Uppaal. The Model Checker.

Patryk Kiepas

March 22, 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:

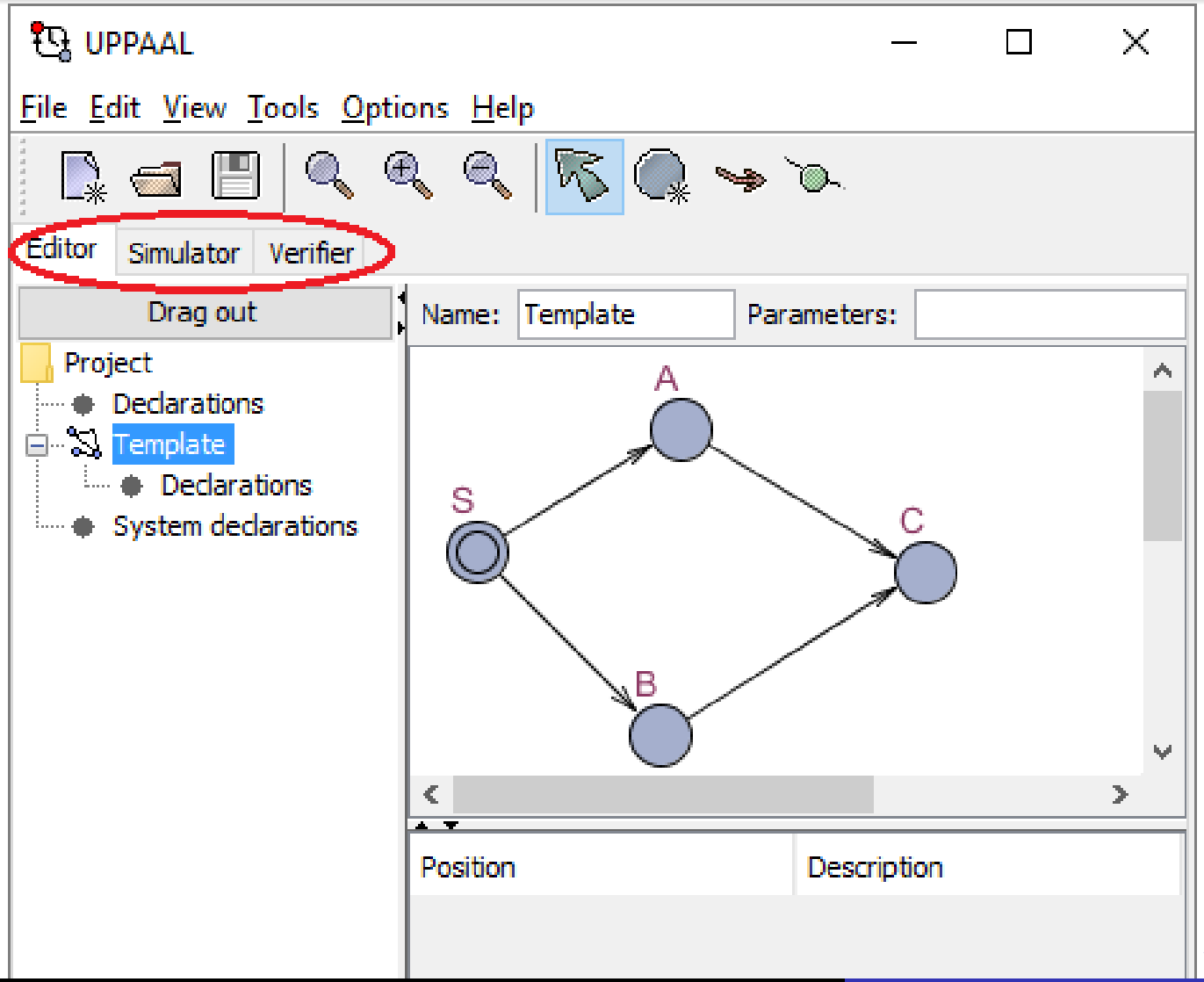
- 1995 - project started
- 1999 - first beta
- 1999/2000 - first stable release (v 3.0.X)
- September 27, 2010 - latest stable release (v 4.0.13)
- July 1, 2014 - preview release (v 4.1.19)

Uppaal variations

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

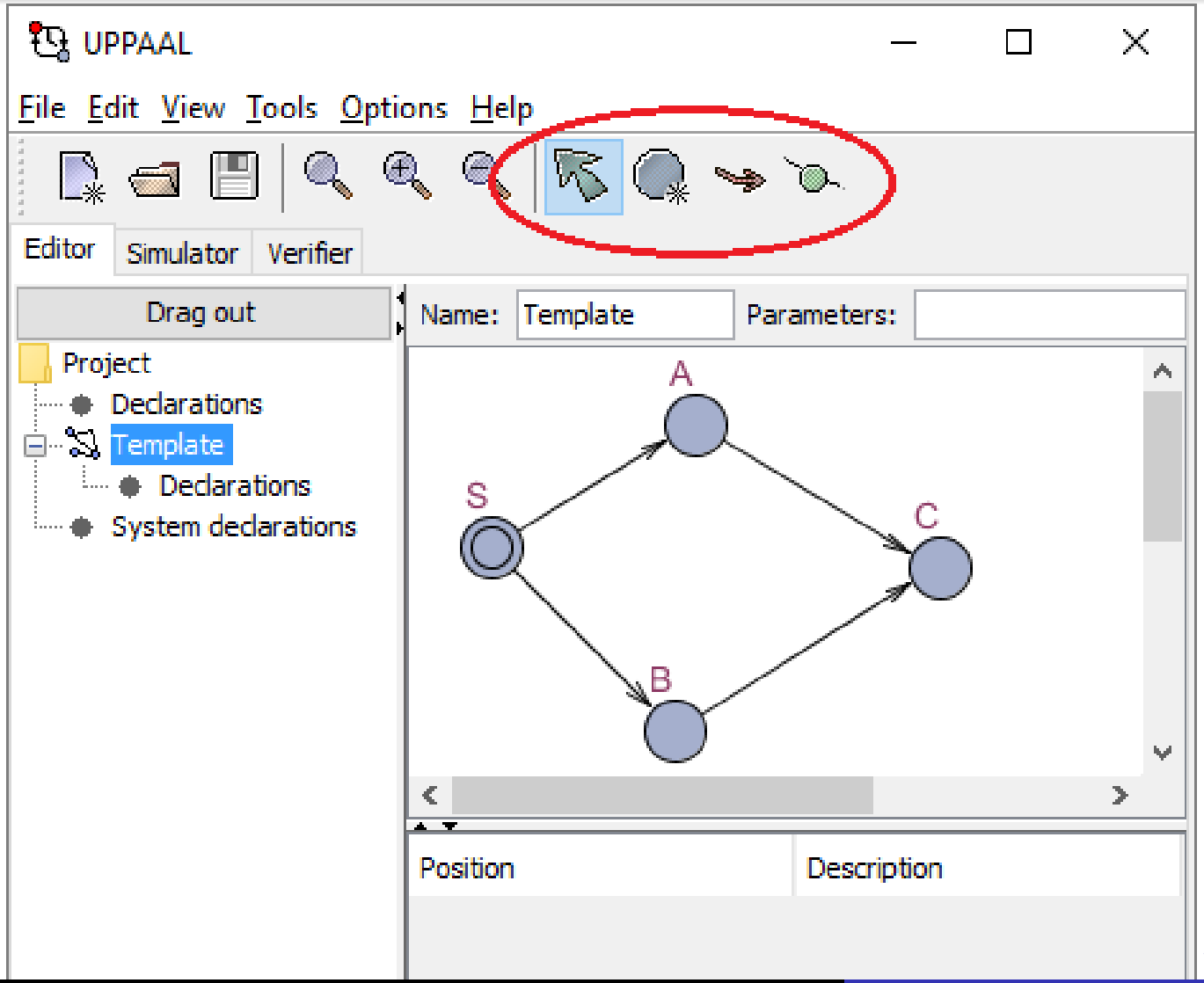
- Academic use (more info: http://www.uppaal.org/)
- Commercial use (more info: http://www.uppaal.com/)

Uppaal GUI - Main parts



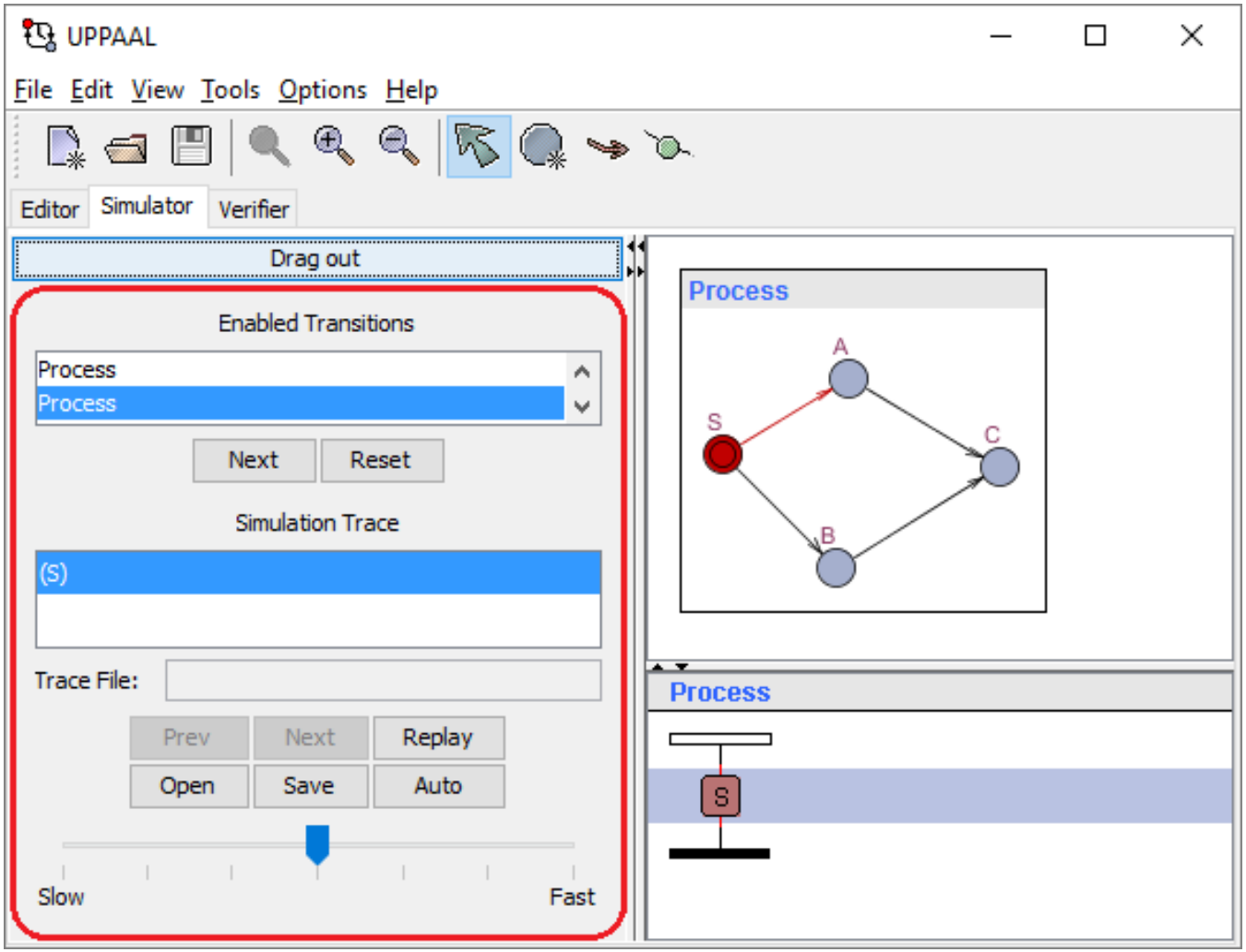
- System editor
- Simulator
- Verifier

Uppaal GUI - System editor



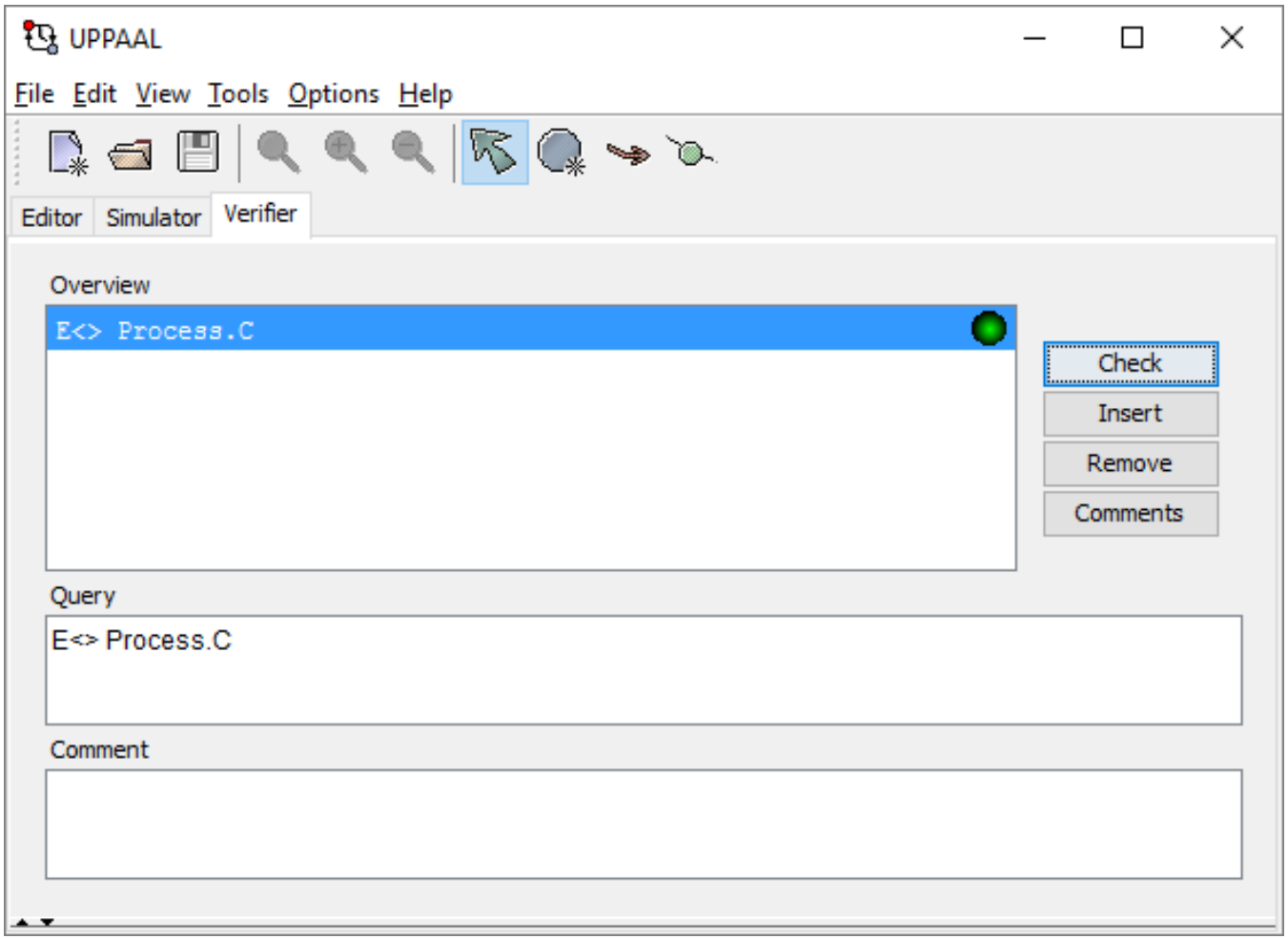
- Name: Template (default)
- Select
- Add location
- Add edge
- Add nail
- Syntax check (for global, local, system declarations)

Uppaal GUI - Simulator



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

Uppaal GUI - Verifier

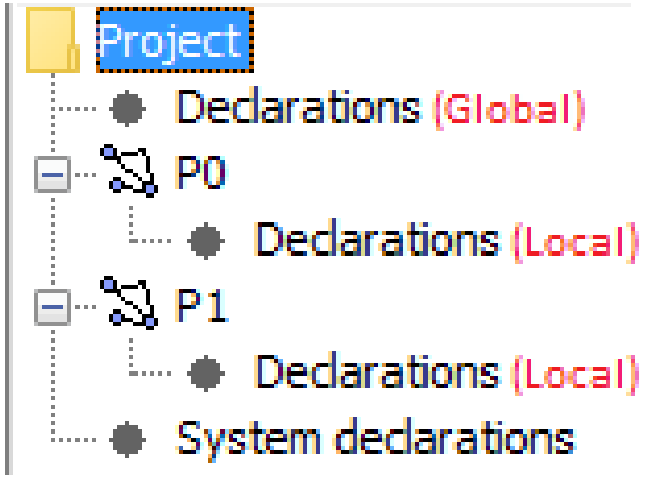


- Query editor
- Check query
- Overview
- Save/load

System/model/project

All systems consist of:

- Global declaration for whole system;
- Concurrent processes modelled using timed automata (here *P0* and *P1*);
- Local declaration for each process;
- System description.



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**.

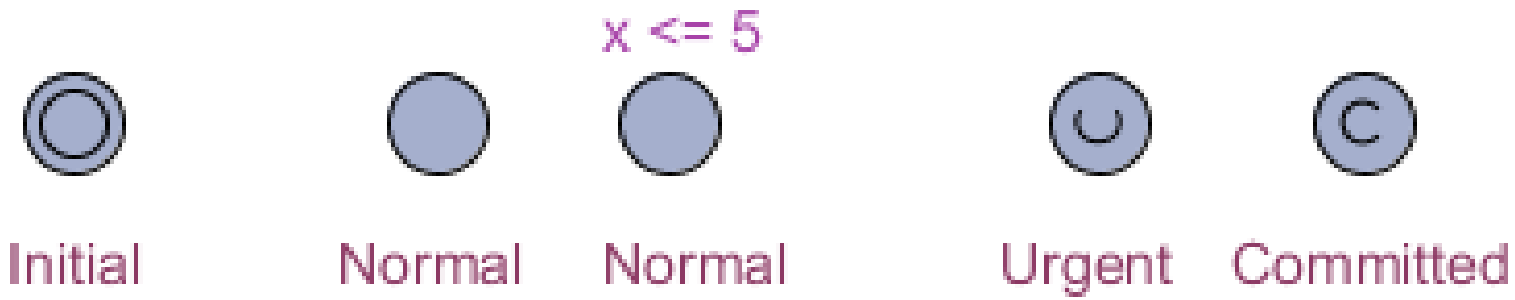
Time and clocks

(!) Are the way to handle time in Uppaal. Time is continuous and the clocks measure time progress. It is allowed to test the value of a clock or to reset it. Time will progress globally at the same pace for the whole system. A system in Uppaal
It allows to track:

- The ordering of events
- How fast some events occur

In Uppaal we can specify upper bounds on timing, using so-called invariants. When we double click the location work easy, a window appears with a field In-variant.
The invariant is a progress condition.
If we want to specify lower and upper bounds on the time that an automaton may stay in a certain location, we can do this in Uppaal using so-called clocks. A clock is a special type of variable, whose domain consists of the set of non-negative real numbers. Just like

Location/state



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

- Initial (at most one initial location in each automaton);
- Normal (with or without **invariants**);
- Urgent (time cannot elapse in this location, so transition to another location must occur **immediately**);
- Committed.

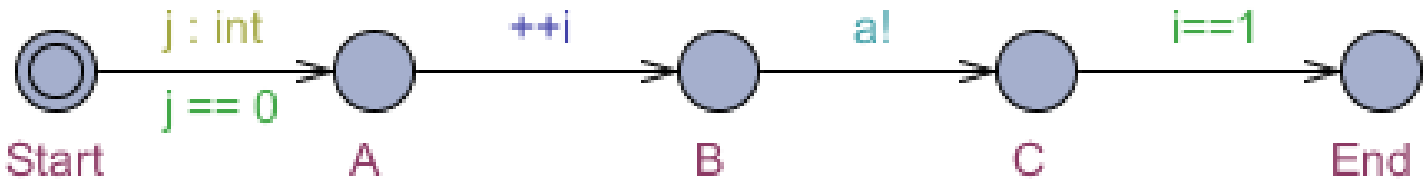
Edge/transition (part I)

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

- Selection - binds a given identifier to a value in a given range; 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.

When no further transition is possible we reach so called **deadlock state**.

Unspecified choice is called non-deterministic:

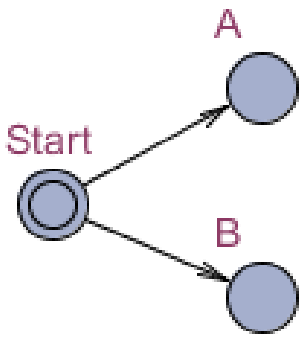


Figure: Non-deterministic choice

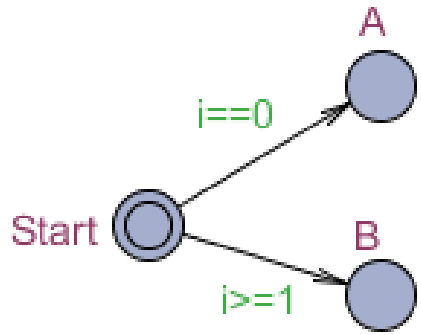


Figure: Deterministic choice

Parameters

Template of process can have parameters with different call semantics (C++ syntax):

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

Clocks and channels must always be call-by-reference parameters.
Uppaal does not allow data parameters for synchronization channels (?).
In template parameters field: urgent chan &get, chan &put. When creating templates: Hammer = Tool(get_hammer, put_hammer); Mallet = Tool(get_mallet, put_mallet);

System declaration

Channels

(!) Uppaal offers urgent channels (defined using urgent chan) that are synchronization that must be taken when the transition is enabled, without delay. Clock conditions on these transitions are not allowed.
(!) There is no value passing through the channels
(!)The synchronization mechanism in Uppaal is a hand-shaking synchronization: two processes take a transition at the same time, one will have an a! and the other an a?, with a being the synchronization channel. When taking a transition, two actions are possible: assignment of variables or reset of clocks.
Uppaal does not allow data parameters for synchronization channels.
When a synchronization channel is urgent, this means that whenever a synchronization with this channel is enabled, time can not advance and a transition has to be taken immediately

Variables

Global/local variables

Examples:

const int $J = 10$;
$int[0, J]$ jobs; // integer with min. value 0 and max. value J

The domain of integer is always bounded. By default *int* has range $[-32768, 32768]$.

Variable assigned with a value outside of its domain generates "run-time error".

Transition can have guards: expressions which use variables. Every transition have guard, by default they return **true**

Expressions with variables

All are C-like:

- jobs++
- jobs = jobs + 1
- jobs := jobs + 1

Verification

Whereas the Verifier has an option to compute the fastest execution leading to a certain state, there is no corresponding option to compute the slowest execution.

Queries (part I)

(!)temporal logic formula
Query is a property that may or may not hold for a given model.
The Verifier can establish whether a Query is **satisfied** or **not**.
Notation:

- $A[]$ - for all reachable states...
- $E <>$ - there exists a reachable state...

Simple queries are in form of $A[]e$ and $E <> e$ where e is an expression build from **boolean combination** of **atomic propositions**.
E¡¿ p: there exists a path where p eventually holds. 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. p –¿ q: whenever p holds q will eventually hold.

Queries (part II)

Atomic propositions can be of the form $A.c$, for A an automaton and c a location. Such a proposition is true in a global state of the model if in this state automaton A 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**

Query examples

- $A[]$ 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

(!)The tool just uses brute force to explore all the reachable global states of the model and to check for each of these states whether both jobbers are working on a hard job.
(!)Model checker engine. It can run in server mode on a more powerful, dedicated machine.
(!) 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].
(!)Then we can ask the verifier to check reachability properties, i.e., if a certain state is reachable or not. This is called model checking and it is basically an exhaustive search that covers

Diagnostic traces

(!)Uppaal can also provide a concrete example that illustrates 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 didnt find anything.

(!)We choose under *Options* the entry *Diagnostic Trace* and then select the option *Shortest*. Simulator will then show found diagnostic trace.

Keeping model manageable

To keep a model manageable, one has to pay attention to some points: The number of clocks has an important impact on the complexity, i.e., on the verification time, since it highly influences the state space. The use of committed locations can reduce significantly the state space, but one has to be careful with this feature because it can possibly take away relevant states. The number of variables plays an important role as well and more importantly their range. One should be careful that the integer will not use all the values from -32000 to 32000 for example. In particular avoid unbounded loops on integers since the values will then span over the full range.

Saving data

We can save various types of data:

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

- 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.