

The UPPAAL tool

M1-GLSD

2017-2018

TOV

L.Kahloul

Outlines of the course

- What and why?
- **Modelling language**: templates, constants & variables, synchronisation, **locations**: committed & urgent, **expressions**: select, guard, synchronisation, update, invariant
- **Verification**: TCTL
- **Simulation**

What and Why is Uppaal?

- **Uppaal**= developed jointly by [Basic Research in Computer Science](#) at **Aalborg** University in Denmark and the [Department of Information Technology](#) at **Uppsala** University in Sweden.
- Tool: Specification+Verification+Simulation
- Implementation: Server (specification)+client (query language for verification)
- Programming: Java, C++, xml

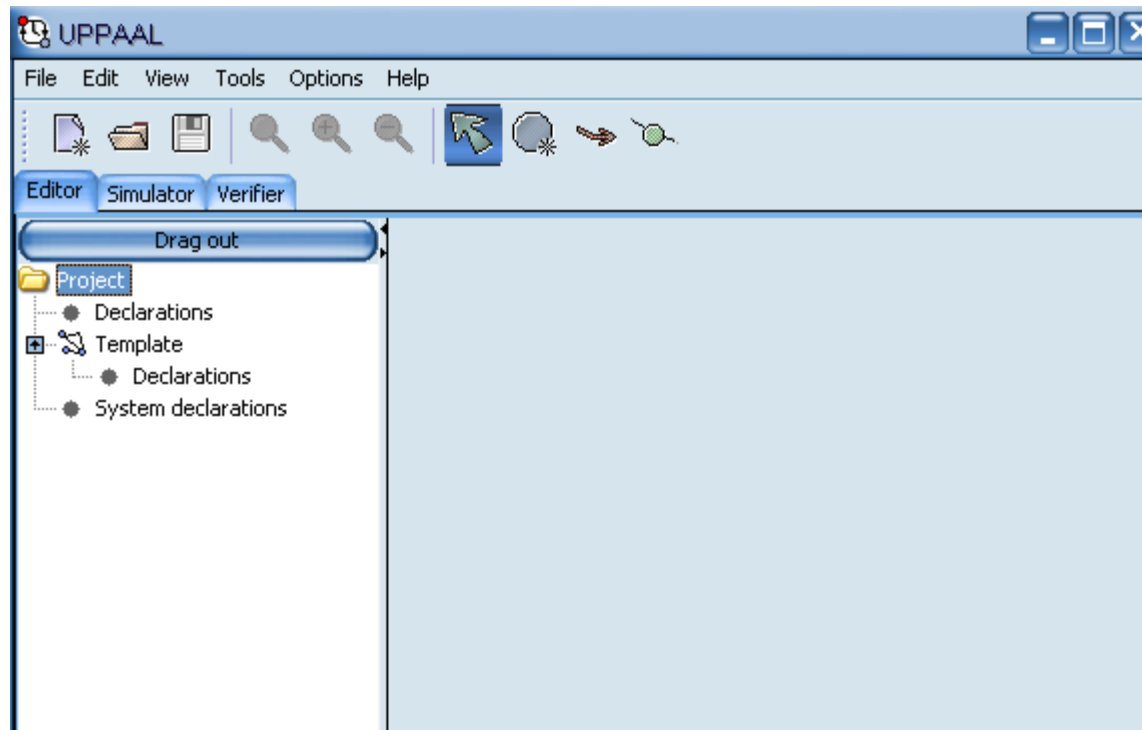
Modelling language of Uppaal

“what is in?”

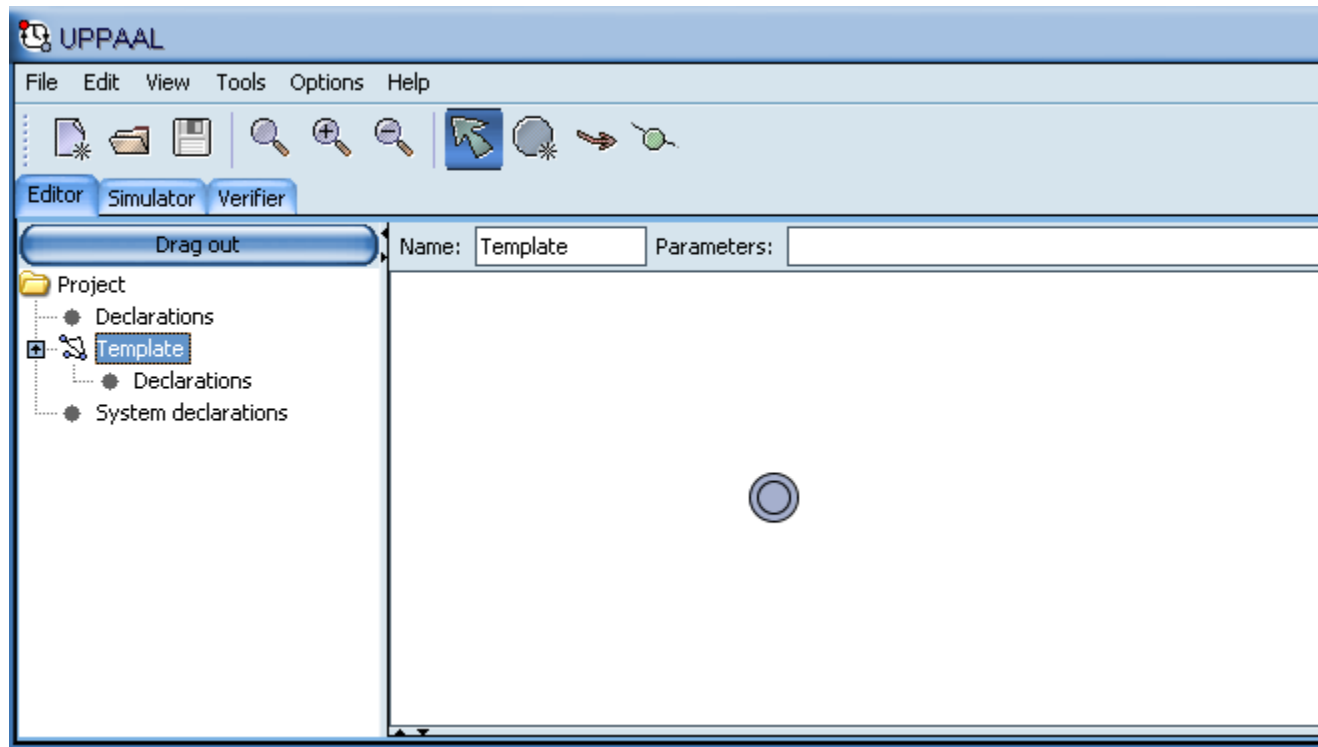
- **Besides** timed-automata, the Uppaal offers a **rich language** that facilitates modelling;
- The language used by uppaal is similar to a programming **language based on C language**;
- The language of uppaal introduces the concepts: **Template**, **constant**, **bounded integer variables**, **binary synchronisation**, **broadcast channels**, **urgent synchronisation**, **urgent or committed locations**, **arrays**, **initialiser**, **record types**, **custom types**, **user function**

Modelling language of Uppaal “templates”

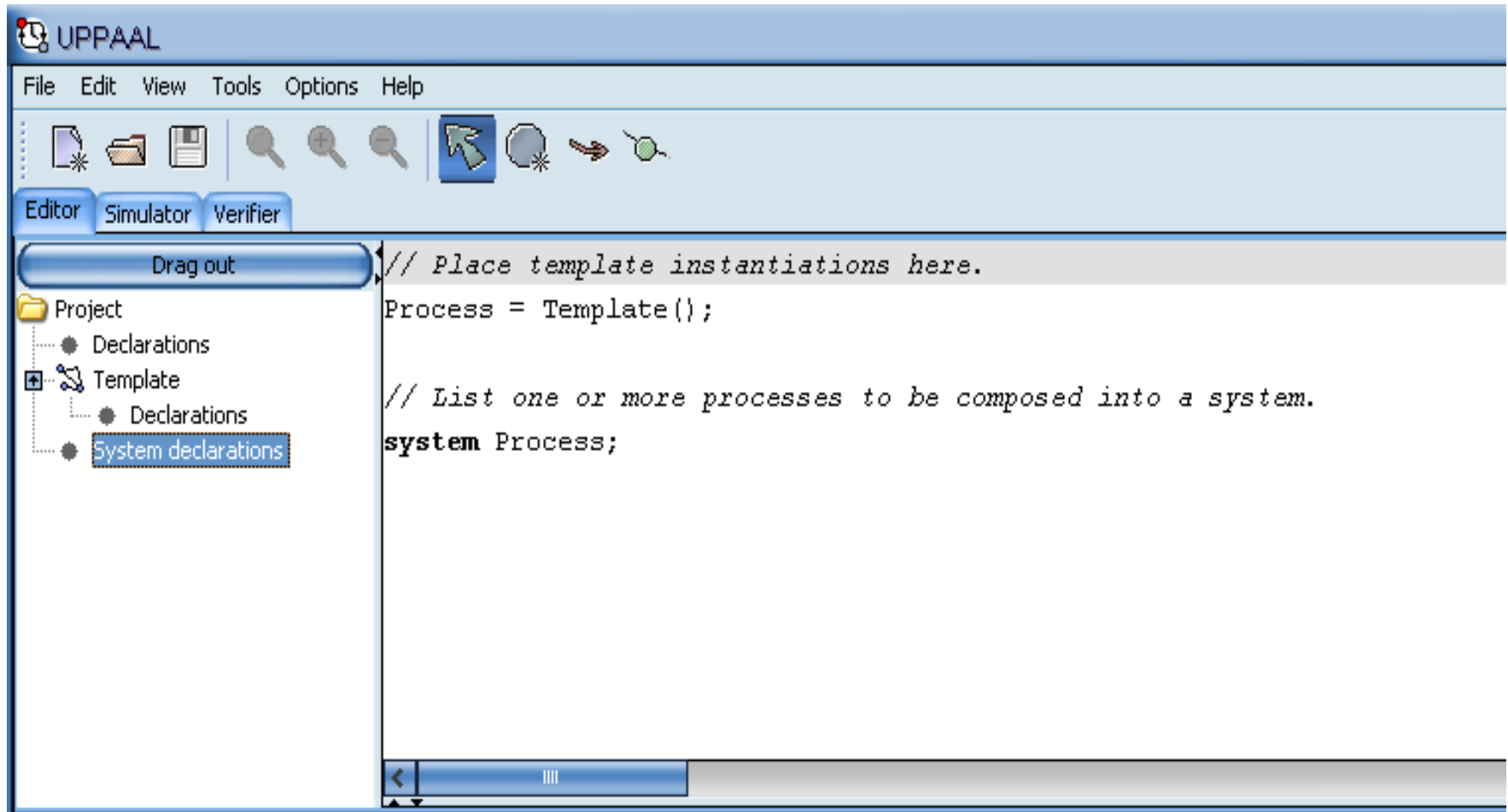
- **Template**: automaton is considered as a template. It can be instantiated and **parameterised** with some parameters.



Modelling language of Uppaal “templates”



Modelling language of Uppaal “templates”



Modelling language of Uppaal

“templates”

The screenshot displays the Uppaal software interface. At the top, the title bar reads "UPPAAL". Below it is a menu bar with "File", "Edit", "View", "Tools", "Options", and "Help". A toolbar contains various icons for file operations and editing. Below the toolbar are three tabs: "Editor", "Simulator", and "Verifier".

In the "Editor" tab, a "Drag out" button is visible. To its right, a form defines a template:

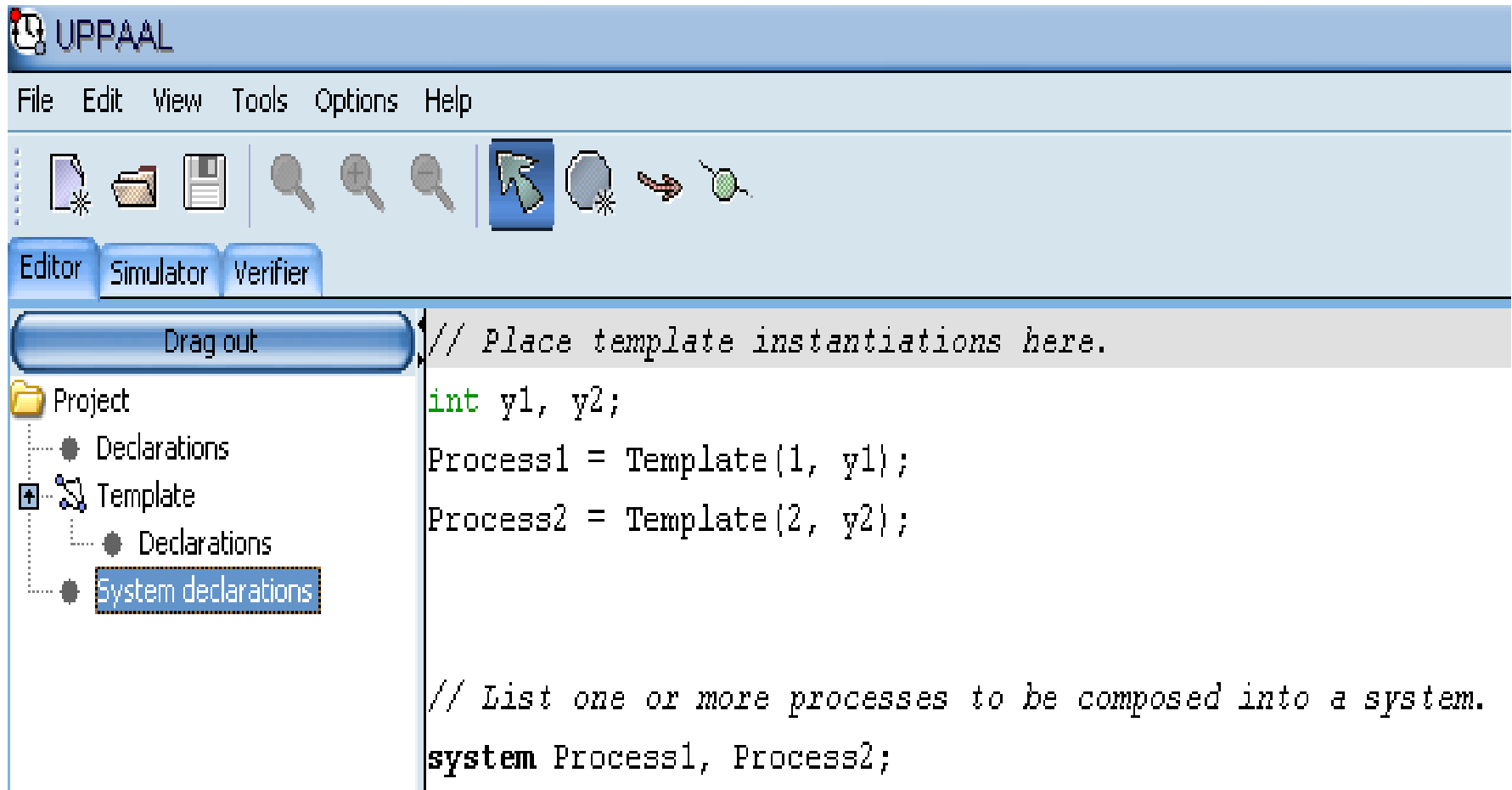
- Name:
- Parameters:

The "Parameters" field is circled in red. Below the form, a state transition diagram is shown, consisting of two states (represented by circles) connected by a transition labeled $y=x+1$.

On the left side, a project tree is visible under the heading "Project":

- Declarations
- Template (highlighted)
- Declarations
- System declarations

Modelling language of Uppaal “templates”



The screenshot displays the Uppaal software interface. At the top is the title bar 'UPPAAL' and a menu bar with 'File', 'Edit', 'View', 'Tools', 'Options', and 'Help'. Below the menu bar is a toolbar with various icons. The main window is divided into three tabs: 'Editor', 'Simulator', and 'Verifier', with 'Editor' being the active tab. On the left side, there is a project tree with a 'Project' folder containing 'Declarations', 'Template', and 'System declarations'. The 'System declarations' folder is selected and highlighted. The main editor area shows the following code:

```
// Place template instantiations here.  
  
int y1, y2;  
Process1 = Template(1, y1);  
Process2 = Template(2, y2);  
  
// List one or more processes to be composed into a system.  
system Process1, Process2;
```

Modelling language of Uppaal

“constants and variables”

- **constant**: `const name value;`

const int N=5; const int x=2;

- **bounded integer variables**:

`Int[min, max] name;`

`-32768 to 32768`

`Example: Int [2, 4] x;`

Modelling language of Uppaal “synchronisation”

- **binary synchronisation**: this requires the declaration of a channel between two templates (or automata).

chan name;

Example: **chan** move;

Two edges labelled **move?** (to receive) and **move!** (to send) must exist, respectively, in the two automata

- Send and receive are **blocking actions**

Modelling language of Uppaal

“synchronisation”

- **broadcast channels**: this requires the declaration of a broadcast channel between several templates (or automata).

broadcast chan name;

Example: broadcast **chan** move;

An edge labelled **move!** (to send) and several **move?** (to receive) must exist, respectively, in the sender and the receivers automata

- send is not a **blocking action**

Modelling language of Uppaal

“synchronisation”

- **urgent synchronisation**: the declaration of the channel is preceded by: **urgent**.

Example: **urgent chan** move;

- Edges using urgent channels for synchronisation cannot have time constraints, i.e., no clock guards.

Modelling language of Uppaal

“locations: urgent, committed”

- **urgent locations:** with a **U** inside the location. **time is not allowed to pass** when the system is in an urgent location.

How can we model this using the usual TA?

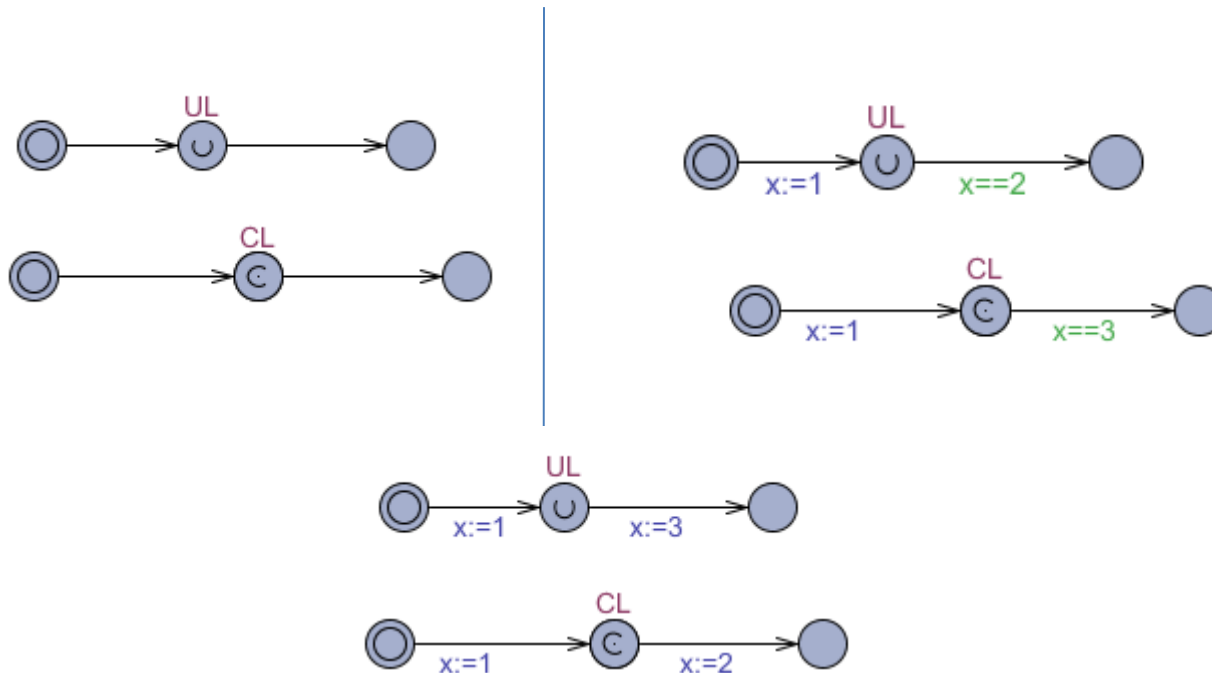
- **committed locations:** with a **C** inside the location. It is an **urgent location** & in a committed state (a state where at least one process is in a committed location) the **system must leave the committed location in the next transition** (i.e. the only possible transition is the one that fires the edge outgoing from a committed location).

why we use these locations ??????????

Modelling language of Uppaal

“locations: urgent, committed”: example

- How can the following processes work?



- Consider the two cases: x is shared or local.

Modelling language of Uppaal

- **arrays**, : we can have arrays of clocks, channels, constants and integer variables:

```
chan c[4];
```

```
clock a[2];
```

```
const int c[2]={0,2};
```

```
int[3,5] u[7];
```


Modelling language of Uppaal

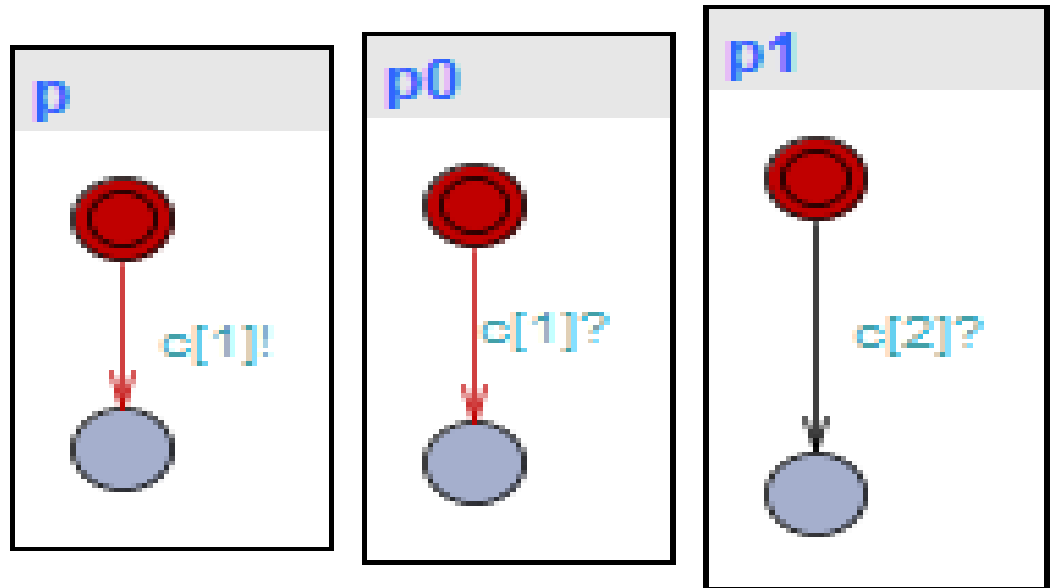
- **arrays of channels:**

chan c[n]:

- The value i is then used both as an array index when deciding what channel to synchronize on,
- and as an argument that can be used after.

Modelling language of Uppaal

- Example: even the three processes p , $p0$, $p1$ use the same channel c , but $c[1]$ synchronises, only, p with $p0$



Modelling language of Uppaal

- **initialiser**, are used to initialise integer variables and arrays of integer variables. Example:

```
int i = 2;  
int i[3] = {1, 2, 3};
```

- **record types** : are declared with the **struct** construct like in C

```
struct {  
  int x;  
  int y;  
} str;
```

Modelling language of Uppaal

- **custom types** : are defined with the C-like **typedef** construct.

```
typedef struct {  
    int x;  
    int y;  
} str_t;
```

```
str_t str;
```

```
str.x=1;
```

Modelling language of Uppaal

- **User function** : defined either globally or locally to templates.
- Template parameters are accessible from local functions.
The syntax is similar to C except that there is no pointer.

```
int f(){
    str_t str;
    str.x=1;
    return str.x;
}
```

Modelling language of Uppaal “expressions”

- Expressions range over clocks and integer variables.
- Four kinds of expressions: **select**, **guard**, **synchronisation**, **update**, **invariant**

Modelling language of Uppaal

“expressions”

- **Select**: (on edges)

Syntax: name1: type1, name2:type2, ...

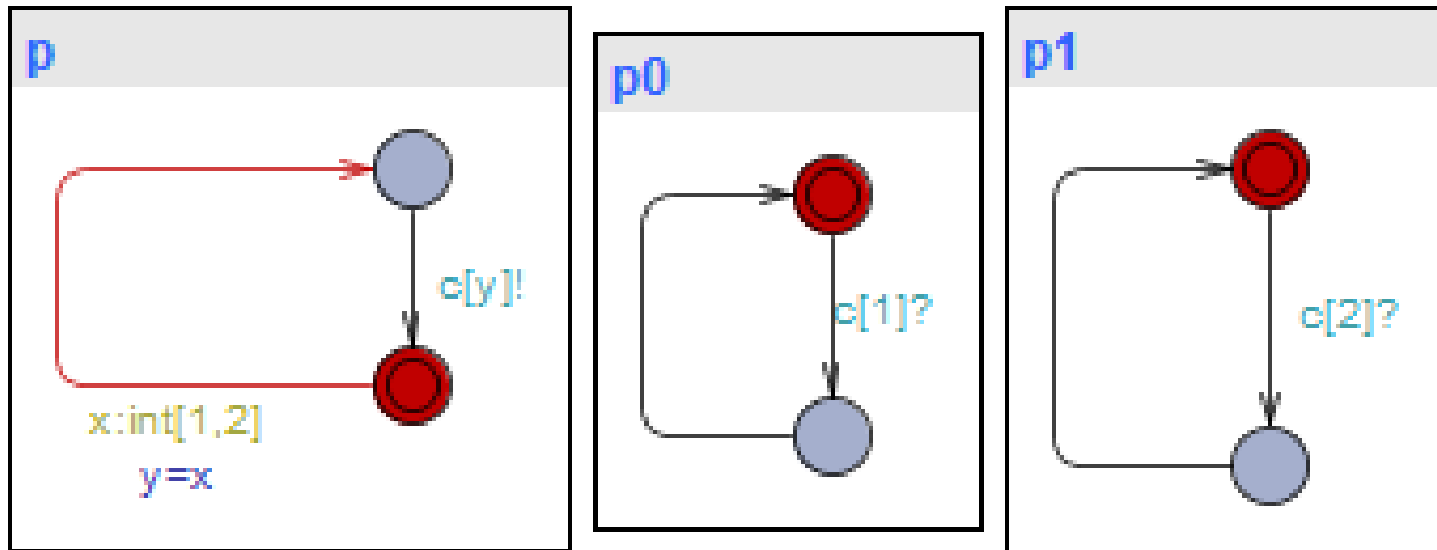
Semantics: assign randomly a value form the type to the name

Example:

- **x:int**; selects an integer random value;
- **x:int[2,3]**; selects an integer random value inside the interval [2,3]

Modelling language of Uppaal “expressions”

- Deduct the behaviour of this example



Modelling language of Uppaal “expressions”

- **Guard:** (on edges)

Syntax: expression1 **op** value1 **and** expression2
op value2 **and** ...

op in {==, <, >, <=, >=}

Example: $x==2$ and $y<=3$ and $x-y<=0$ and ...

Modelling language of Uppaal “expressions”

- **Synchronisation**: (on edges)

Syntax: chan_name!

chan_name?

Semantics: synchronise with another transition in another automaton

- **Update**: (on edges)

Syntax: var_name1:=value1, var_name2:=value2, ...

Example : x:=1, y:=2, z:=4, ...

Modelling language of Uppaal “expressions”

- **Invariant: (on locations)**

Syntax: expression1 **op** value1 **and** expression2
op value2 **and** ...

op in {==, <, >, <=, >=}

Example: $x==2$ and $y<=3$ and and $x==y$ and $x-$
 $y<=0$ and ...

Remarks: (1) x, y can be variables or clocks

Verification with Uppaal

“TCTL”

- Two kinds of formulae
 - 1) State formulae describe individual states:
(**name_proc.name_loc**)
 - 2) Futur (F) is written: $\langle \rangle$, and Globally (G) is written $[]$
 - 3) Path formulae (quantify over paths or traces of the model):
 - reachability: $\mathbf{E}\langle \rangle \varphi$
 - safety: something good is invariantly true. $\mathbf{A}[] \varphi$
 - liveness: something will eventually happen $\mathbf{A}\langle \rangle \varphi$

Verification with Uppaal

“TCTL”: example



If `proc` is the name of this process,

- **$A\langle\rangle \text{proc.loc2}$** : the location `loc2` is reachable eventually ($\langle\rangle$) in all paths (**A**),
- **$A[] \text{proc.loc2}$** : the location `loc2` is reachable globally ($[]$) in all paths (**A**)
- **$A\langle\rangle \text{proc.x} \geq 1$** ???
- **$E\langle\rangle \text{proc.x} \geq 1$** ???
- **$p \dashrightarrow q$ is equivalent to $A[](p \Rightarrow A\langle\rangle q)$**