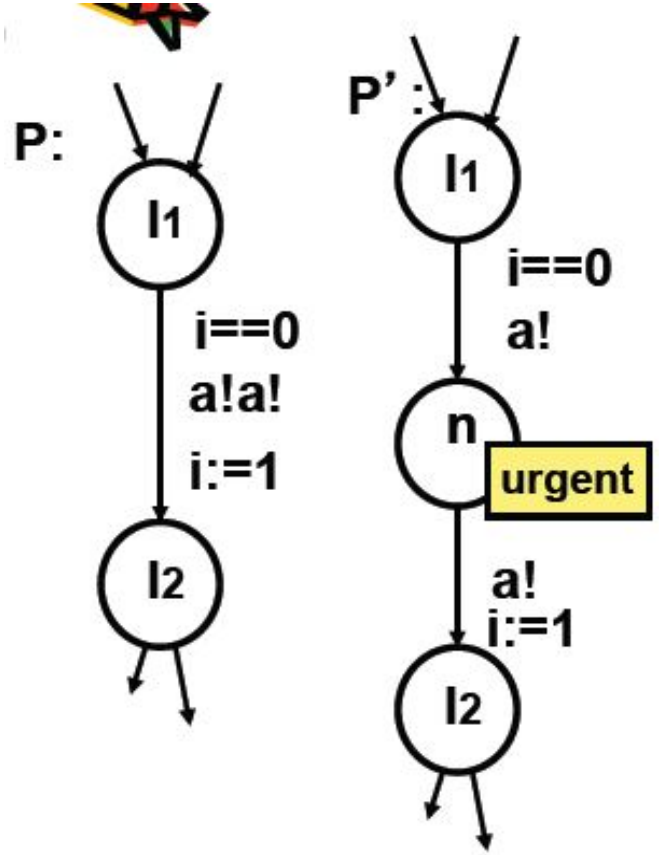
Click "Urgent" in State Editor.

Informal Semantics:

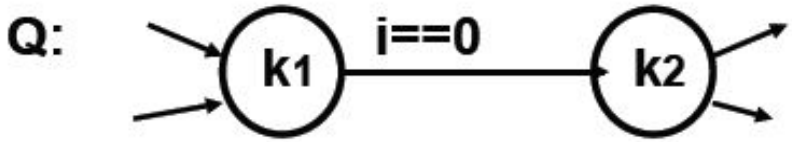
- No delay in urgent location.

Note: the use of urgent locations **reduces** the number of clocks in a model, and thus the complexity of the analysis.

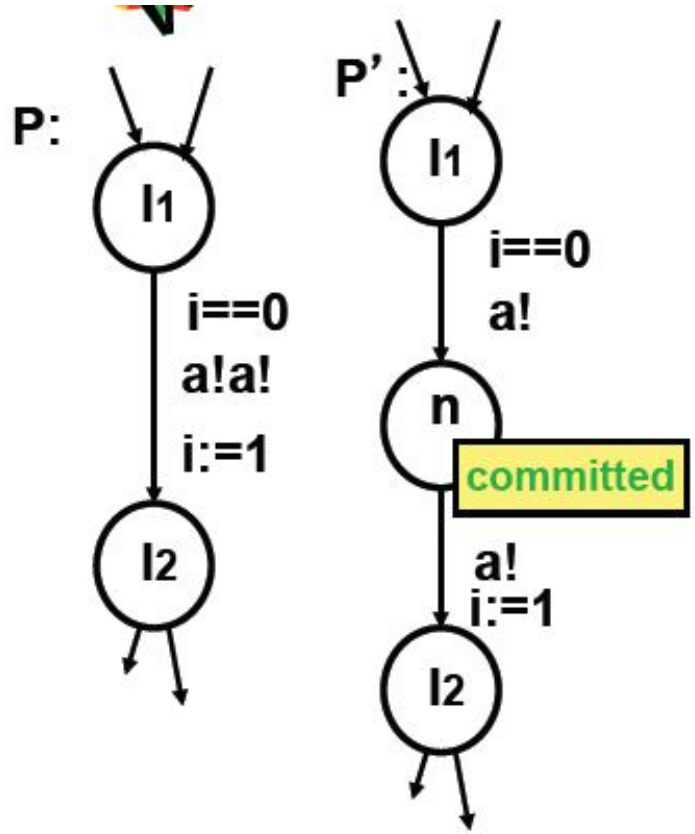
Committed location: Example 1



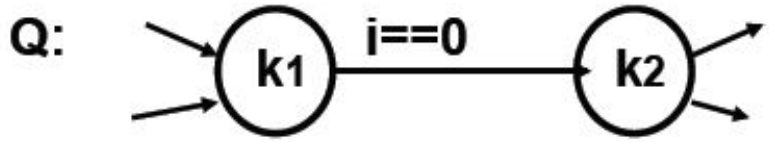
- **Assume:** we want to model a process (P) simultaneously sending message (a) to two receiving processes (when i==0).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interleave (which is wrong):



Committed location: Example 1



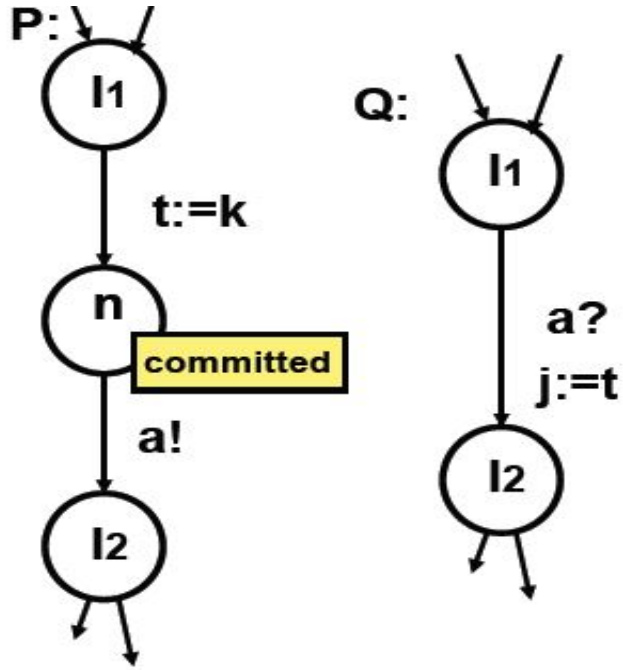
- **Assume:** we want to model a process (P) simultaneously sending message (a) to two receiving processes (when i==0).
- P' sends "a" two times at the same time instant, but in location "n" other automata, e.g. Q may interleave (which is wrong):



- **Solution:** mark location n "committed" in automata P' (instead of "urgent").

Committed location: Example 2

- **Assume:** we want to pass the value of integer "k" from automaton P to variable "j" in Q.
- The value of k can be passed using a global integer variable "t".
- Location "n" is committed to ensure that no other automata can assign "t" before the assignment "j:=t".



Committed location

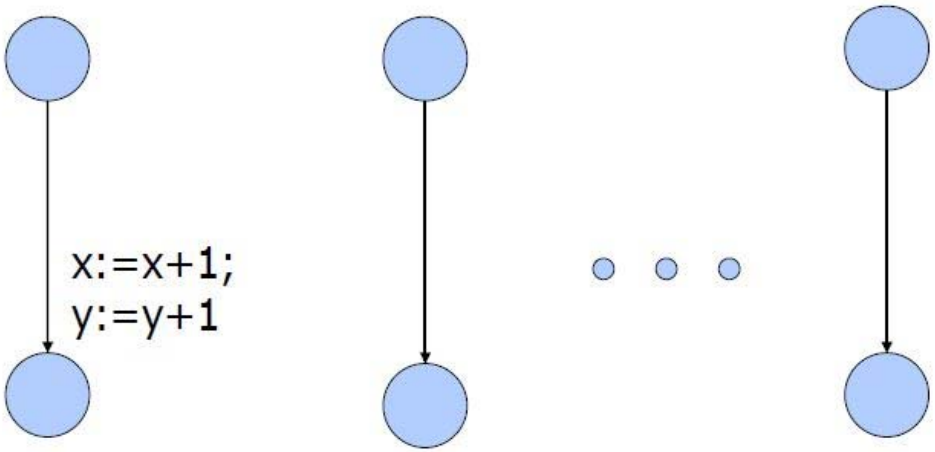
Click "Committed" i State Editor.

Informal Semantics:

- No delay in committed location.
- Next transition must involve automata in committed location.

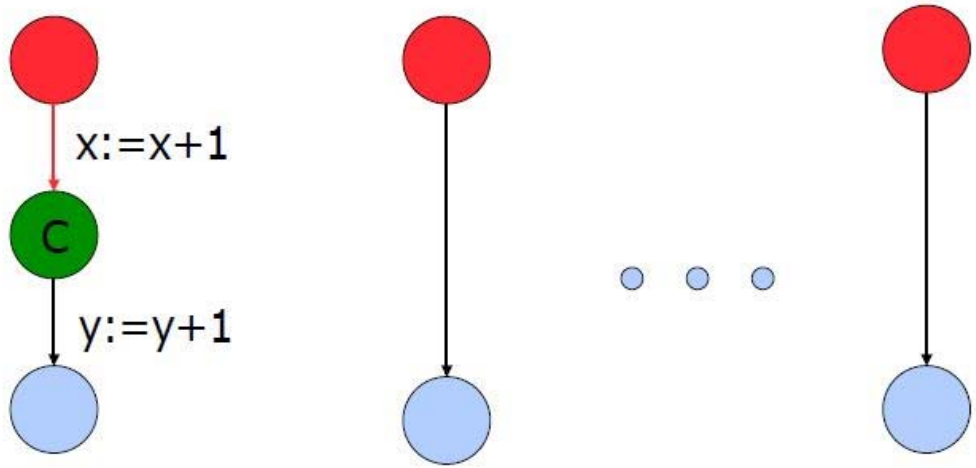
Note: the use of committed locations **reduces** the number of clocks in a model, and allows for more space and time efficient analysis.

Committed location example: (atomic sequence in a network)



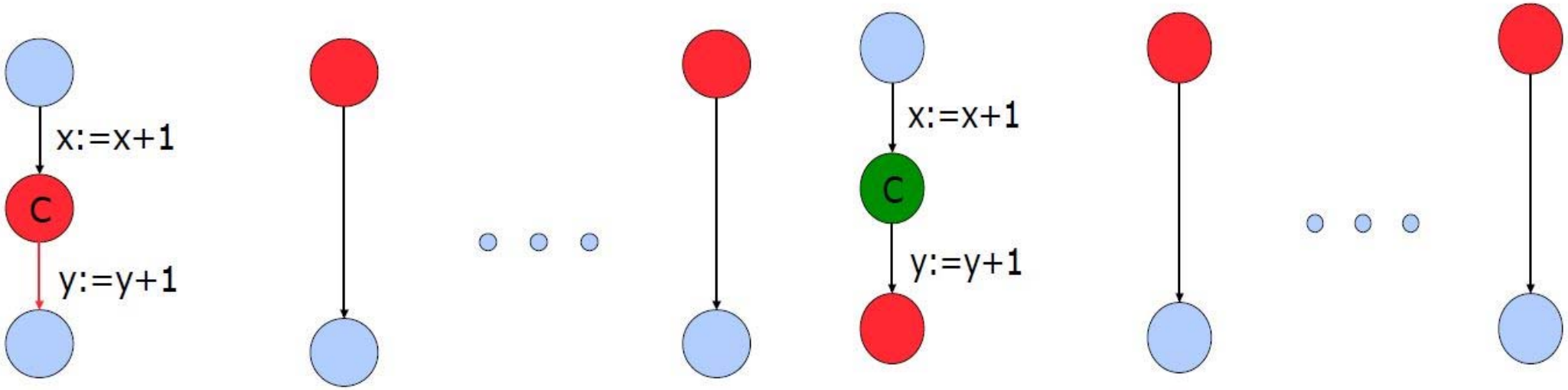
If the sequence becomes too long, you can split it ...

Semantics: the time spent on C-location should be zero !



Committed location example: (atomic sequence in a network)

Semantics: the time spent on C-location should be zero !



Now, only the committed (red) transition can be taken!

Committed location

- A trick of modeling (e.g. to model multi-way synchronization using handshaking)
- More importantly, it is a simple and efficient mechanism for state-space reduction!

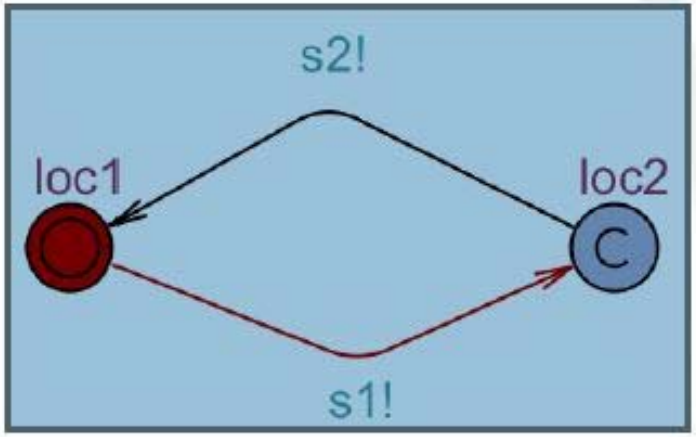
  In fact, it is a simple form of 'partial order reduction'
- It is used to avoid intermediate states, interleavings:

  Committed states are not stored in the passed list

  Interleavings of any state with a committed location will not be explored

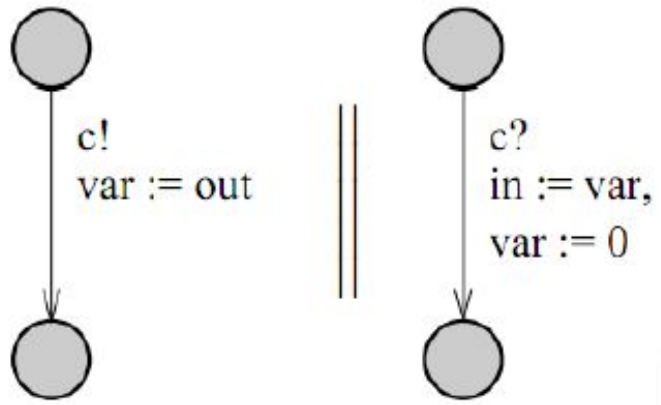
Atomicity

- Some times it is necessary to model atomic behaviors
- How to model atomicity in Uppaal?
  - Committed locations
    - When a committed location is entered the execution flow must continue through such a location



Synchronous value passing: one-way

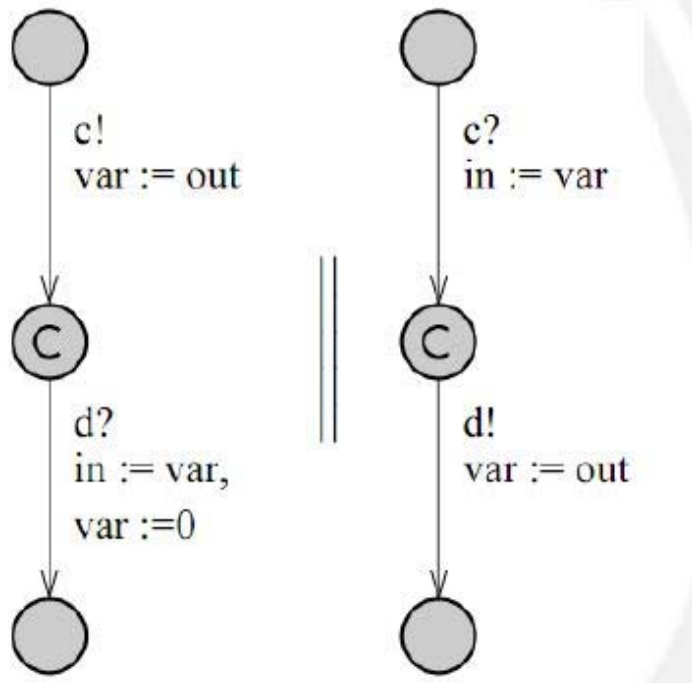
- `c` is a binary channel
- `var` is a shared variable (i.e., global)
- `in` and `out` are local variables
- Resets
  - The resets of the sender is executed before the resets of the receiver
  - Given a list of reset statements, these are executed sequentially



Synchronous value passing: two-way

Synchronous Value Passing: two-way (III)

- `c` is a binary channel
- `var` is a shared variable (i.e., global)
- `in` and `out` are local variables



More expressions and More on types

- Operators (not clocks):
  - Logical:
    - && (logical and), || (logical or), ! (logical negation)
  - Bitwise:
    - ^ (xor), & (bitwise and), | (bitwise or),
  - Bit shift:
    - << (left), >> (right)
  - Numerical:
    - % (modulo), ? (max)
  - Assignments:
    - +=, -=, *=, /=, ^=, <<=, >>=, :=
  - Prefix and postfix:
    - ++ (increment), -- (decrement)
- Multi dimensional arrays
  - e.g. int b[4][2];
- Array initialiser:
  - e.g. int b[4] := { 1, 2, 3, 4 };
- Arrays of channels, clocks, constants.
  - e.g.
  - chan a[3];
  - clock c[3];
  - const k[3] { 1, 2, 3 };
- Broadcast channels.
  - e.g. broadcast chan a;

Extensions

Select statement

- Models non-deterministic choise
- `x : int[0,42]`

Types

- Record types
- Type declarations
- Meta variables:
  not stored with state
  `meta int x;`

Forall / Exists Expressions

- `forall (x:int[0,42]) expr`
  true if expr is true for *all* values in [0,42] of x

- `exists (x:int[0,4]) expr`
  true if expr is true for *some* values in [0,42] of x

Example:

```
forall
(x:int[0,4])array[x];
```

Check UPPAAL Tutorial: Gossip girl example

UPPAAL specification language

- The main purpose of a model checker is verify the model w.r.t. a requirement specification
- Uppaal uses a simplified version of CTL for defining the specifications
- The query language consists of
  - State formulae
    - Describe individual states
  - Path formulae
    - Quantify over paths of the model
      - Reachability
      - Safety
      - Liveness

UPPAAL specification language: TCTL quantifiers in UPPAAL

- E - exists a path ("**E**" in UPPAAL).
- A - for all paths ("**A**" in UPPAAL).
- G - all states in a path ("**[]**" in UPPAAL).
- F - some state in a path ("**<>**" in UPPAAL).

You may write the following queries in UPPAAL:

- **A[]p, A<>p, E<>p, E[]p and p --> q**

AG p

AF p

EF p

EG p

p and q are "local properties"

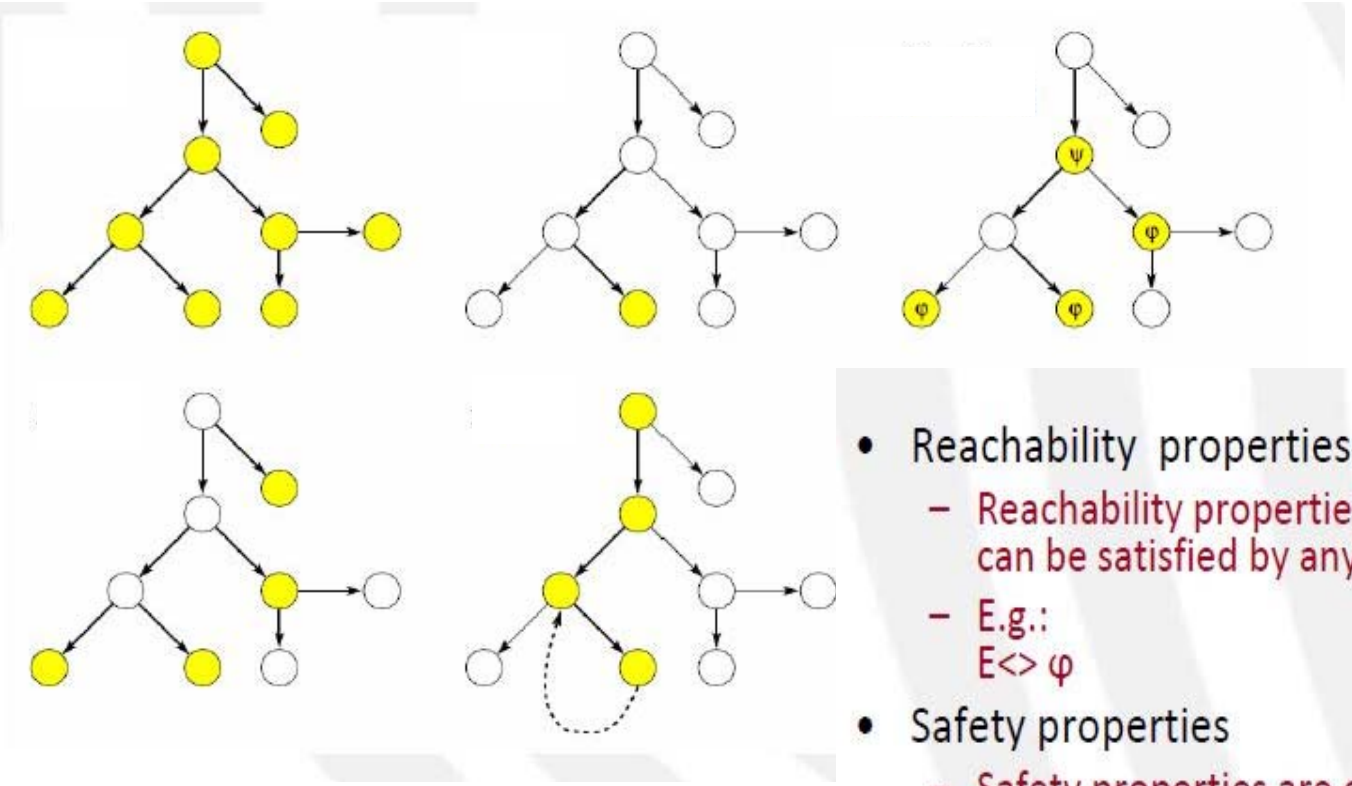
State Formulae

- A *state formula* is an expression that can be evaluated for a single state

  SF ::= `Proc.loc` | `deadlock` |
  `x == n` | `x<=n` | `x < n` | `x > n` | `x >= n` |
  SF `and` *SF* | *SF* `or` *SF* |
  SF `imply` *SF* | `not` *SF*

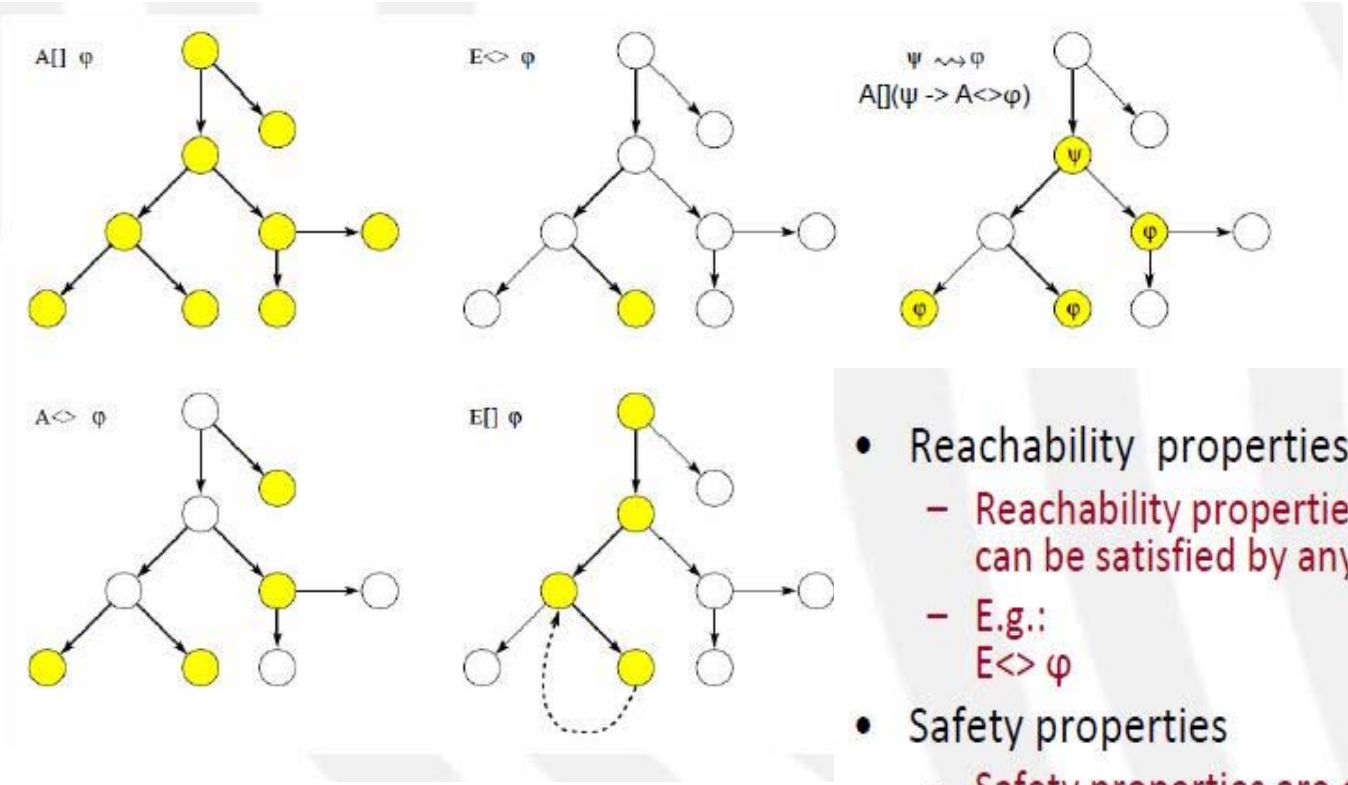
- where
  - `x` is a clock or a discrete variable
  - `n` is an integer

Path formulae



- Reachability properties
  - Reachability properties ask whether a given state formula φ *possibly* can be satisfied by any reachable state
  - E.g.:
    E<> φ
- Safety properties
  - Safety properties are on the form: "something bad will never happen"
  - E.g.:
    A[] φ , E[] φ
- Liveness properties
  - Liveness properties are on the form: "something will eventually happen"
  - E.g.:
    A<> φ , φ --> ψ

Path formulae

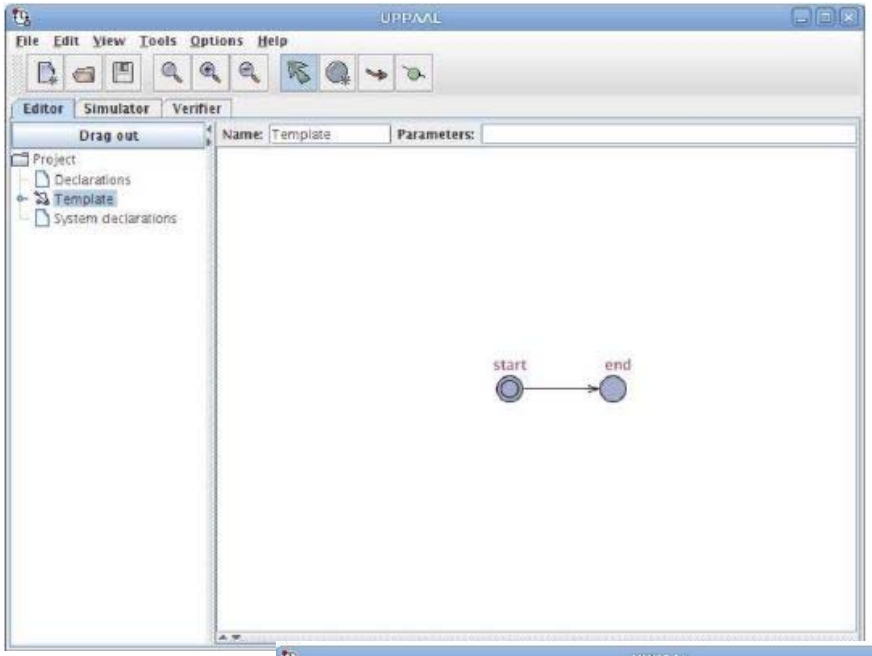


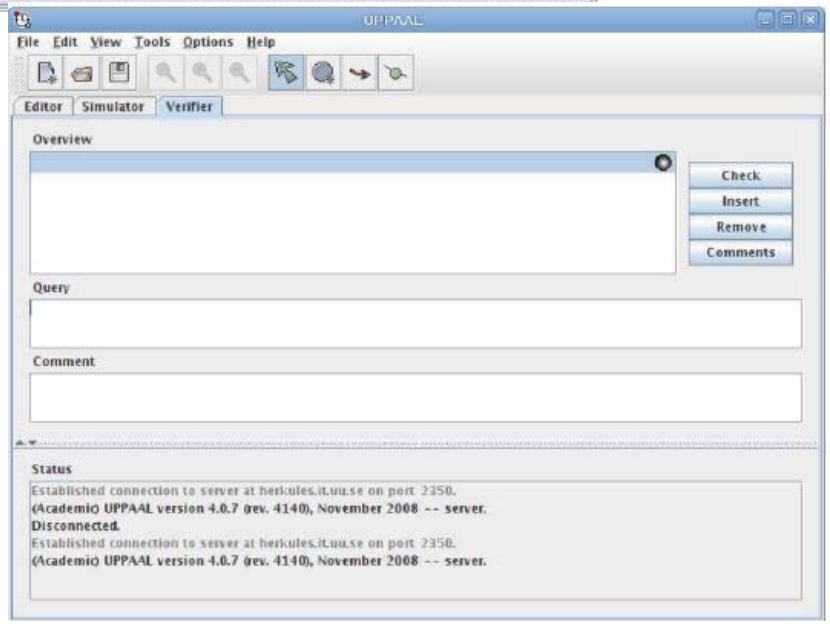
- Reachability properties
  - Reachability properties ask whether a given state formula φ *possibly* can be satisfied by any reachable state
  - E.g.:
    E<> φ
- Safety properties
  - Safety properties are on the form: "something bad will never happen"
  - E.g.:
    A[] φ , E[] φ
- Liveness properties
  - Liveness properties are on the form: "something will eventually happen"
  - E.g.:
    A<> φ , φ --> ψ

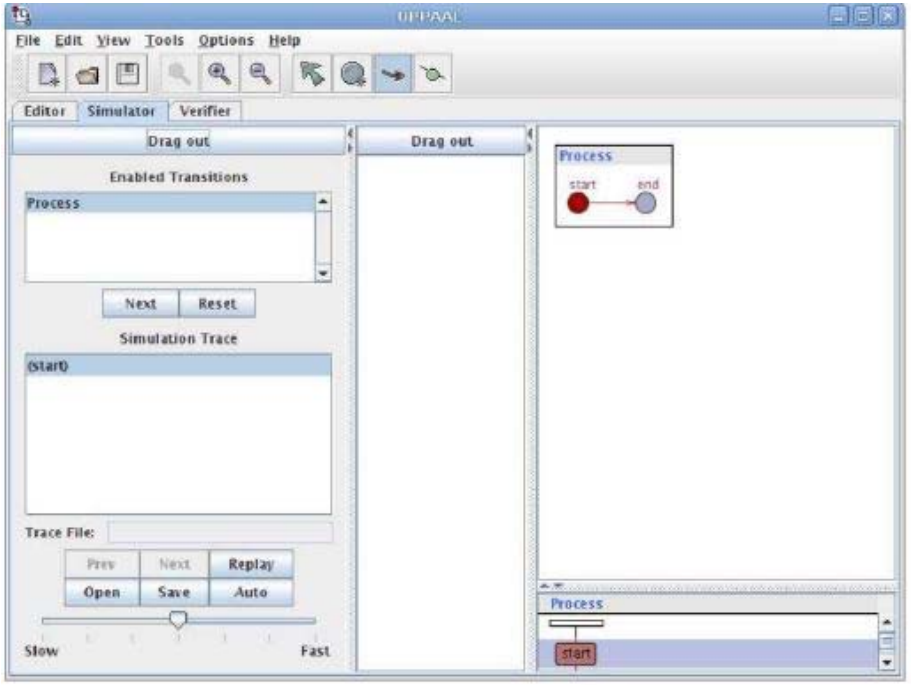
Queries Examples

- A deadlock never occurs
  - A[] not deadlock
- An automaton *A1* remains into a state *q* for at least 10 seconds
  - E<> A1.q and x > 10
- An automaton *A2* may never enter a state *q*
  - E[] not A2.q

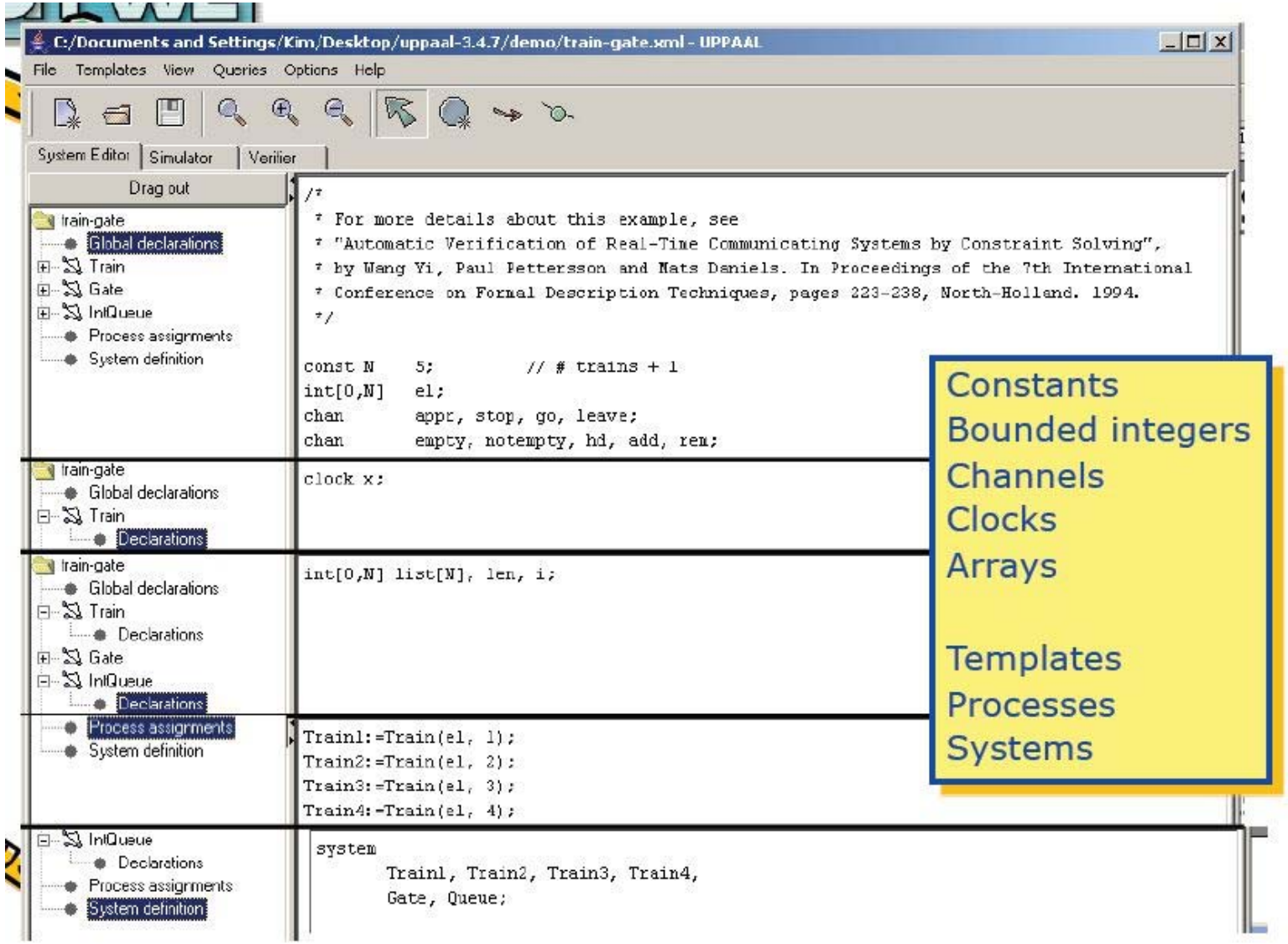
Overview of the UPPAAL toolkit



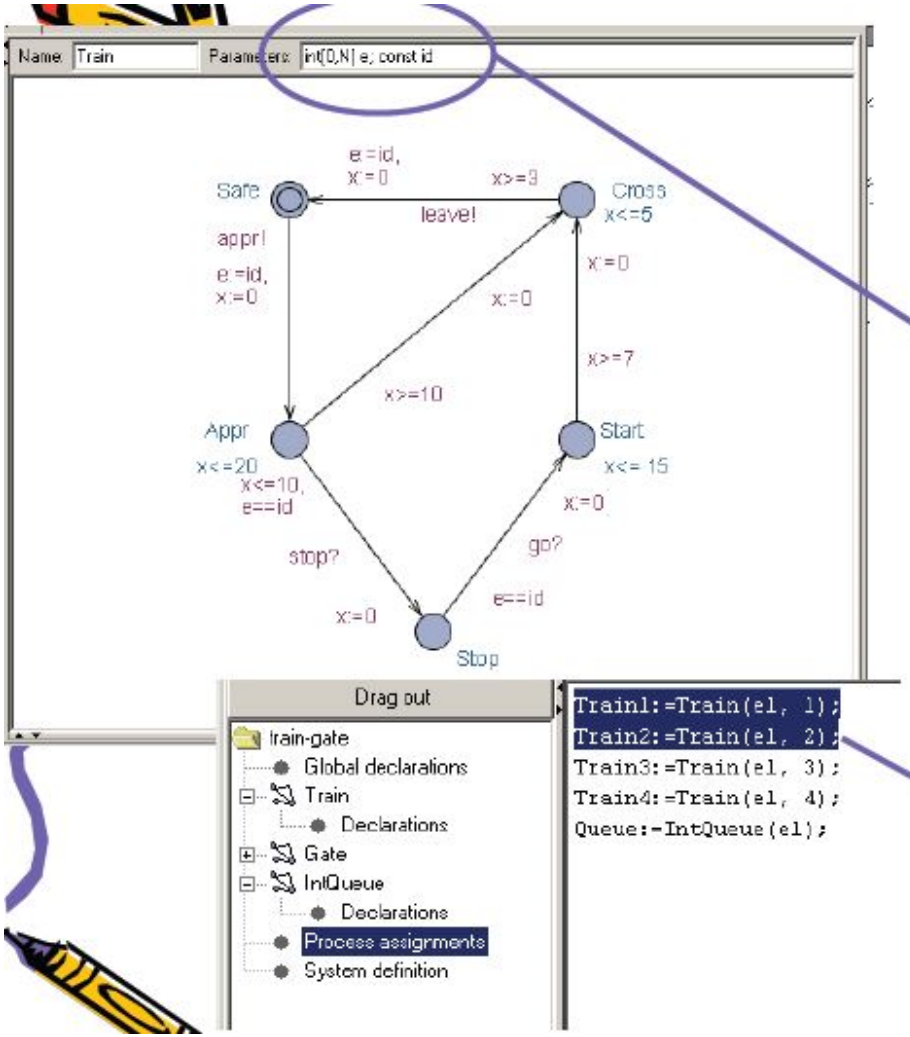




Declarations



Templates

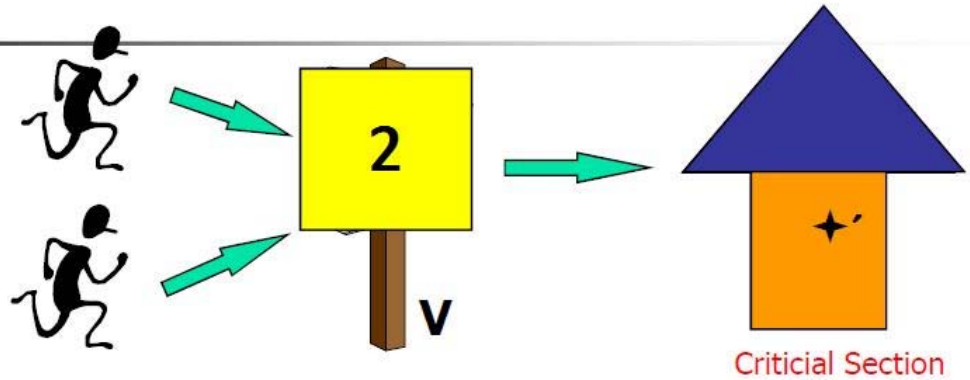


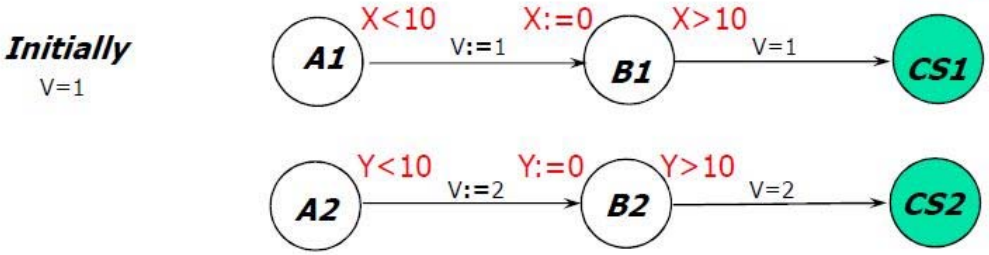
- Templates may be parameterised:
  - `int v; const min; const max`
  - `int[0,N] e; const id`
- Templates are instantiated to form processes:
  - `P:= A(i,1,5);`
  - `Q:= A(j,0,4);`
  - `Train1:=Train(el, 1);`
  - `Train2:=Train(el, 2);`

Bridge Puzzle

- 4 men, river, bridge, night, 1 flashlight
- at most 2 man on a bridge, flashlight necessary
- flashlight cannot be thrown
- men – different time to cross: 5 min, 10 min, 20 min, 25 min
- can they cross in 60 minutes?
- can they cross is less than 60 minutes?

Fischer's Protocol





- real-time protocol – correctness depends on timing assumptions
- simple, just 1 shared variable, arbitrary number of processes
- assumption: known upper bound D on reading/writing variable in shared memory
- each process has it's own timer (for delaying)

- https://github.com/kangeu/Fall2015

- Lab Assignments:

  Run Bridge Puzzle

- Work on the seminar paper “Gear Controller Case Study”

- Refer to UppalTutorial (Try a series of examples in the tutorial)