Uppaal. The Model Checker.

Patryk Kiepas

March 23, 2016

Uppaal. What is it?

Uppaal is a model checker for real-time systems (in mind of embedded systems). What we can do with it?

1. Modeling
2. Simulation
3. Verification

Internal representation of model consists of:

- Network of timed automata
- Extended with data types

Where to use?

"Any system can be analysed using a model checker, as long as it has *states* and *transitions* between states" (from Chapter 1: A First Introduction to Uppaal by Frits Vaandrager)

Reactive systems such as:

- Hardware components
- Embedded controllers
- Network protocols
- Others...

Whenever there is need to handle real-time issues (the timing of transitions).

Brief history

Uppaal was started by Uppsala University, Sweden and Aalborg University, Denmark.
Time-line of development:

- 1995 - project started
- 1999 - first beta
- 1999/2000 - first stable release (v 3.0.X)
- September 27, 2010 - latest stable release (v 4.0.13)
- July 1, 2014 - preview release (v 4.1.19)

Uppaal variations

Versions: Windows, Mac, Linux, 32/64 bits
Available licenses:

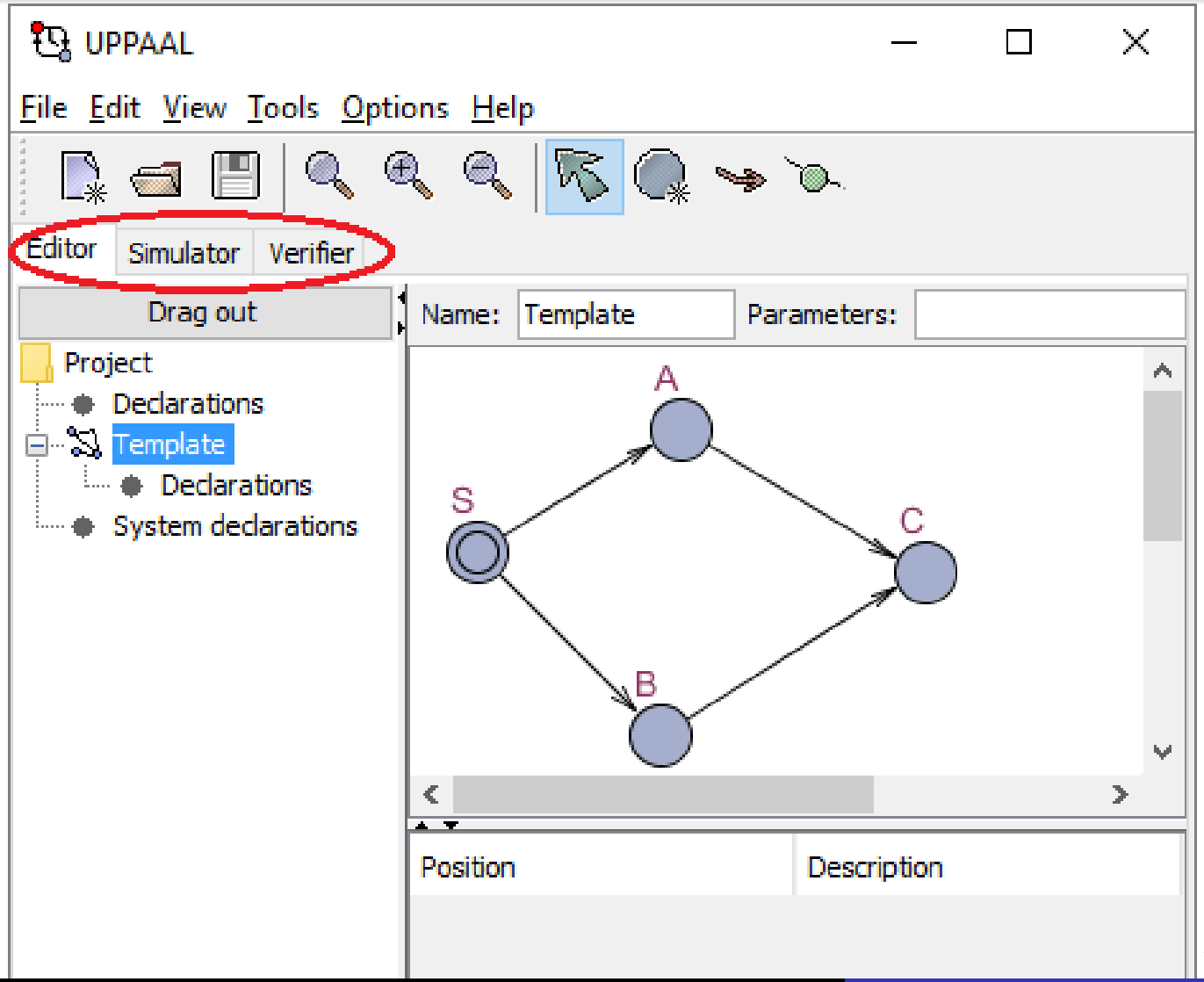
- Academic use (more info: http://www.uppaal.org/)
- Commercial use (more info: http://www.uppaal.com/)

Uppaal GUI was programmed in Java, and Uppaal checking engine was developed using C++.

Extensions

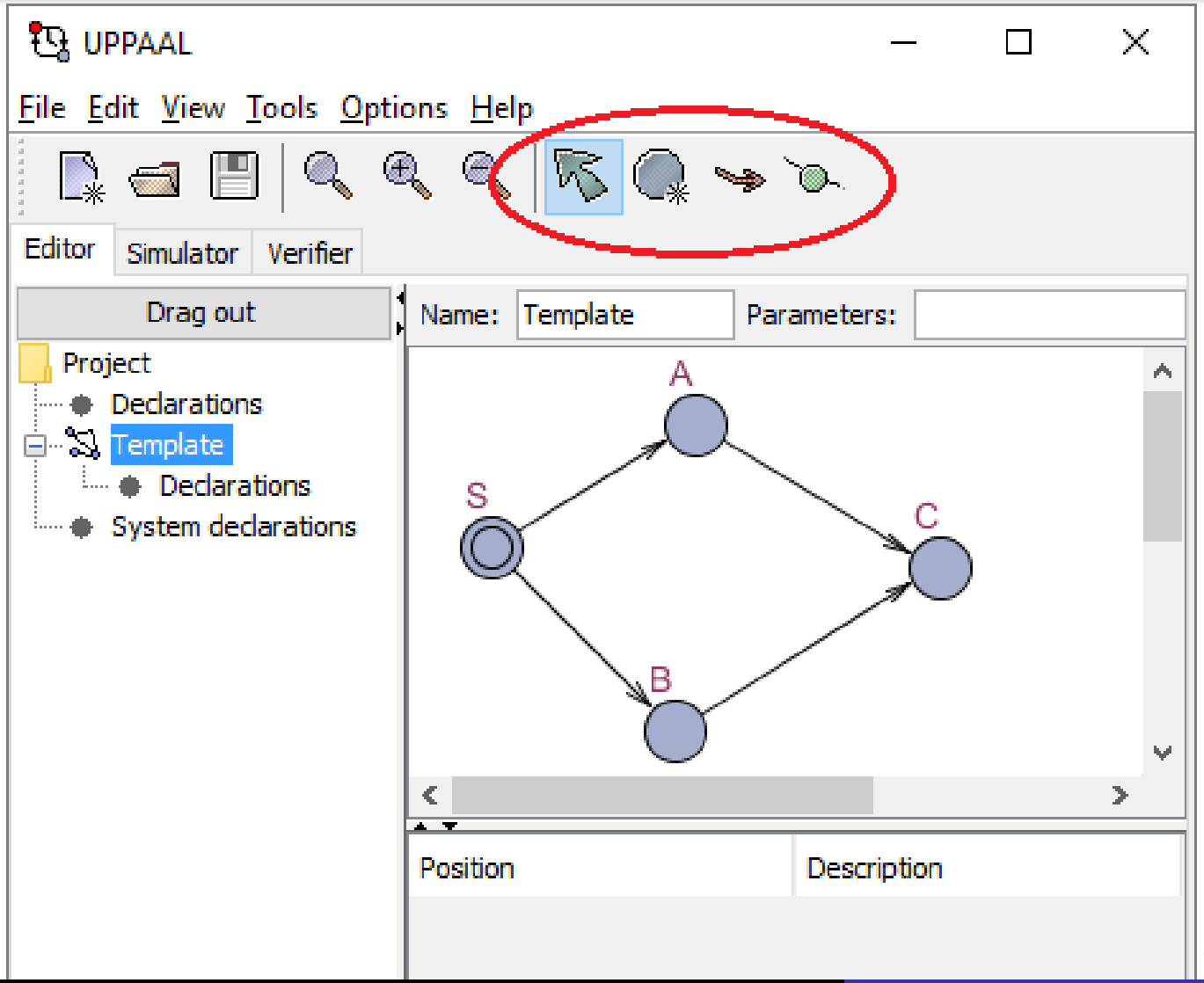
- **Cora** - Cost Optimal Reachability Analysis;
- **Tron** - Testing Real-time systems ON-line (black-box conformance testing);
- **Cover** - COVERerage-optimal off-line test generation;
- **Tiga** - TImed GAmes based controller synthesis;
- **Port** - component based timed systems, exploiting Partial Order Reduction Techniques;
- **Pro** - PRObabilistic reachability analysis;
- **Times** - A Tool for Modelling and Implementation of Embedded Systems;
- **Stratego** - Strategy synthesis, learning, optimization and evaluation;
- **SMC** - Statistical Model-Checker.

Uppaal GUI - Main parts



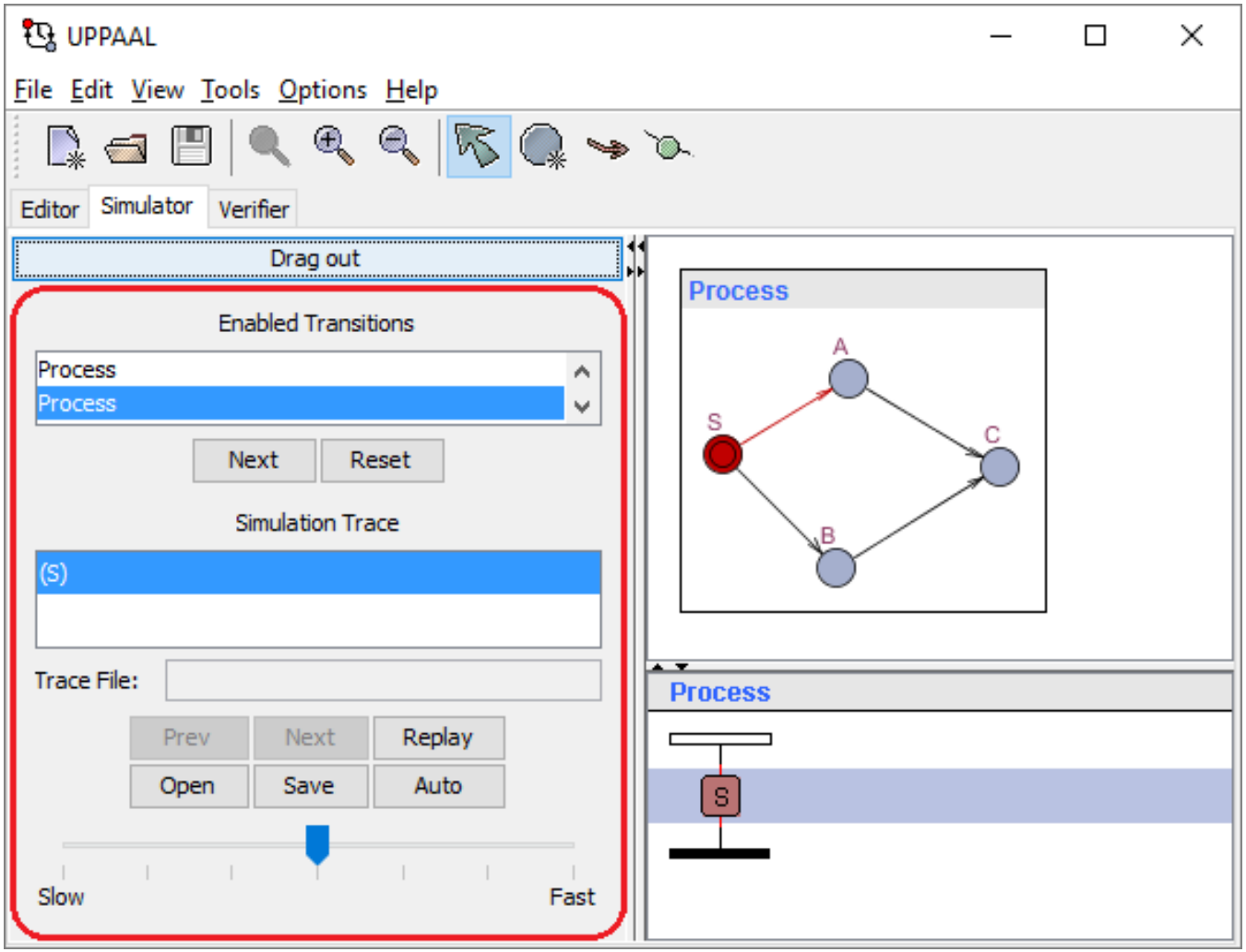
- System editor
- Simulator
- Verifier

Uppaal GUI - System editor



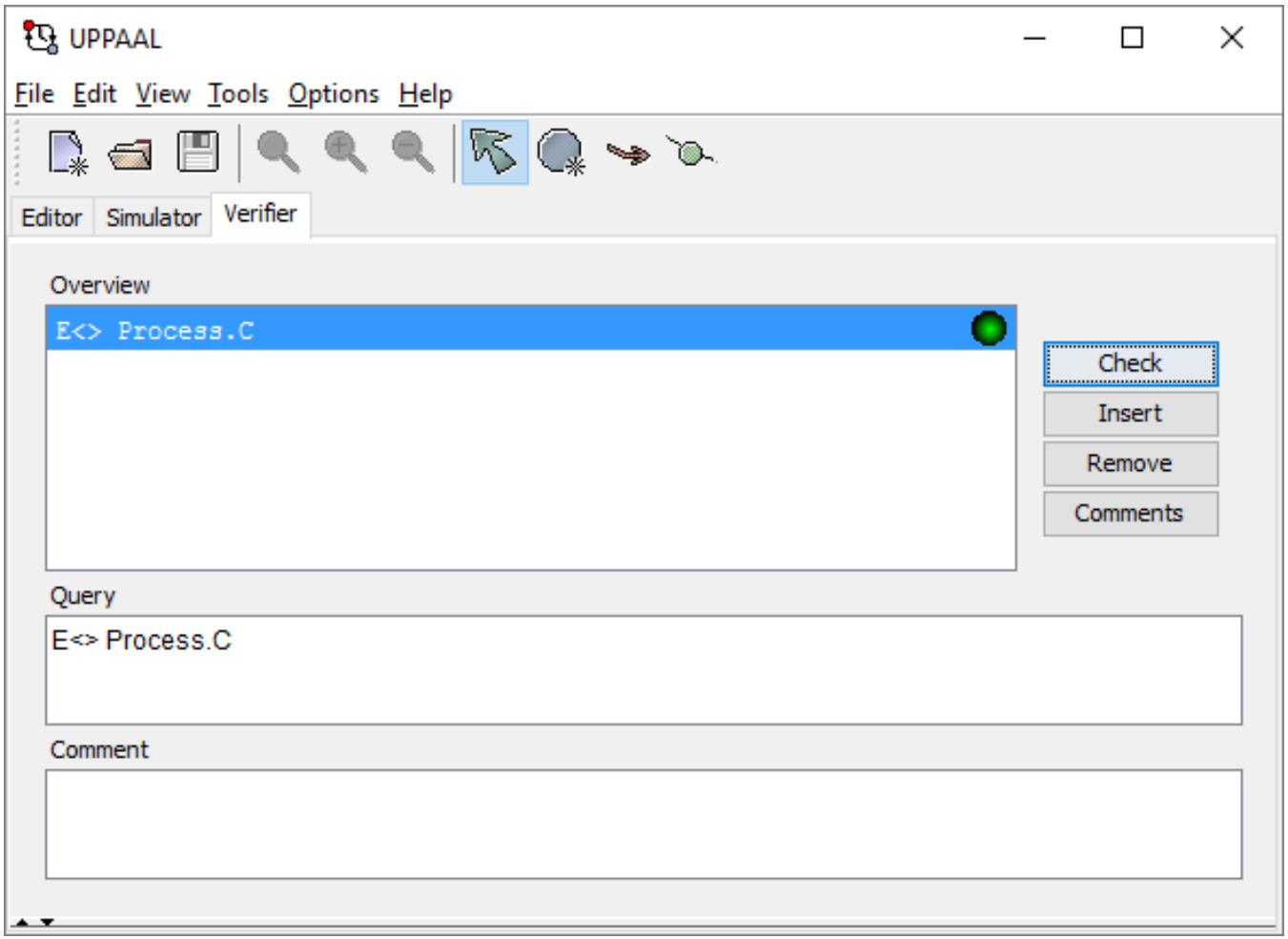
- Name: Template (default)
- Select
- Add location
- Add edge
- Add nail
- Syntax check (for global, local, system declarations)

Uppaal GUI - Simulator



- Select transition
- Track simulation
- Control (Prev/Next/Auto)
- Visualization

Uppaal GUI - Verifier

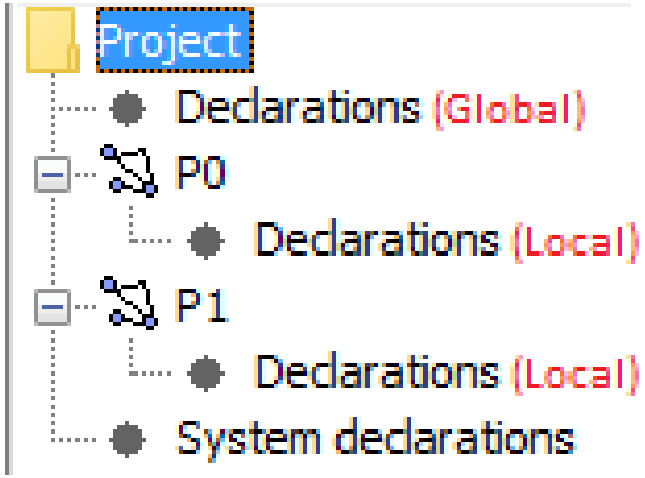


- Query editor
- Check query
- Overview
- Save/load

System/model/project

Description of system consist of:

- Concurrent process templates modelled using timed-automata (here *P0* and *P1*);
- Local declarations for each process;
- Global declarations for whole system;
- System definition.



Process/automata

Process is a timed-automata represent as diagram with states (called locations) and transitions between states (called edges). Timed-automata is finite state machine with time (clocks).

Each process has only one **initial location**.

Processes execute concurrently and they can be synchronized using channels.

Time/clocks

Time is continuous and the clocks measure time progress. **Time progress globally** and clocks values **increase at the same rate** for the whole system.

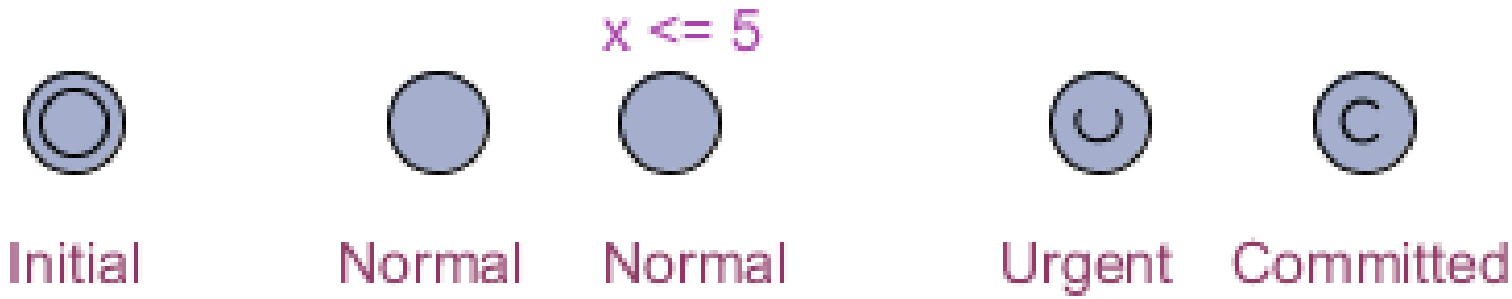
A clock is a special type of variable with domain being a set of non-negative real numbers. At system start, all clocks have value 0.

Using clocks we can specify:

- **Invariants** - upper bounds on timing, describes how long we can stay in given *location* (e.g. $x <= 5$);
- **Guard** - lower bound on timing, describes after what amount of time a *transition* can be executed (e.g. $x > 2$).

We can test the value of clock using standard expressions or we can reset clock.

Location/state (part I)



Location represents state of the system. There are four types of locations:

- **Initial** (one for each process);
- **Normal** (with or without **invariants**);
- **Urgent** (time cannot elapse in this location, so transition to another location must occur **immediately**);
- **Committed** (they freeze time, if any process is in committed location then the next transition must involve and edge from one of the committed locations).

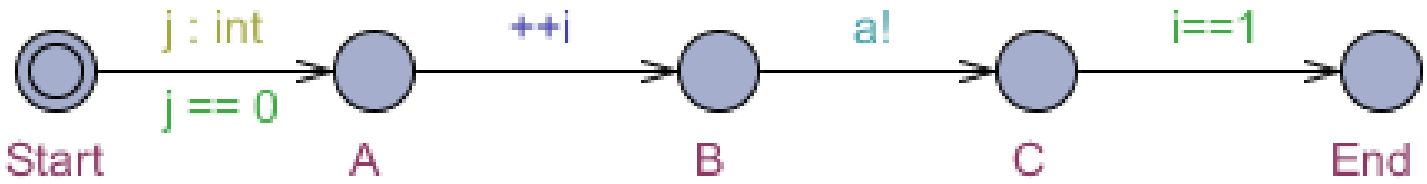
Edge/transition (part I)

An edge/transition connects two locations. Edges can have four types of annotations:

- **Selection** - binds a given identifier to a random value in a scope of current transition; allowed types: *boundend integers*, *scalar sets*; defined here identifiers will shadow local/global variables.
- **Guard** - transition is enabled only if guard's expression evaluates to **true** and consists of:
  - Conjunction of simple conditions on clocks;
  - Differences between clocks;
  - Boolean expressions not involving clocks.
- **Synchronization** - transition labelled with complementary actions (e.g. $a!$ and $a?$) will synchronize over a common channel (here channel a)

Edge/transition (part II)

- **Update** - evaluate given expression when transition occur.



Example of transitions with annotations:

- Selection: *j : int*;
- Update: *++i*;
- Synchronization: *a!*;
- Guard: *j==0*, *i==1*.

Edge/transition (part III)

- Without prior specification, **all transitions occur instantaneously** and do not take time (they have default guard which always evaluate to **true**);
- When no further transition is possible we reach so called **deadlock state**;
- Unspecified choice is called non-deterministic (look at figures below).

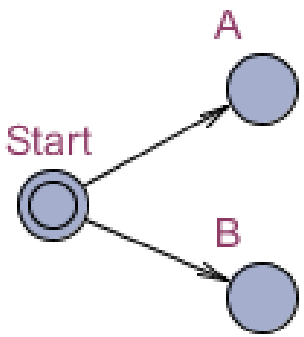


Figure: Non-deterministic choice.

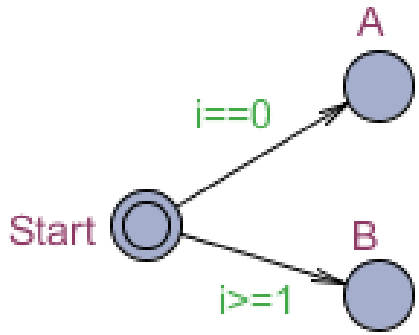


Figure: Deterministic choice.

Location/transition - a thing to remember

If not specified, *locations* can wait **forever**, but *transitions* will execute **immediately**.

Parameters of process template

Template of a process can have input parameters with different pass semantics (C++ syntax & semantic):

- **pass-by-value** (using local copy, e.g. *int a*)
- **pass-by-reference** (using original value, e.g. *int& a*)

Clocks and channels should be pass-by-reference parameters.

Examples:

- *P(clock &x, bool bit)*
- *Q(clock &x, clock &y, int i1, int &i2, chan &a, chan &b)*

Creating such template: *P0 = P(clock, true);*

This pass semantics are true also for functions' parameters.

Declarations

Declarations can be global (in one place) or local (to a template). They mostly consist of programming statements and expressions with syntax similar to C++. Declarations mostly consist of:

- Clocks (e.g. *clock x, y;*)
- Bounded integers (e.g. *int[0,100] a=5;*)
- Channels (e.g. *chan d; urgent chan e;*)
- Arrays (e.g. *int a[2][3]* $= \{\{1, 2, 3\}, \{4, 5, 6\}\}$)
- Records (e.g. *struct* $\{$*int a; clock c;*$\}$ *S;*)
- Types (e.g. *typedef* keyword)
- Functions

Uppaal does not support enumerated types.

Data types (part I)

We have four predefined data types:

1. Integer (*int*) - default value range is $[-32768, 32767]$;
2. Boolean (*bool*) - **true** (1)/**false** (0) values;
3. Clock (*clock*) - non-negative real values;
4. Channel (*chan*) - with three subtypes: **normal**, **urgent** and/or **broadcast**.

Data types (part II)

- We can have **constant** integers, booleans and arrays/records over integers and booleans (e.g. *const int i = 1;*).
- We can have these variables marked as **meta**, so that they are stored in the state vector, but they are not part of state. Meta variables won't affect state a space exploration (e.g. *meta bool edgeVisited[N_EDGES];* or for swapping variables).
- Variable assigned with a value outside of its domain generates "run-time error".

Functions

Almost identical to C++ like functions. Some simple examples:

```
void initialize(int& a[10]) {
    for (i : int[0,9]) {
        a[i] = i;
    }
}

void swap(int &a, int &b) {
    int c = a;
    a = b;
    b = c;
}
```

System declaration

We can define here global variables, channels and functions. They are not accessible in process templates because they are defined before them.
First we instantiate processes then we create system. Examples:

```
process ProcA() {}
process ProcB(int& x, bool b, int y) {}
(...)
// System declaration starts here!
int x = 5;
P0 = ProcA();
P1 = ProcB(x, true, 15);
// Partial instantiation
Q(int& z) = ProcB(x, true, z);
P2 = Q(10);

system P0, P1, P2;
// Or with priorities
system P0 < P1 < P2;
```

Synchronization and channels

In Uppaal there is a hand-shaking synchronization: two processes take a transition at the same time, only when one will have an **a!** (emit) and the other an **a?** (receive), where **a** is a channel used.

First the **update expression** is executed on transition synchronized on **a!** then on other transition annotated with **a?**. We have three types of channels:

1. **Binary/Normal**;
2. **Broadcast** - allows 1-to-many synchronization.
3. **Urgent** - source state can't delay triggering a synchronized transition, it must be triggered instantly when possible; clock conditions on these transitions are not allowed;

There is no value passing through the channels.

Priorities

Channels and processes can have specified priorities. Examples for channels:

```
chan priority a, b < default < c;
```

Processes priorities are specified in system declaration:

```
system P0 < P1 < P2, P3;
```

Queries (part IA)

Query describe a property expressed as temporal logic formula, that may or may not hold for a given model. The Verifier can establish whether the query is **satisfied** or **not**.
Basic queries:

- *A[] p* : for all paths *p* **always holds**;
- *E[] p* : there exists a path where *p* **always holds**;
- *A<> p* : for all paths *p* will **eventually hold**;
- *E<> p* : there exists a path where *p* **eventually holds**;
- *p –> q* : whenever p holds q will eventually hold.

Simple queries are in form of e.g *A[] p* where *p* is an expression build from **boolean combination** of **atomic propositions**.

Queries (part IB)

Property equivalences:

Name	**Property**	**Equivalent to**
Possibly	*E<> p*	
Invariantly	*A[] p*	*not E<> not p*
Potentially always	*E[] p*	
Eventually	*A<> p*	*not E[] not p*
Leads to	*p –> q*	*A[] (p imply A<> q)*

Queries (part II)

The simplest atomic proposition can be of the form *P0.C*, where *P0* is an automaton and *C* is a location. Such a proposition is **true** if process *P0* is in location *C*.

Expression	**Name**	**True when...**
e && *f*	and	*e* and *f* evaluate to **true**
e || *f*	or	*e* or *f* evaluate to **true**
e == *f*	equality	*e* and *f* evaluate to the same value
e imply f	implication	*e* evaluate to **false** or *f* evaluate to **true**
e not f	negation	*e* evaluates to **false**

Queries (part III)

- $A[]$ not deadlock
- $E <>$ Process_1.C
- $E <>$ (Process_1.C && Process_2.C)
- $A[]$ now $>=$ 200 imply (Belt.end && Jobber1.begin &&Jobber2.begin)
- $A[]$ Obs.taken imply ($x >= 2$ and $x <= 3$)
- $E <>$ Obs.idle and $x > 2$
- $A[]$ Obs.idle imply $x <= 3$

Reasoning (part I)

The checking engine do an exhaustive search that covers all possible dynamic behaviours of the system in order to check specification properties described in form of queries.

"More precisely, the engine uses **on-the-fly verification** combined with a **symbolic technique reducing the verification problem** to that of **solving simple constraint systems** [YPD94, LPY95]. The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using testing automata [JLS96] or the decorated system with debugging information [LPY97]." (*Uppaal 4.0 : Small Tutorial*)

Reasoning (part II)

Uppaal provide a concrete example illustrating why the property holds (for $E<>$ properties) or not (for $A[]$ properties, so called: *counterexample*).

In the case of $E<>$ properties that do not hold, or $A[]$ properties that hold, Uppaal can only report that it exhaustively checked all the reachable states of the model and didn't find anything.

It is worth to point out that model checker engine can run in **server mode** on a more powerful, dedicated machine.

Saving data

We can save various types of data:

- Model/system - **.xml* file;
- Queries - **.q* file;
- Traces (binary) - **.xtr* file.

Keeping models manageable

- **Committed locations** - reduce significantly the state space, but on the other hand they can take away relevant states;
- **Variables with ranges** - use small amount of variables with the shortest value ranges possible;
- **Minimum amount of clocks** - has an important impact on the complexity of the model.

Examples

1. Non-deterministic process with synchronization (with one and two loops);
2. Taken-idle with loop;
3. Production line.

- F.W. Vaandrager. *"A First Introduction to Uppaal"* In J. Tretmans, editor. Quasimodo Handbook. To appear.
- *Uppaal 4.0: Small Tutorial. A short description of the tool as well as some examples.*
- *Uppaal Help/Documentation*. Built-in. Uppaal 4.0.14, May 2014.