Uppaal 4.0 : Small Tutorial*

16 November 2009

1 Introduction

This document is intended to be used by newcomers to Uppaal and verification. Students or engineers with little background in formal methods should be able to use Uppaal for practical purposes after this tutorial.

Section 2 describes Uppaal and Section 3 is the tutorial itself.

2 Uppaal

Uppaal is a tool box for validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems. It consists of two main parts: a graphical user interface and a model-checker engine. The user interface is implemented in Java and is executed on the users work station. Version 4.0 of Uppaal requires that the Java Runtime Environment 5 or higher is installed on the computer. The engine part is by default executed on the same computer as the user interface, but can also run on a more powerful server.

The idea is to model a system using timed automata, simulate it and then verify properties on it. Timed automata are finite state machines with time (clocks). A system consists of a network of processes that are composed of locations. Transitions between these locations define how the system behaves. The simulation step consists of running the system interactively to check that it works as intended. 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 all possible dynamic behaviours of the system.

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].

3 Learning Uppaal

Uppaal is based on timed automata, that is finite state machine with clocks. The 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 is composed of concurrent processes, each of them modeled as an automaton. The automaton has a set of locations. Transitions are used to change location. To control when to take a transition (to "fire" it), it is possible to have a guard and a synchronization. A guard is a condition on the variables and the clocks saying when the transition is enabled. The synchronization mechanism in Uppaal is a hand-shaking synchronization: two processes take a

*This description covers version 4.0.7

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.

The following examples will make you familiar with this short description.

3.1 Overview

The UPPAAL main window (Figure 1) has two main parts: the menu and the tabs.

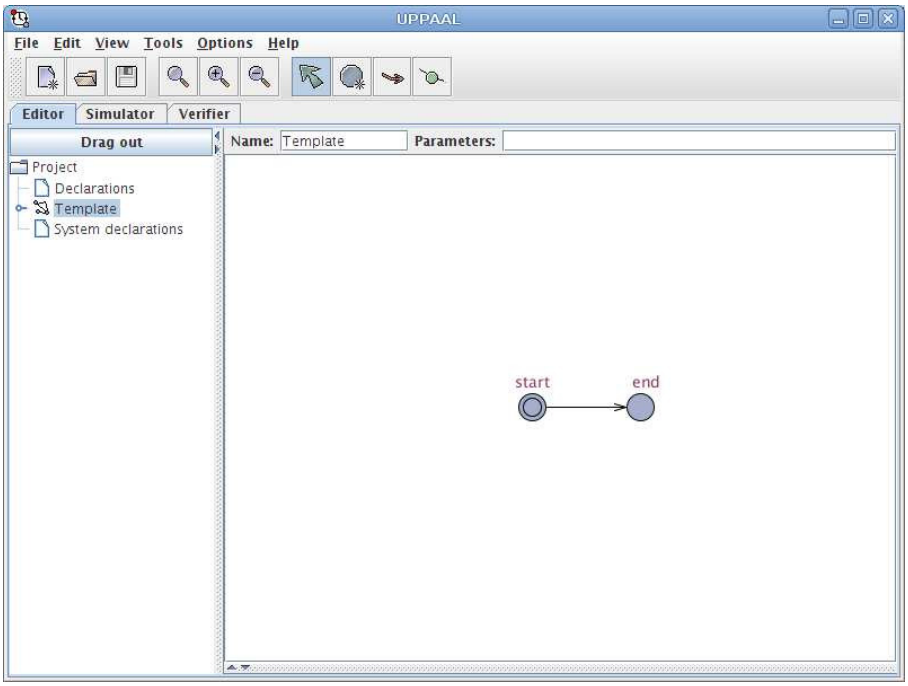

Figure 1: Overview of UPPAAL.

The menu is described in the integrated help, accessible through the `help` menu. The help further describes the used syntax and the GUI in detail, so this tutorial will focus on how to use the tool. The three tabs give access to the three components of UPPAAL that are the *editor*, the *simulator* and the *verifier*.

Figure 1 shows the editor view. The idea is to define templates (like in C++) for processes that are instantiated to have a complete system. The motivation for the templates is that system often have several processes that are very alike. The control structure (i.e., the locations and edges) is the same, only some constant or variable is different. Therefore templates can have symbolic variables and constants as parameters. A template may also have have local variables and clocks.

When you start UPPAAL, there is one location pre-created in the drawing area. It is the initial location of the automaton, so it has an additional circle inside. To add a second location, click on the "Add Location" button in the tool bar (tool tips will help you) and then click in the drawing area, a bit next to the initial location. Use the "Selection Tool" to give them the names `start` and `end`. Then choose the `Add Edge` button, click on the start location and on the end location. You have your first automaton ready, as depicted in Figure 2.

There is another important detail in the editor which you should know about. On the bottom of the screen, there is a bar which you can "drag up". This will open a table with rows *Position* and *Description* which is most likely empty at that moment. Later, it will contain very useful information about syntactical compile errors that your model contains. Using this will make it much easier to find and fix them.

Figure 2: Your first automaton.

Now, click on the `Simulator` tab to start the simulator, click on the `yes` button that will pop up and you are ready to run your first system.

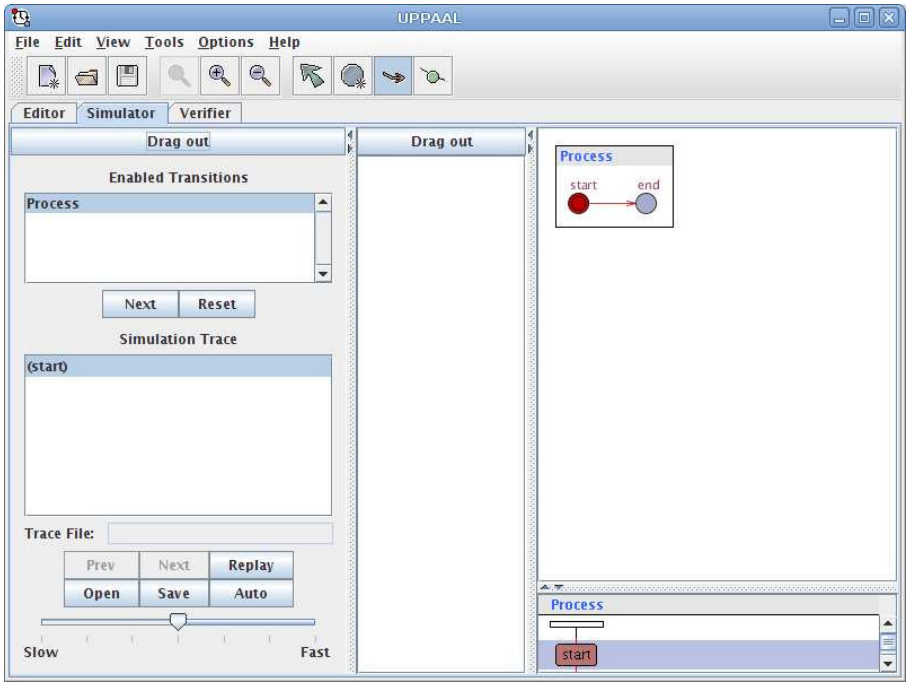

Figure 3: A snapshot of the graphical simulator.

Figure 3 shows the simulator view. On the left you will find the control part where you can choose the transitions (upper part) and work on an existing trace (lower part). In the middle are the variables and on the right you see the system itself. Below the system, you will see what happens in which process.

To simulate our trivial system pick one of the enabled transitions in the list in the upper left part of the screen. Of course there is only one transition in our example. Click `Next`. The process view to the right will change (the red dot indicating the current location will move) and the simulation trace will grow. You will note that more transitions are actually not possible, so the system is deadlocked.

We have now simulated our system and will proceed with verification. Click on the `Verifier` tab. The verifier view as in Figure 4 is displayed. The upper section allows you to specify queries to the system. The lower part logs the communication with the model-checking engine.

Enter the text `E<>Process.end` in the *Query* field below the Overview. This is the UPPAAL

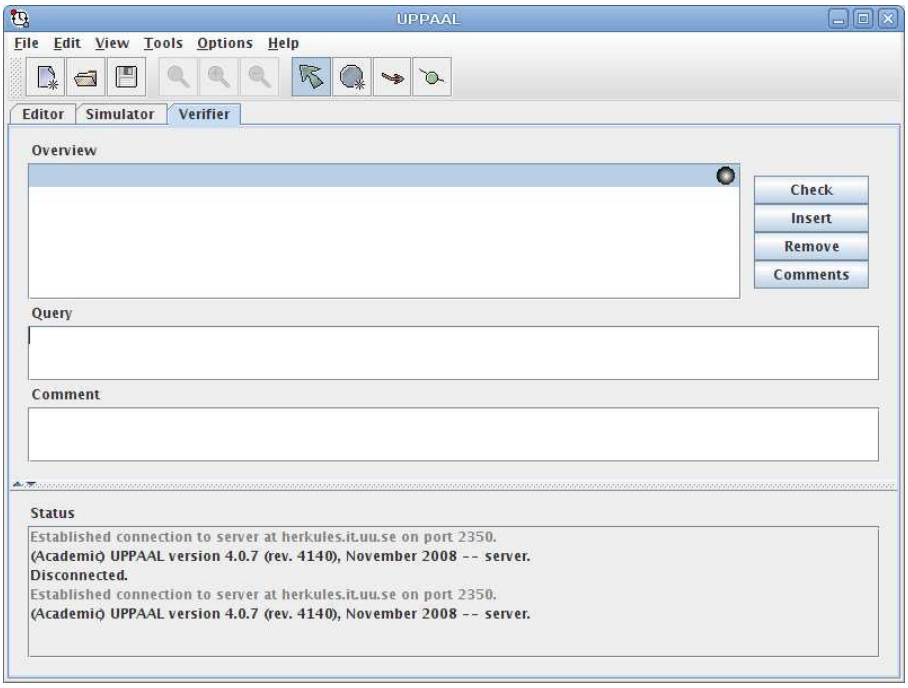

Figure 4: A snapshot of the verifier view.

notation for the temporal logic formula $\exists\diamond Process.end$ and should be understood as "it is possible to reach the location `end` in automaton `Process`". Click `Check` to let the engine verify this. The bullet in the overview will turn green indicating that he property indeed is satisfied.

The goal of the rest of this document is to explore some key points of UPPAAL through examples.

3.2 Mutual Exclusion Algorithm

We will study now the known Petterson's mutual exclusion algorithm to see how we can derive a model as an automaton from a program/algorithm and check properties related to it.

The algorithm for two processes is as follows in C:

Process 1	Process 2
`req1=1;`	`req2=1;`
`turn=2;`	`turn=1;`
`while(turn!=1 && req2!=0);`	`while(turn!=2 && req1!=0);`
// critical section:	// critical section:
`job1();`	`job2();`
`req1=0;`	`req2=0;`

You will construct the corresponding automata. Notice that the protocol is symmetric, so we may use a *template* of UPPAAL to simplify the model. First reset the system (`New system`) to clear the "Hello World" example. Rename the default template `P` to `mutex`.

We will abstract the actual work in the critical section since it has no interest here. The protocol has four states that come directly from the described algorithm, similar to goto labels. Using these, both processes can be written as follows:

Process 1	Process 2
idle:	**idle:**
`req1=1;`	`req2=1;`
want:	**want:**
`turn=2;`	`turn=1;`
wait:	**wait:**
`while(turn!=1 && req2!=0);`	`while(turn!=2 && req1!=0);`
CS:	**CS:**
//critical section	//critical section
`job1();`	`job2();`
`req1=0;`	`req2=0;`
//and return to idle	//and return to idle

As automata, the processes will look as depicted in Figure 5.

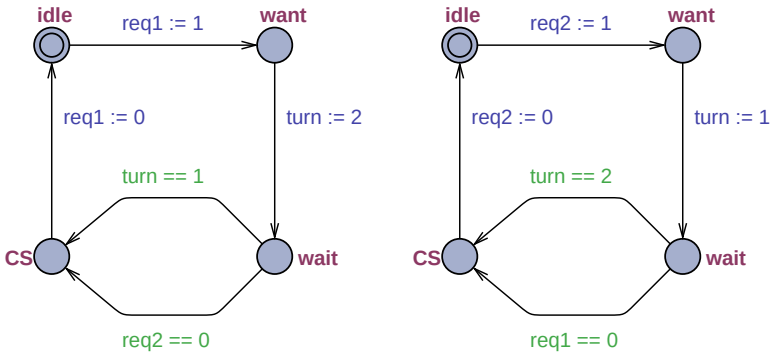

Figure 5: Mutex automata

The blue expressions are *assignments* to variables, that are executed when the corresponding transition is taken. The green expressions are *guards* that have to be true in order for the corresponding transition to be enabled.

You may notice that the two automata look very alike. The only difference are certain values and variable names. In order to save work, we will define just one template automaton, and then instantiate this template twice. The template may look as in Figure 6. Note that we replace the variables `req1` and `req2` with placeholders `req_self` and `req_other` which we will instantiate later. The new variable `me` will indicate to the instance, which process it represents.

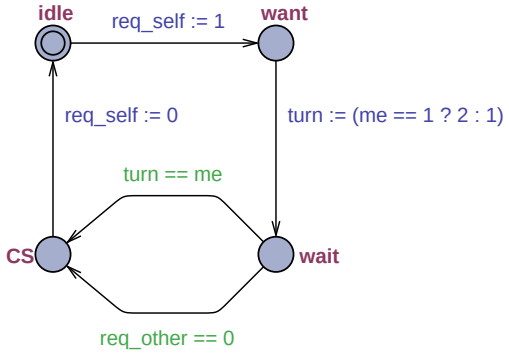

Figure 6: Mutex template

In order to create a template, you first draw the automaton from Figure 5 as before. Let's

call this template "mutex", which you specify in the *Name:* field right above the drawing area. In addition, you must specify the template parameters me, req_self and req_other. You do this by writing them in the *Parameters:* field right next to the *Name:* field. Write the following:

```
const int[1,2] me, int[0,1] &req_self, int[0,1] &req_other
```

This means that you define three variables for instantiation of type integer, bounded to two values each. The first one, me, is a constant, and the other two have to be variables with boolean values (0 or 1).

As you can guess now from your drawing, two instances of the type `P1 = mutex(1, req1, req2);` and `P2 = mutex(2, req2, req1);` will do the job. Examine how the expression (like C syntax) `turn := (me == 1 ? 2 : 1)` will evaluate. To create the instances open the `System declarations` label in the `Project` tree and type the declarations above (replacing the default line that instantiates the default template). In addition, specify that the system now consists of `P1` and `P2` by writing them behind the keyword `system`, comma-separated. The `System declarations` contain now:

```
// Place template instantiations here.
P1 = mutex(1, req1, req2);
P2 = mutex(2, req2, req1);


// List one or more processes to be composed into a system.
system P1, P2;
```

Something is still missing: the variables also have to be declared. Click on the `Declarations` label and declare: `int[0,1] req1,req2;` and `int[1,2] turn;`

Now you have defined your template, instantiated it twice, used the instantiations in the system and declared proper variables. As you may notice, the variables declared are global. This is used for `turn` that is shared. The scope of the name declaration are local first and then global. Before we go on, do a syntax check (*Tools* menu or press `F7`) and have a look in the list that you can drag up below the drawing area. It should be empty, otherwise fix the reported issues.

Now, click on the `Simulator` tab and examine how the two automata were instantiated. Look particularly at the names of the two automata that are symmetric. You can simulate your system by choosing interactively the transitions. Try to reach the critical section in both processes at the same time ... well you cannot, that's the point of the protocol. But using simulation, we can't be sure about this. A better idea is to use the verifier.

Click on the `Verifier` tab, click on the `Insert` button, click in the `Query` text area and write the mutual exclusion property: `A[] not (P1.CS and P2.CS)`. Press the `Check` button and you are done. There should be a green button lighted on, which means that the property was verified. If the button were red it would mean that the property was not verified. The property `A[]` is a safety property: you check that `not (P1.CS and P2.CS)` is always true. Another type of property, the `E<>` may be used for reachability properties. For example insert a new property `E<> P1.CS`, that checks if process P1 may reach the critical section.

If the system was not correct UPPAAL can return an diagnostic trace. First change the model so it is faulty. E.g. change the guard `req_other == 0` to `req_other == 1`. Then go to the `Options` menu and activate one of the `Diagnostic Trace` options; select the mutual exclusion property, then press the `Check` button. Now this property should not be satisfied. You can now return to the simulator and go through the found trace. Choose the first entry in the trace and then press `Replay` for this.

You have now modeled, simulated and verified a simple mutual exclusion protocol. In the demo folder in the distribution directory there are a few other simple examples. For example the file `fischer.xml` contains another mutual exclusion protocol, together with some example queries in `fischer.q`.

3.3 Time in UPPAAL

This sub-section intends to explain intuitively the concept of time in UPPAAL.

The time model in UPPAAL is continuous time. Technically, it is implemented as regions and the states are thus symbolic, which means that at a state we do not have any concrete value for the time, but rather differences [AD94]. To grasp how the time is handled in UPPAAL we will study a simple example. We will use an *observer* to show the differences. Normally, an observer is an add-on automaton in charge of detecting events without perturbing the observed system. In our case the reset of the clock (`x := 0`) is delegated to the observer to make it work. Note that by doing this, the original behaviour of the system – with the clock reset directly on the transition `loop` to itself – is not changed.

Figure 7 shows the first model with its observer. Time is used through *clocks*. In the example, `x` is a clock declared as `clock x;` in the global `Declarations` section. A channel `reset` is used for synchronization with the observer, which is also declared (as `chan reset`) in the `Declarations` section. The channel synchronization is a hand-shaking between `reset!` and `reset?` in our example. So in this example the clock may be reset after 2 time units. The observer detects this and actually performs the reset.

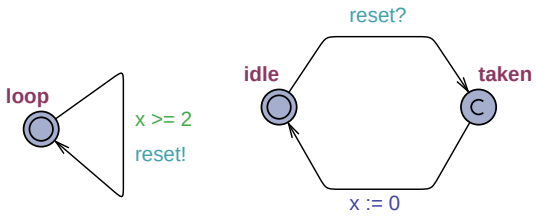

Figure 7: First example with the automata `P1` and `Obs` observer.

Draw the model, name the automata `P1` and `Obs`, define them in the system. A new template is created with *Insert Template* in the *Edit* menu. (You can just write the template names directly in the `system` statement, UPPAAL will automatically instantiate them for you.) Notice that the state `taken` of the observer is of type `committed`. If you simulate the system you will not see much. To train to interpret what you see we will use queries and modify the system progressively. The expected behaviour of our system is depicted in Figure 8.

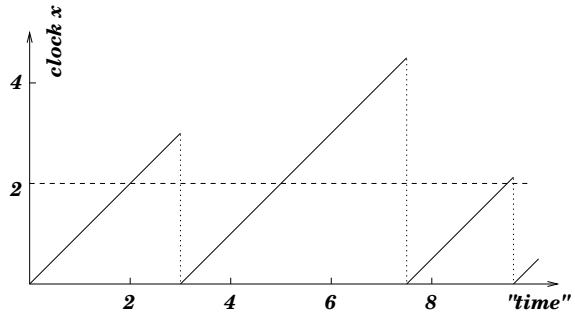

Figure 8: Time behaviour of the first example: this is one possible run.

Try these properties to exhibit this behaviour:

- `A[] Obs.taken imply x>=2` : all fall-down of the clock value (see curve) are above 2. This query means: for all states, being in the location `Obs.taken` implies that `x>=2`.

- `E<> Obs.idle and x>3` : this is for the waiting period, you can try values like 30000 and you will get the same result. This question means: is it possible to reach a state where `Obs` is in the location `idle` and `x>3`.

Add now an invariant to the `loop` location as shown in Figure 9.

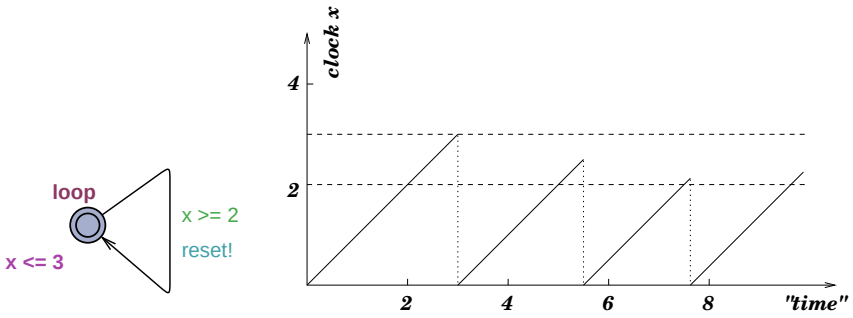

Figure 9: Adding an invariant: the new behaviour.

The invariant is a *progress condition*: the system is not allowed to stay in the state more than 3 time units (more precisely: only as long as the clock `x` is no larger than 3), so the transition has to be taken and the clock reset in our example.

To see the difference, try the properties:

- `A[] Obs.taken imply (x>=2 and x<=3)` to show that the transition is taken when `x` is in the interval $[2, 3]$.
- `E<> Obs.idle and x>2` : it is possible to take the transition with `x` in the interval $(2, 3]$.
- `A[] Obs.idle imply x<=3` : to show that the upper bound is respected.

The former property `E<> Obs.idle and x>3` no longer holds.

Now, remove the invariant and change the guard to `x >= 2 and x <= 3`. You may think that it is the same as before – but it is not! The system has no progress condition anymore, just a new condition on the guard now. Figure 10 shows the new system.

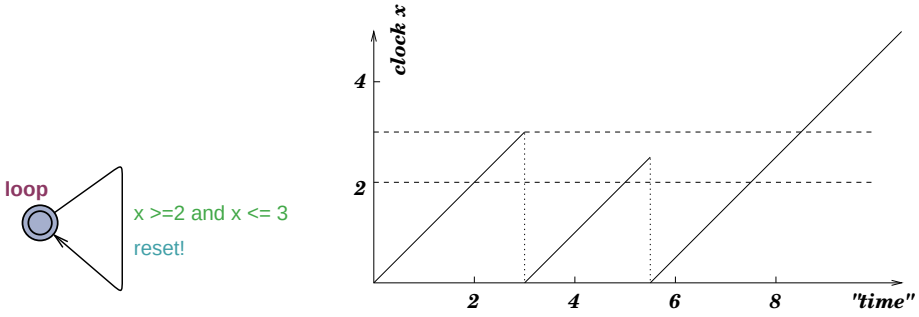

Figure 10: No invariant and a new guard: the new behaviour.

As you can see the system may take the same transitions as before, but there is now a deadlock: the system may be stuck if it does not take the transition after 3 time units. Retry the same properties, the last one does not hold now. Actually you can see the deadlock with the following property: `A[] x>3 imply not Obs.taken`, that is after 3 time units the transition can not be taken any more.

3.4 Urgent/Committed Locations

We will now look at the different kinds of locations of UPPAAL. You already saw the type `committed` in the previous example. There are three different types of locations in UPPAAL:

Normal locations with or without invariants (like `x <= 3` above),

Urgent locations and

Committed locations as used in the `Obs` automaton.

Draw the automata depicted in Figure 11 and name them `P0`, `P1` and `P2`. Define the clocks `x` locally for each automaton, in order to try this feature: open the sub-tree of their templates in the `Project` tree. There you will see a `Declarations` label under the template in question. Click on it and define `clock x;`. Repeat for the other two automata.

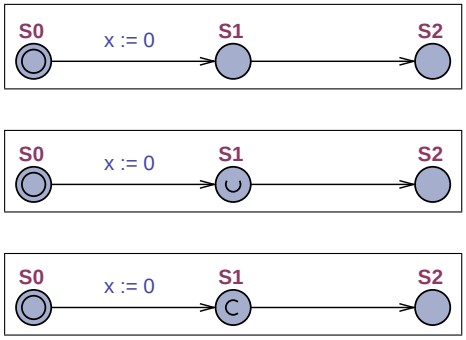

Figure 11: Automata with normal, urgent and committed states.

The location marked "U" is *urgent* and the one marked "C" is *committed*. Try them in the simulator and notice that when in the committed state, the only possible transition is always the one going out of the committed state. The committed state has to be left immediately. To see the difference between normal and urgent state, go to the verifier and try the properties:

- `E<> P0.S1 and P0.x>0` : it is possible to wait in `S1` of `P0`.
- `A[] P1.S1 imply P1.x==0` : it is not possible to wait in `S1` of `P1`.

Time may not pass in an urgent state, but interleavings with normal states are allowed as you can see in the simulator. Thus, urgent locations are "less strict" variants than committed ones.

3.5 Verifying properties

In the examples above we have used the verifier several times. We will now give a more complete treatment of the language that the verifier understands. In summary, the queries available in the verifier are:

- `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.

where p and q are state formulas like for example `(P1.cs and x<3)`. The full grammar of the query language is available in the on-line help. Note the useful special form `A[] not deadlock` that checks for deadlocks.

3.6 Some Modeling Tricks

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. It is possible to encode "urgent transitions" with a guard on a variable, i.e. busy wait on a variable, by using urgent channels. Use a dummy process with one state looping with one transition `read!`. The urgent transition will be `x>0 read?` for example.

There is no value passing through the channels but this is easily encoded by shared variable: define globally a variable `x`, and use it for reading and writing. Notice that it is not clean to do `read! x:=3;` and `read? y:=x;` but it is better to use a committed location in between: First the `read?` transition into the committed location, then the transition setting `y:=x;`.

Since UPPAAL 4, broadcast communication is also possible: The intuition is that 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. More details are given in the online help.

Arrays of integers may be useful, declare them as `int a[3];` to have an array indexable from 0 to 2. The index can be another variable `i` typically `int[0,2] i;` to be clean.

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.

References

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7*th *International Conference on Formal Description Techniques*, 1994.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88, August 1995.

[JLS96] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of 2*nd *International Workshop on the SPIN Verification System*, pages 1–20, August 1996.

[LPY97] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller: an Industrial Case Study using UPPAAL. In preparation, 1997.

[AD94] R. Alur and D. Dill. A Theory for Timed Automata In *Theoretical Computer Science*, volume 125, pages 183–235, 1994.

Version history

March 2001 First version by Alexandre David.

28 Apr 2001 Corrections by Alexandre David. Bug in a requirement, added: chan declaration, bug in declarations: int[0,1] req1,req2, turn; turn is int, not int[0,1]!

17 Dec 2001 Updates by Alexandre David. Added how to mark initial states (because the new UPPAAL does not make the first state initial by default anymore).

16 0ct 2002 Updates by Tobias Amnell. Changed screen-shoots to recent version (3.2.11), added verification walk-through in start-end example, added section on query language plus text updates on several places.

16 Nov 2009 Updates by Martin Stigge. Adapted document to version 4.0.7 of UPPAAL (syntax and screenshots). More detailed version of the mutex example to decrease confusion. A bunch of minor fixes and clarifications, thanks to Pontus Ekberg.