Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group
Critical Embedded Systems

Short UPPAAL introduction

András Vörös
Based on the document written by Dániel Darvas and Gergő Horányi

13. October 2013

1 Introduction

This short summary serves as a short introduction to the modelling in UPPAAL framework. This summary is based on the tutorial written by Behrmann, David and Larsen [1]. This summary is neither complete nor a mathematically precise introduction, the definitions and descriptions are simplified for the sake of lucidity. The motivation of this tutorial is to give a short introduction to the concepts of modelling with UPPAAL and make the first steps easier.

1.1 *About UPPAAL*

UPPAAL[1] is a framework for the modelling and analysis of real time systems. UPPAAL is developed at the Uppsala University and Aalborg University. The first release came out in 1995, the newest version is 4.0.

2 Modelling in UPPAAL

We can model networks of timed automata in UPPPAAL, where a timed automaton is a finite automaton (finite state machine) extended with clock variables. The first part of the section overviews the finite automaton formalism and then we introduce also the extensions.

2.1 *Finite automata in UPPAAL*

The networks of timed automata in UPPAAL consist of concurrent processes, each one of them represented by a finite automaton. The description of the automaton of the single process is similar to the description of the State Machine in UML (we are going to compare the two formalisms is some examples).

The basic building blocks of the (single) state machine models are:

- locations (states): they have a name, only one of them is active at a certain time point
- arcs, edges (transitions, state changes): they represent the possible state changes of the system, guards and actions can be assigned to them

The following figure depicts a simple state machine model.

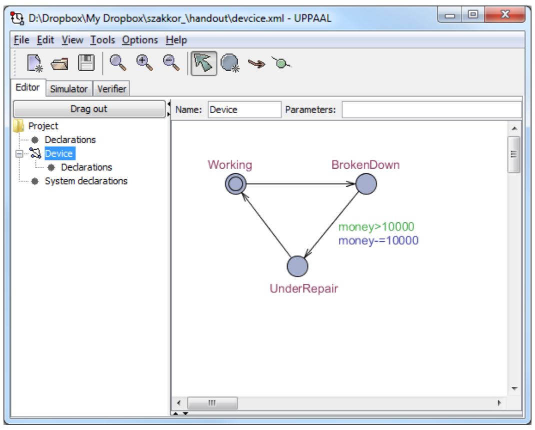


Figure 1. A state machine in UPPAAL

In Figure 2. and Figure 3. we compare the state machine models in the formalisms of UML and UPPAAL. The state machine model is a simple device, which can be in the following states: *working*, *broken down* or it can be *under repair*. The state machine can change its state: a working system can become wrong i.e. it breaks down. Then faulty system can go under repair,

[1] UPPAAL can be downloaded from the following address for academic use: http://www.uppaal.org/.

and hopefully it becomes working again. The arcs (arrows) show this evolution of the system. However, there are state changes which are not possible: for example we do not want to repair a working system, so the models do not contain an arc in with this direction.

Remark: More than one enabled transition can go out from one state. This is a natural way to model nondeterminism of the system, in this case we choose nondeterministically the next state we go into. This concept is depicted in Figure 4. where the system can go to the working state from the repair state, or in the case it could not be repaired now, it may return to the *BrokenDown* state.

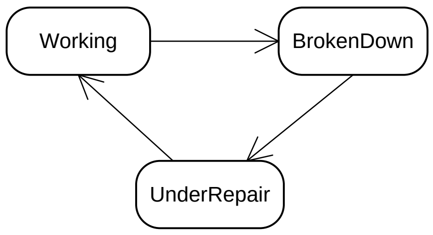


Figure 2. Simple UML State Chart model

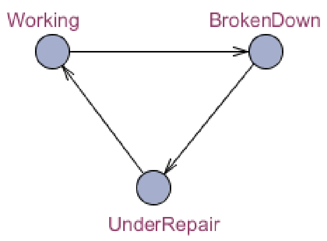


Figure 3. Simple UPPAAL state machine

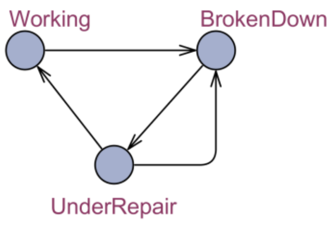


Figure 4. Nondeterministic model with multiple outgoing edges from *UnderRepair*

There is a special state where the state machine starts its behaviour: this state is the initial state. In UPPAAL it is signed with a double circle (Figure 6.) We can set this state by double clicking on the state and choosing the *Initial* checkbox in the window (Figure 7.).

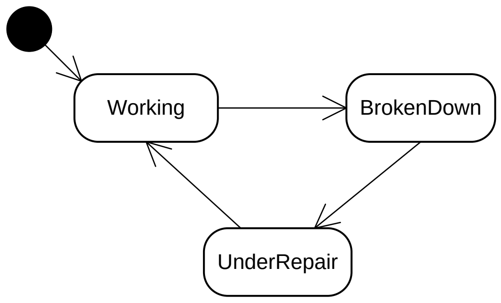


Figure 5. Simple UML State Chart model with initial state

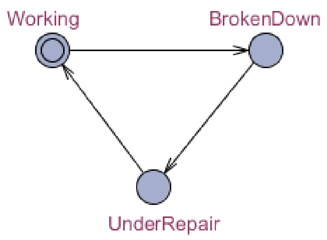
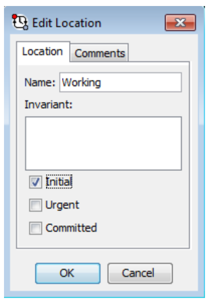


Figure 6. Simple UPPAAL state machine with initial state



Name: Working
Invariant:
[✓ Initial]
Urgent
Committed

Figure 7. Setting the initial state

We can add conditions to the transitions, for example we can restrict the repair by enabling the arc going to the repair state (*UnderRepair*) only if we have enough money. Otherwise this transition is not enabled and our system remains broken down. In the following example in Figure 9. the repair can start only if we have more than 10000 money. In UPPAAL we can add conditions to the arcs by assigning guards to them: these guards can express conditions on the formerly defined variables of the model. The guard specified in the example enables the transition only if "`money>10000`" and it can be added by double clicking on the arc representing the transition. The window where the addition of a guard to an arc can be defined can be seen on Figure 10.

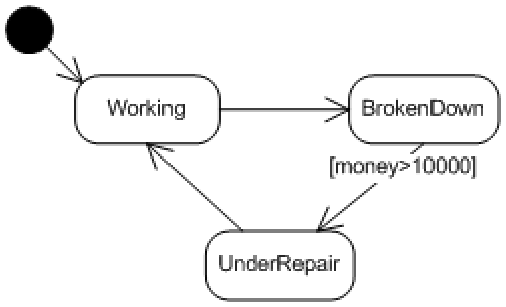


Figure 8. UML State Chart model with guard

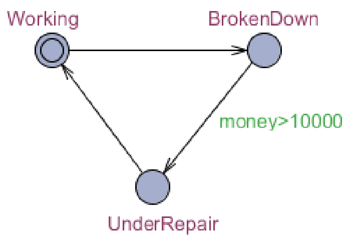
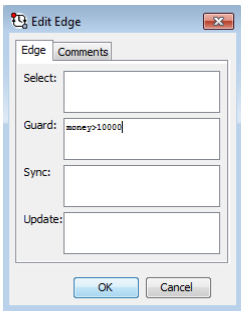


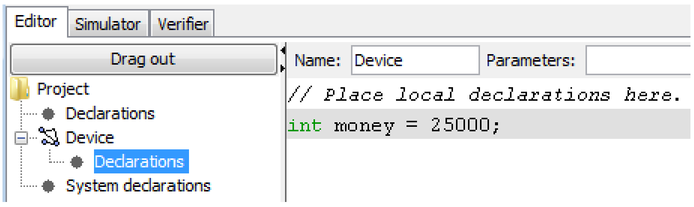
Figure 9. UPPAAL state machine model with guard



Guard: money>10000

Figure 10. Setting a guard

The variables used in UPPAAL have to be declared: a declaration is either local or global. The variables used in a guard have to be declared locally in the "*Declarations"* of the state machine model or globally in the "*Project*". In Figure 11. a local integer variable is declared in the *"Device"* template.



Name: Device

```
// Place local declarations here.
int money = 25000;
```

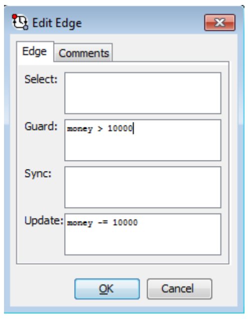
Figure 11. Local declaration

The declaration of variables is expressed in a C like syntax, in the example:

```
int money = 25000;
```

This document does not discuss the syntax of the declarations further, the proper definitions can be found in [3] or in the UPPAAL help.

Guards can restrict the possible state changes by disabling transitions, however we can also extend the possible transitions. We can form more complex transitions by assigning actions to the transitions: the action and the transition are executed together. Let us now consider our example: the repair can only be started if we paid the cost, so we reduce the available amount of money (that is the money variable) by 10000. Actions can be defined by double clicking on the arc and setting the "*Update*" parameter of the window (Figure 12.).



Select:
Guard: money > 10000
Sync:
Update: money -= 10000

Figure 12. Assigning action to a transition

In the example the action decreases the money variable by 10000:

```
money -= 10000
```

In the following figures we depict the State Chart model and the corresponding UPPAAL model with an action.

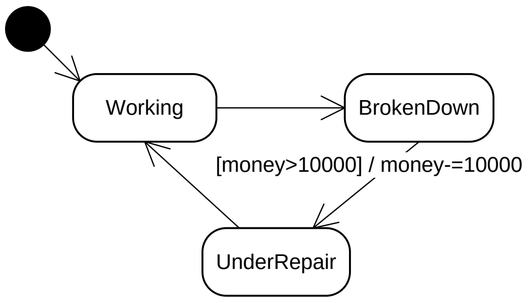


Figure 13. UML State Chart model with action

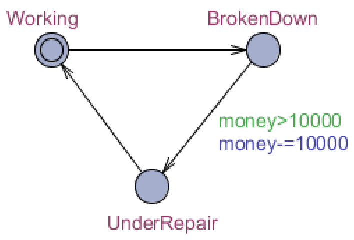
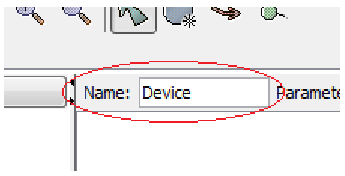


Figure 14. UPPAAL model with action

2.2 *Simulation in UPPAAL*

UPPAAL has not only modelling, but it has also good simulation capabilities. At first we have to point out the main differences between state machines in UML State Chart and UPPAAL. In UML we define state machine instances, while in UPPAAL we have only defined the type of the "*Device*" as a state machine: it is just a template. However, there is still no working instance of this template (of this type of state machine). We can only simulate instantiated systems in UPPAAL.

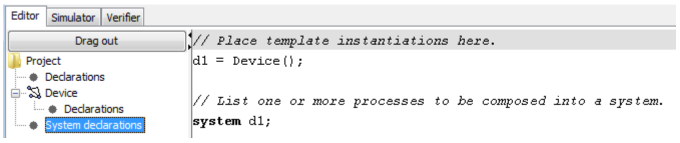
At first we have to name our template as it can be seen in the next figure (note that in the former examples the template had already been named):



Name: Device

Figure 15. Setting the name of a template

After setting the name of the template, choose *System declarations* from the tree view at the left side! Templates can be instantiated here (see Figure 16.).



```
// Place template instantiations here.
d1 = Device();

// List one or more processes to be composed into a system.
system d1;
```

Figure 16. Instantiation of Template “Device”

In order to create a d1 instance from the “*Device*” template, we have to write the following code:

```
// Place template instantiations here.
d1 = Device();

// List one or more processes to be composed into a system.
system d1;
```

Now we can try simulating our system and visualize its behaviour. We have to choose the *Simulator* tab depicted on Figure 17. UPPAAL asks the question if you want to refresh the model („Do you want to upload the model now?”) in the simulator after each modification (of the model). Choose *Yes* to work always with the actual model.

Now we can see the d1 instance of the “*Device*” template. The location coloured red is the actual state of the state machine. In the middle of the window (left from the state machine) the actual values of the variables are depicted. By clicking on the button “*Next”* we can step into the next state of the state machine. We can also choose automatic stepping also by clicking on the “*Auto”* button (signed with red on Figure 17.). This way we can see that the “*Device”* stops working after a while according to the decrease of the money variable. The model goes into deadlock.

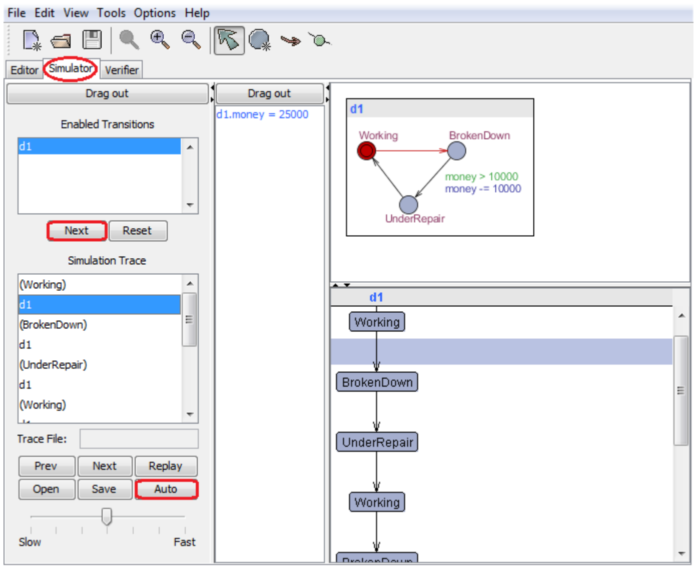


Figure 17. Simulator of UPPAAL

2.3 *Timing*

UPPAAL is able to handle models of real time systems and provides modelling and analysis support for them. Clock variables represent timing information in UPPAAL, which can be declared similarly to other variables. A clock variable represents logically a clock: time dependant behaviour can be added to the system by reading and setting these (clock) variables. Usually we do not know the exact values of the clocks but we can compare them to constants and examine if a certain time out has already happened: in general we know the time interval of the clock.

Important properties of clock variables:

- The values of the clock variables increase monotonically or we can reset them explicitly
- The system may stay in a state for arbitrarily long (but finite) time or it can step further immediately (staying in a state for 0 time unit). We can use invariants, guards and special states (Urgent, Committed) to control the time spent in a state.
    - Consequence: It is possible that at the beginning only one outgoing arc (transition) is enabled from a state, but as time elapses new transitions become enabled and one of them brings the model to the next state.

The declaration of clock variables is similar to other variables. In order to declare the clock variable `cl` we have to use the following code:

```
clock cl;
```

Now we can express conditions on the clock variable in the guards and we can reset clock variable in actions of the transitions.

We are going to use a simple example to introduce the concept of using clock variables (Figure 18.). The action of transition A→B sets the value of clock variable `cl` to 0. So we can easily infer that in state B the value of the clock variable is: $cl \geq 0$.

Stepping forward to the next location (next state) does not necessarily happens immediately when a transition becomes enabled. When at least 2 time units elapsed in state B then the clock variable equals or is greater than 2 ($cl \geq 2$), transition B→F becomes enabled. However, nothing forces here the system to go on, so it can stay is state B as long transition B→C becomes also enabled, and system steps to the state C. When actual state of the system is location C, then the value of the clock variable is: $cl \geq 3$. We do not know the exact value of the clock in this state as the transition may happen in any time when the following holds: $cl \geq 3$, for example we can reach state C when for example $cl = 4$ or $cl = 40$.

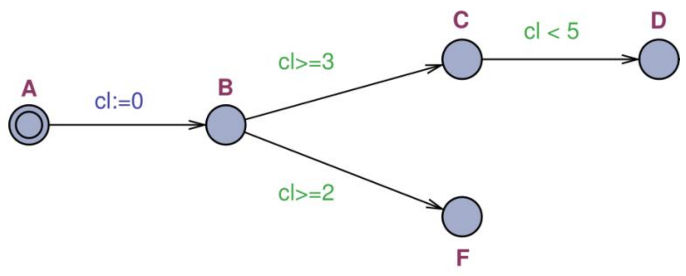


Figure 18. Clock variable example

There are 2 possible behaviours from state C in the model. When the value of the clock variable is $3 \leq cl < 5$, then the model can execute the state change C→D so finally it reaches location D. However, as the transitions can happen nondeterministically, the model may stay in state C until reaching $cl = 5$. Then it is stuck, the model reaches a deadlock and cannot leave state C.

If transition C→D was executed and we reach state D, then what we can claim that the value of the clock variable: $cl \geq 3$. We might point out that the transition could happen when the value of `cl` is between 3 and 5, as far as the model may stay arbitrary long time in state D, we cannot claim any upper bound for the value of `cl` here.

Sometimes we also need a mechanism to force the model to stay in a certain location for at most some formerly defined amount of time. Invariants can be defined for the states in order to maximize the time spent there. For example in Figure 19. we defined an invariant in location B:

cl < 4. This invariant means that the model can stay in location B only when the clock variable `cl` is smaller than 4, in our case at most for 4 time units (because we reset the clock on the incoming arc). So the behaviour of the model is modified in a way that it cannot spend arbitrarily long time in state B.

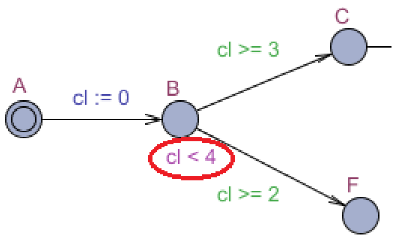


Figure 19. Invariant example

2.4 Synchronisation

UPPAAL is used to model and analyse networks of timed automata, so we can define multiple processes as finite state machines and we are interested in how they collaborate, do they have correct behaviour together. In order to be able to model and examine the collaboration, UPPAAL introduces synchronisation operators.

There are two different types of synchronisation in UPPAAL: synchronisation on *simple* channel or on *broadcast* channel. Both synchronisations require the declaration of the *channel* of the synchronisation: message sending is realized on these channels. (There are two different kinds of simple channels: *regular* and *urgent* channels, in this document we are only dealing with regular channels)

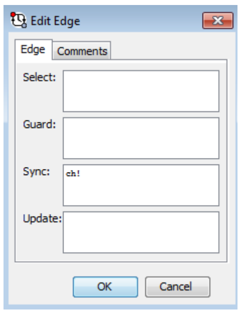
A simple synchronisation channel `ch` can be defined with the following line of code:

```
chan ch;
```

Declaration of a broadcast channel requires the "broadcast" keyword:

```
broadcast chan bch;
```

After the declaration of the channel we can use them: we can add synchronisation operation to the arcs (transitions) we want to synchronise by double clicking on the edge and setting the "*Sync*" parameter (see Figure 20.). We can define sending or receiving operation on an edge. Sending on the channel `ch` is defined by using `ch!` as the synchronisation operator (for example on Figure 20.) and the reception on channel `ch` is declared by setting `ch?`.



Select:
Guard:
Sync: ch!
Update:

Figure 20. Setting the Sync parameter of an edge

There is a synchronisation on a single channel `ch` only if there is a process (state machine) with an actual state from where there is an enabled outgoing edge (enabled transition) on which `ch!`

is set, and there is an other process with an enabled transition on which `ch?` is set. This is depicted on Figure 21. In the frames there are parts (locations) of the different processes, the red locations are the actual states of the processes. As far as the process in the left frame is able to send a synchronisation message and the other process can receive it, the synchronisation is enabled and both transitions are executed together. However, if there were not a synchronisation message sending transition or a receiver transition, then the synchronisation would be disabled and the transitions are also disabled.

If there are multiple receivers on the channel, the synchronisation is executed only with one of them (chosen randomly).

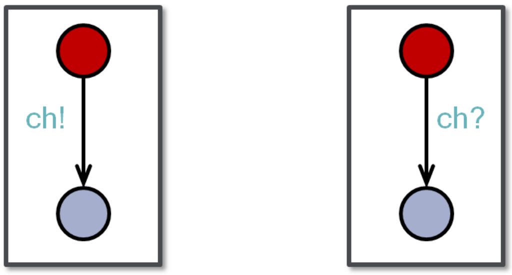


Figure 21. Simple synchronisation example

Broadcast synchronisation happens between one sender and multiple (0..*) receiver(s). The receiver behaves similarly to the simple synchronisation (if the transition is enabled and there is a synchronisation message, the transition can fire). However there is a difference from the sender point: sender can execute the transition with synchronisation if there are multiple receivers and all of the receiver processes execute the synchronisation transition.

References for preparation

[1] Gerd Behrmann, Alexandre David, and Kim G. Larse: A Tutorial on Uppaal 4.0
http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf

[2] UPPAAL2k: Small Tutorial
http://www.it.uu.se/research/group/darts/uppaal/tutorial.pdf

[3] Gerd Behrmann: Introduction to UPPAAL
http://people.cs.aau.dk/~srba/courses/SV-05/slides/l11.pdf