Modelling tips for Times*

18 September 2003

This document will help newcomers to Times to get started with modelling and verification. It is assumed that you have access to the User Manual[1].

Learning TIMES

Times is based on *timed automata with tasks*, that is finite state machine extended with clocks, data variables and executable tasks. A system in Times is composed of concurrent *processes*, each modelled as an automaton. An automaton has a set of locations connected with edges. A state of the system is determined by the current location of each process and the values of the clocks and data variables. A transition from one state to the next follows an edge in one or two processes and updates the variables according to the assignment label(s) of the edge(s).

To control when to fire a transition, it is possible to have guards and synchronizations. A guard is a condition on the variables and the clocks saying when a transition is enabled. The synchronization mechanism in Times is a hand-shaking synchronization: two processes take a transition at the same time. One of them will have a `a!` and the other a `a?`, `a` being the synchronization channel. When taking a transition actions are possible: assignment of variables or reset of clocks.

Clocks are the way to handle time in Times. 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 all clocks in the whole system.

Tasks A task in Times is an executable piece of code with known parameters, such as worst case execution time and priority. A task performs a computation or interacts with the hardware and has a well known behaviour.

Templates Each process in a system is an instance of a *template*. The motivation for the templates is that systems often have several processes that are very alike. The control structure, that is the locations, edges and most of the guards and assignments, is the same only some constants or variables are different. Therefor templates can have symbolic variables and constants as parameters. A template may also have local variables and clocks.

The rest of this document explores some key points of Times through examples.

Mutual Exclusion Algorithm

As our first exercise, we study 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:

*This document is based on Alexandre DAVID:s Uppaal2k: Small Tutorial

[1]The User Manual is available at `http:\\www.timestool.com`

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;`

Notice that the protocol is symmetric, so we may use a *template* of Times to simplify the model. On our way towards a model of the algorithm we also observe that the algorithm has four states, we mark them with a notation similar to goto labels:

Process 1

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

We are only interested in the control structure, not in the work done in the critical section, so we abstract that part of the algorithm. Now, create a new template called mutex and draw the automaton depicted in Figure 1.

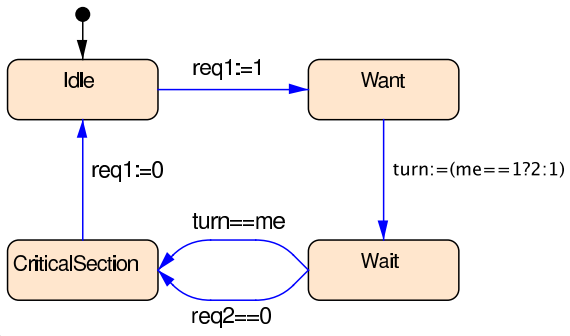

Figure 1: Mutex template

The template should have three parameters, two integers bounded between 0 and 1 and one constant. Enter them by typing `int[0,1] req1; int[0,1] req2; const me` in the Parameters field.

As you may guess from your drawing we need two instances of the template. Especially examine how the expression (using C syntax) `turn := (me==1?2:1)` evaluates when the templates are instantiated. To create the instances open the `Project` tab and add two processes and choose the template to create them from. Name the first process P1 and give it the parameters `req1, req2, 1` and name the second P2 with parameters `req2, req1, 2`.

At this point something is still missing: the variables, they have to be declared in the `Project` tab. Right-click in the `Global declarations` panel and add declarations for: `int[0,1] req1`,`int[0,1] req2` and `int[1,2] turn`.

You have now modelled the algorithm by defining templates, instantiating them and declaring variables. Now choose `Simulation` in the `Run` menu. You can simulate your system by choosing an enabled transition and clicking the "Step forward" button. Try to reach the critical section in both processes at the same time ... well you cannot, you may also use the verifier to make sure of this.

Verification Choose `Verification...` in the `Run` menu. Enter the mutual exclusion property: `A[] not (P1.Critical_Section and P2.Critical_Section)` and press the `OK` button.[2] The verifier should answer with 'SATISFIED'. If the answer were 'NOT SATISFIED' it would mean that the property does not hold. The property `A[]` is a safety property: you check that `not (P1.Critical_Section and P2.Critical_Section)` is always true. Another type of property, the `E<>` may be used for reachability properties. For example enter a new property `E<> P1.Critical_Section`, that checks if process P1 may reach the location corresponding to the critical section.

Diagnostic traces If the system was not correct Times can return a diagnostic trace. First change the model so it is faulty. For example change the guard `req2==0` to `req2==1`. Then choose `Configuration...` in the `Options` menu and make sure that the the check-box `Generate Trace` is checked on the `Verifier` tab. Now run the mutual exclusion property again. This time it should not be satisfied and you will get a dialog window with an option to show the trace, click the "Show Trace"-button to go to the simulator. You can now examine the found trace.

Tasks and Scheduling in TIMES

To continue our presentation of Times we now discuss how to use tasks. A task in Times can have either *periodic* or *controlled* arrival patterns. A periodic task is simply added to the task table and the period is given. For controlled tasks we use automata to control when the task should be released. Each location in an automaton may have an associated task . This means that when the location is entered the associated task is released to the task queue.

Times let us choose a scheduling strategy for the task queue, for example *fixed priority scheduling*, *earliest deadline first* or *rate monotonic*. The scheduling strategy is used by Times when it performs simulation, verification and schedulability analysis.

As an exercises we will create a simple automaton controlling the releases of two tasks. First add two *controlled tasks* P and Q to the task table. Set their parameters as in the table below:

Name	Computation time	Deadline
P	3	10
Q	4	7

Create a new template and draw an automaton like the one shown in Figure 2. The automaton will control the release of the tasks. Initially task P is released, then after 3 time units the automaton may proceed to Location 2 where task Q is released. Since task P has computation time 3 the queue is empty when task Q is released. In Location 2 the automaton may after 5 time units loop back and release task Q again, or return to Location 1 and release task P.

We can study the behaviour in the simulator where the Message Sequence Chart (MSC) view help us see how the automaton executes. The MSC will include two extra automata: SCHEDULER and PERIODIC_TASKS. These are created automatically by Times based on the selected scheduling strategy to control the execution of the released tasks.

When we simulate we will also see new locations named REL_P_1 and REL_Q_1 in the process we created (Process_1). These are auxiliary locations introduced by Times to handle the release of tasks.

[2] Note that spaces in names of locations and processes are replaces by underscores (_).

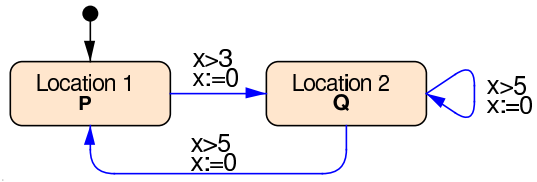

Figure 2: Automaton controlling tasks P and Q.

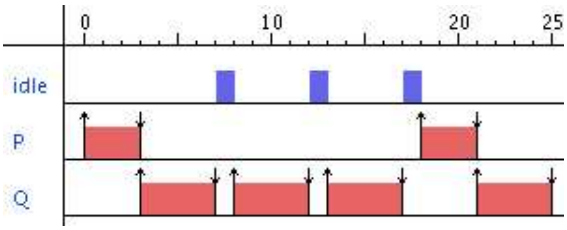

Figure 3: One possible schedule for the system in Figure 2 with EDF scheduling strategy.

In the panel below the MSC we can see how the tasks are scheduled in a Gantt chart. One possible schedule where earliest deadline first (EDF) is used as scheduling strategy is shown in Figure 3.

To verify that the model is schedulable we can use one of the main features of Times, namely schedulability analysis. Choose the scheduling strategy you want to use in the task table and choose `Schedulability analysis` in the `Run` menu. Times will answer 'SATISFIED', this simple example is schedulable with all the available scheduling strategies.

Time in TIMES

We continue with a more in-depth discussion of the concept of time in Times. We will explain the concepts through examples. You are encouraged to draw the examples and make your own experiments.

The time model in Times is continuous time. Technically, it is implemented as zones and the states are thus symbolic, which means that in a state we do not have concrete values of the clocks, but rather intervals [AD94]. For example the time in a state could be defined as that clock x is between 2 and 4 and clock y is between 7 and ∞.

To understand how time is handled in Times we will study the simple automaton shown in Figure 4(a). We also use the *observer* shown in Figure 4(b) for these experiments. 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 so that all steps in the behaviour is accessible for simulation and verification. The original behaviour of the simple automaton where the reset is on the loop is not changed by adding the observer.

Time is accessed through the *clock* x. A channel `reset` is used for synchronization with the observer. The channel synchronization is a hand-shaking between `reset!` and `reset?`. When the clock reaches 2 in our example the automaton may synchronise with the observer that performs the reset of the clock.

Draw the model, name the automata P1 and Obs and instantiate them in the project tab. Notice that the state `Taken` of the observer is of type `committed`. Declare the channel with `chan reset;` in the global variables section. If you only simulate the system you may not detect all the important details of the behaviour. Instead we will use queries to the verifier and progressively modify the system. The expected behaviour of our system is depicted in Figure 5.

(a) Simple automaton (b) Observer

Figure 4: Simple timed automata and observer

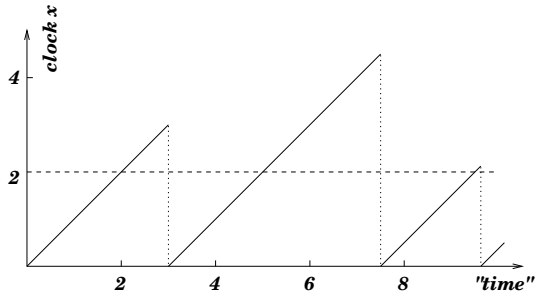

Figure 5: Time behaviour of first example: this is one possible run.

Try these properties to exhibit this behaviour:

- `A[] Obs.Taken imply x>=2` : all fall-downs 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`.

From these queries we learn a very important fact. If a guard is enabled it does not mean that the transition have to be taken. That is we must explicitly tell the system that is should make progress!

To ensure progress we may use invariants. Add the invariant `x<=3` to the `Loop` location as shown in Figure 6. The invariant is a progress condition: the system is not allowed to stay in the location when the invariant is false, so the transition have to be taken and the clock reset in our example.

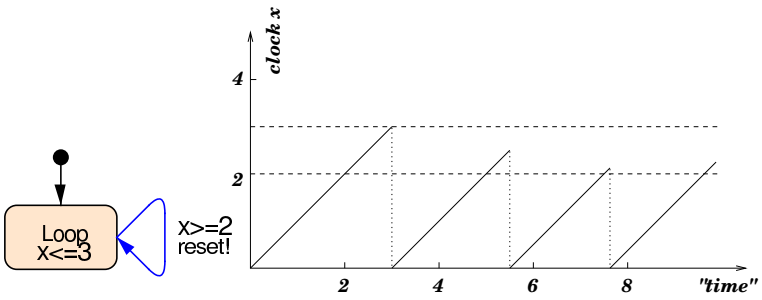

Figure 6: Adding an invariant: the new behaviour.

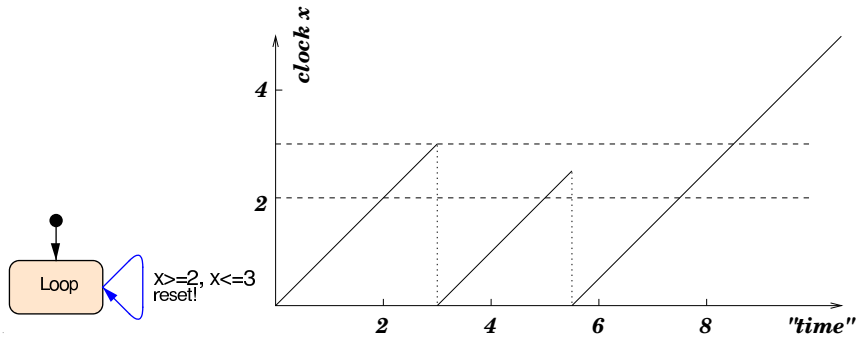

Figure 7: No invariant and new guard.

To see the difference from before, try the properties:

- `A[] Obs.Taken imply (x>=2 and x<=3)` to show that the transition is taken when in the interval 2-3.
- `E<> Obs.Idle and x>2` : it is possible to take the transition 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.

Remove the invariant and change the guard to `x>=2, x<=3`. You may think that it is the same as before but it is not! The system has no progress condition, just a new condition on the guard. Figure 7 shows the new system.

As you can see the system may take the same transitions as before, but there is now a possible deadlock: the system may get stuck if it does not take the transition within 3 time units.

To see what happens retry the same properties, the last one does not hold now. You can see the deadlock with the following property: `A[] x>3 imply not Obs.Taken`, that is after 3 time units the transition is not taken any more. Another alternative is to use the property `A[] not deadlock`, that is not satisfied either.

Urgent/Committed Locations

We will now look at the different kinds of locations of Times. You already saw the type `committed` in the previous example. There are three different types of locations in Times: normal locations with or without invariants, urgent locations and committed locations. Draw the automata depicted in Figure 8 in three different templates. Define the clocks x locally in each template.

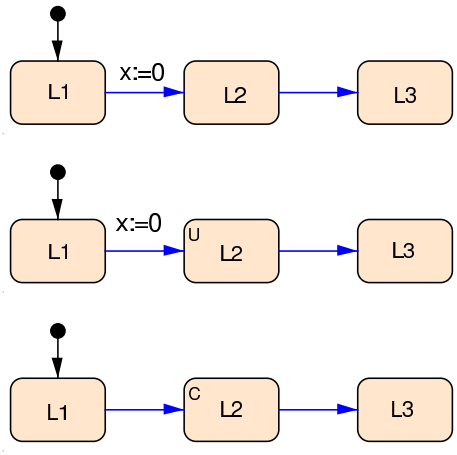

Figure 8: Automata with normal, urgent and committed locations.

Name the automata Pnormal, Purgent and Pcommited respectively. The state marked U is urgent and the one marked C is committed. Try them in the simulator and notice that when in the commit location, the only possible transition is always the one going out of the committed location. The committed state has to be left immediately. To see the difference between normal and urgent states, use the verifier to try these properties:

- `E<> Pnormal.L1 and Pnormal.x>0` : it is possible to wait in `L1`.
- `A[] Purgent.L1 imply Purgent.x==0` : it is not possible to wait in `L1`.

Time may not pass in an urgent state, but interleavings with normal states are allowed as you can see in the simulator.

Verification properties

In the examples above we have used the verifier several times. We have only used safety properties (`A[]`) and reachability properties (`E<>`). While these are the most useful types of properties that Times provides, the tool also understands a couple of others. In summary, the queries available in the verifier are:

- `E<> p`: there exists a path where p eventually hold.
- `A[] p`: for all paths p always hold.
- `E[] p`: there exists a path where p always hold.
- `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 of the form: `(P1.cs and x<3)` and a path is a sequence of transitions in the system. The full grammar of the query language is available in the User Manual. Note the useful special form `A[] not deadlock` that checks for deadlocks.

Some Modeling Tricks

Finally we have collected some useful modelling tricks that can make your life as a Times-user much easier.

Urgent channels and urgent transitions Times offers *urgent channels* 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" by using urgent channels. Use a dummy process with one location and a loop labelled with the urgent channel `go!`. To make a transition urgent in another automaton, add a synchronisation on `go?` as shown in Figure9.

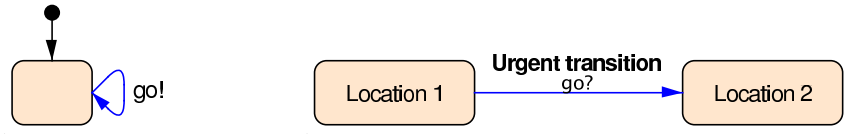

Figure 9: Encoding urgent transitions using urgent channels

Value passing in synchronisations Times does not directly support value passing in synchronisations, but this is easily encoded using a shared variable: define a global variable x, and use it to write and read the passed value as in Figure 10. Notice that it is not clean to use `read! x:=3;` and `read? y:=x;` on a single edge, even though it works. Instead it is better to use an intermediate commit location as in Figure 10.

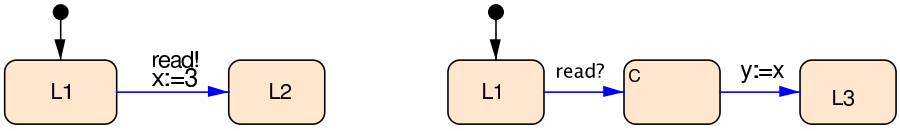

Figure 10: Value passing using global variable x.

Broadcast and multicast The current version of Times does not support broadcast or multicast communication: synchronization is only between pairs of automata. Instead, you can encode multicast using a series of committed locations. The sending automaton would have a sequence of locations like in Figure 11 and each receiving automaton an edge with the corresponding `go1?`, `go2?` and `go3?`. Several solutions are possible.

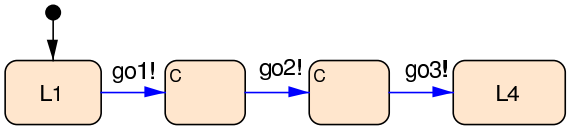

Figure 11: Multicast.

Large state-spaces Verifying a large model can take a long time and use up large amounts of memory. To keep a model manageable, one has to pay attention to some points:

- The number of clocks has an important impact on the complexity, so use as few as possible.
- The use of committed locations can significantly reduce the state space. But you have to be careful not to take away relevant interleavings of states.
- The number of variables plays an important role as well, and more importantly their range. Whenever possible you should define a range for the variables as we have done in the examples above. In particular, avoid unbounded loops on integers since the values will then span over the full range.

Periodic tasks A tasks can be declared to be periodic with a period T in the task table. Such tasks cannot be released by automata but are handled by the automatically created process PERIODIC_TASKS. Another way to declare periodic tasks is to declare the task as controlled and create an automaton like the one shown in Figure 12.

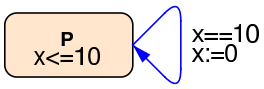

Figure 12: Controlled periodic task.

Sporadic task Another type of semi-periodic tasks where the period varies within a range cannot be declared in the task table. Instead we can use an automaton to control the task. Assume that a task arrives non-deterministically with an interval between 10 and 20 time units. An automaton that releases the task Q in this fashion is shown in Figure .

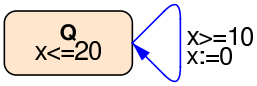

Figure 13: Controlled semi-periodic task.

Locked locations In some cases a task must finish before the controller can continue. This is for example the case when the task reads a sensor value that the controller uses. The following trick makes use of the task interface to lock the controller in a location until the task finishes.

Declare a global boolean variable `int[0,1] TaskDone` and edit the interface of the task to set the variable to `TaskDone:=1`. Then we can lock the controller to the location until R is finished by adding the labels to the edges leading out of the location as shown in Figure 14. Note that the transition is made urgent using the trick with an urgent channel `go?` presented above.

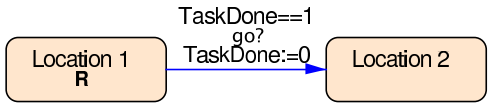

Figure 14: Automaton locked until task R is done.

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* 2nd *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. Dill and D.L. Dill. A Theory for Timed Automata In *Theoretical Computer Science*, volume 125, pages 183–235, 1994.