BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Static Analysis and Verification 2020/2021

http://fit.vut.cz

Uppaal Model Checker

Matúš Liščinský, xlisci02@stud.fit.vut.cz,
Faculty of Information Technology, Brno University of Technology

1. Introduction

Uppaal is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. The tool is designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronisation [1]. Typical application areas include real time controllers and communication protocols in particular, those where timing aspects are critical. It has been applied successfully in case studies ranging from communication protocols to multimedia applications [5].

The following description covers current stable release Uppaal 4.0, although there is available Uppaal 4.1 (development snapshot) which includes SMC (Statistical Model Checking) extension described in [3], I decided to work with the stable version.

This report consists of three main parts, the first is devoted to the description of the tool, the second part contains examples demonstration, and finally the third part is intended for own experiments with the tool.

2. About Uppaal

Before we start, it is appropriate to mention that this section is built mostly on the papers [1] and [2]. Uppaal consists of two main parts: a graphical user interface and a model-checker engine. The idea is to model a system using timed automata, simulate it and then verify properties on it.

For reasons of efficiency, the model checking algorithms that are implemented in Uppaal are based on (sets of) clock constraints[1], rather than on explicit (sets of) regions. By dealing with (disjoint) sets of clock constraints, a coarser partitioning of the (infinite) state space is obtained. Working with clock constraints allows to characterise $Sat(\varphi)$ without explicitly building the region automaton a priori [4].

[1] Here, the property that each region can be expressed by a clock constraint over the clocks involved, is exploited.

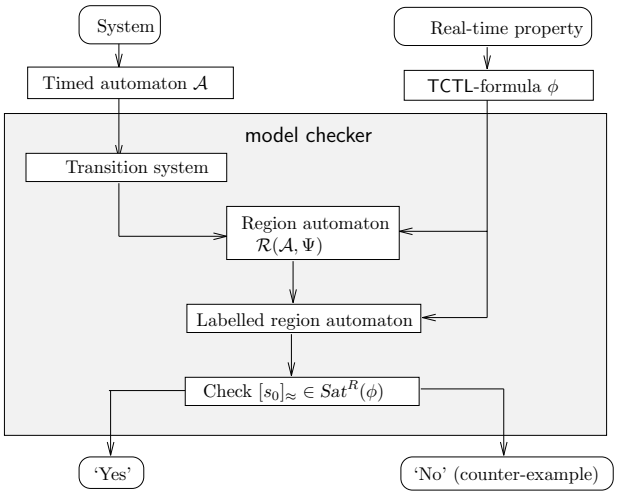


Figure 1. Overview of model checking TCTL over a timed automaton. Taken from [4].

Simulation and Verification. 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. Reachability properties are checked by exploring the state space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints. This is called model-checking and it is basically an *exhaustive* search that covers all possible dynamic behaviours of the system. The engine of model-checker uses *on-the-fly verification* combined with a *symbolic* technique reducing the verification problem to that of solving simple *constraint* systems [2]. To facilitate modelling and debugging, the Uppaal model checker may automatically generate a diagnostic trace that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model checker may be graphically visualised using the *Simulator*.

Languages. Uppaal works with *modelling* and *query* language. *Modelling* language extends timed automata with additional features such as bounded integer variables, urgency and so on. *Query* language, which is used to specify properties to be checked, is a subset of TCTL (Timed Computation Tree Logic).

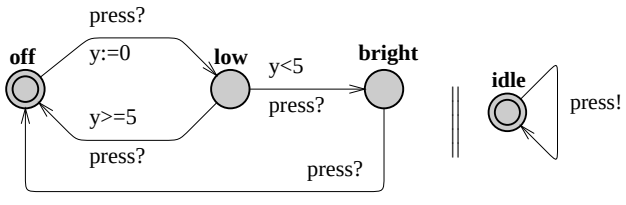
2.1 The Modelling Language

Uppaal is based on timed automata, which is a finite state machine with clock variables. A system is modelled as a network of several timed automata in parallel. The model is further extended with bounded discrete variables that are part of the state. These variables can be read, written to, and be a subject to common arithmetic operations. A state of the system is then defined by the locations of all automata, the clock values, and the values of the discrete variables.

The automaton has a set of locations and transitions, which are used to change location. To control when to take a transition, it is possible to have a *guard* and a *synchronisation*. A *guard* is a condition on the variables and the clocks, that says when the transition is enabled. The *synchronisation* mechanism in Uppaal works as a hand-shaking synchronisation: two processes take a transition at the same time if one will have an `a!` and the other an `a?`, where `a` is the synchronisation channel. In general, a synchronisation pair is chosen non-deterministically if several combinations are enabled. When taking a transition, two actions are possible: assignment of variables or reset of clocks.

Since Uppaal 4, broadcast communication is also possible. In a broadcast synchronisation, one sender `b!` can synchronise with an arbitrary number of receivers `b?`. Any receiver than can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the `b!` action, i.e. broadcast sending is not blocking.

In Figure 2 is illustrated a timed automaton modelling a simple lamp. The lamp process has three locations: `off`, `low`, and `bright`. When the user "presses a button", i.e., synchronises with `press?`, the lamp will turn on. If the user presses the button again, the lamp can either turn off or become bright. This depends on how fast will user press the button, i.e. the time difference between first and second press is crucial. This is modelled by resetting the y clock with the first press and guards $y >= 5$ and $y < 5$ on the transitions from `low` to `off` and `bright`, respectively.



(a) Lamp. (b) User.

Figure 2. Simple lamp as an example of timed automaton. Taken from [1].

The Uppaal modelling language extends timed automata with additional features, a complete list of them can be found in [1].

2.2 The Query Language

The main purpose of a model-checker is to verify the model w.r.t. a requirement specification, which must be expressed in a formally well-defined and machine readable language. Several such logics exist in the scientific literature, and Uppaal uses a simplified version of TCTL. Like in TCTL, the query language consists of path formulae and state formulae[2]. State formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be classified into three types of properties: *reachability*, *safety* and *liveness*. Each type is described below.

State Formulae. A state formula is an expression that can be evaluated for a state without looking at the behaviour of the model. The syntax of state formulae is a superset of that of guards, i.e. a state formula is an expression without side-effects. It is also possible to test whether a particular process is in a given location using an expression on the form `P.l`, where `P` is a process and `l` is a location.

Deadlock. In Uppaal, deadlock is expressed using a special state formula. The formula simply consists of the keyword `deadlock` and is satisfied for all deadlock states. A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. The deadlock state formula can only be used with reachability and invariantly path formulae (due to current limitations in Uppaal).

Reachability Properties. Reachability properties are the simplest form of properties. They ask whether a given state formula φ, possibly can be satisfied by any reachable state. Reachability properties do not guarantee the correctness of the model, but they validate its basic behaviour. We express that some state satisfying φ should be reachable using the path formula $E \diamond \varphi$. In Uppaal, we write this property using the syntax $E <> \varphi$.

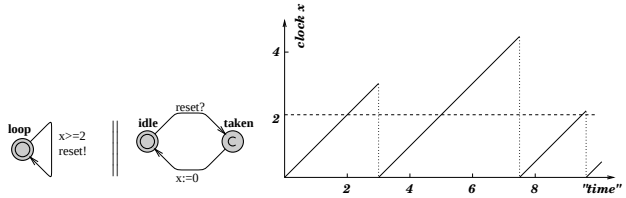
Safety Properties. Safety properties are on the form: "something bad will never happen". In Uppaal, these properties are formulated positively, e.g., "something good is invariantly true". Let φ be a state formulae. We express that φ should be true in all reachable states with the path formulae $A\Box\varphi$ (notice that $A\Box\varphi = \neg E\Box\neg\varphi$), in Uppaal we write $A[]\varphi$.

[2] In contrast to TCTL, Uppaal does not allow nesting of path formulae.

Liveness Properties. Liveness properties are of the form: "something will eventually happen". In its simple form, liveness is expressed with the path formula $A \diamond \varphi$, meaning φ is eventually satisfied. The more useful form is the *leads to* property, written $\varphi \rightsquigarrow \psi$ which is read as whenever φ is satisfied, then eventually ψ will be satisfied. In Uppaal these properties are written as $A <> \varphi$ and $\varphi --> \psi$, respectively.

2.3 Time in Uppaal

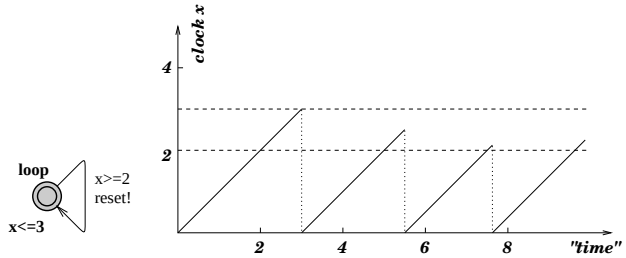
Uppaal uses a continuous time model. Technically, it is implemented as regions and the states are thus symbolic, which means that at a state we do not have any concrete value for the time, but rather differences. To understand the concept of time one can look at Figure 3, which shows a model with its observer and one possible run of this example. We have two automata in parallel. The first automaton has a self-loop guarded by $x >= 2$, x being a clock, that synchronises on the channel `reset` with the second automaton. The second automaton, the observer, detects when the self loop edge is taken with the location taken and then has an edge going back to idle that resets the clock x. So in this example the clock may be reset after 2 time units.



(a) Test. (b) Observer. (c) Behaviour: one possible run.

Figure 3. Example with an observer. The location `taken` is committed (marked c) to avoid delay in that location. Taken from [1].

Now we add an invariant $x <= 3$ to the `loop` location and look how the behaviour has changed (Figure 4). The *invariant* is a progress condition: the system is not allowed to stay in the state `loop` more than 3 time units, so that the transition has to be taken and the clock reset in our example. Now the clock x has 3 as an upper bound.

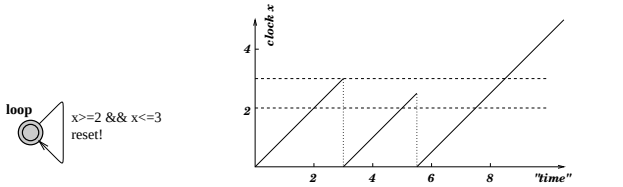


(a) Test. (b) Updated behaviour with an invariant.

Figure 4. Updated example with an invariant. The observer is the same as in Figure 3 but it is not shown here. Taken from [1].

If we remove the invariant and change the guard to $x >= 2 \wedge x <= 3$, one 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 5 shows what can happen: the system may take the same transitions as before, but deadlock may also occur. The system may be stuck if it does not take the transition after 3 time units.



(a) Test. (b) Updated behaviour with a guard and no invariant.

Figure 5. Updated example with modified guard and no invariant. Taken from [1].

3. Examples Demonstration

Firstly we may want to start easy with modelling and verifying the properties of the examples from the previous section. These examples are useful for getting to know and start working with the tool.

Example with an observer (Figure 3). To create a model, I followed the steps intended for this example from [2]. It was necessary to draw the model and set the guard, synchronisation and the clock updates on corresponding transitions. To make the model work the clock x and channel `reset` must be declared in the global *Declarations* section. If we simulate the model, we will not see much. Hence I moved to the *Verifier* to check the properties, whether:

- the system is deadlock-free (satisfied),
- it is possible to reach the state where `Observer` is in the location idle and x is bigger than 3 (satisfied),
- for all reachable states, being in the location `Observer.taken` implies that x>=2 (satisfied).

The source of this model and this queries I made available here: model, queries. The model works as we expected and described in the previous section.

Updated example with an invariant (Figure 4). The first model was updated by adding an invariant to the location `loop`. Here, I checked the properties, whether:

- the system is deadlock-free (satisfied),
- when the transition is taken, the x is between 2 and 3 (satisfied),
- it is possible to take the transition when x is greater than 2 (satisfied),
- the upper bound (invariant $x <= 3$) is respected (satisfied).

Sources to this example are online, so this queries and the model can be found here: model, queries.

Updated example with modified guard and no invariant (Figure 5). In the last modification, we remove the invariant and change the guard, what can mistakenly leads to the idea that it is the same automaton as before. We can verify it and see the difference by checking whether:

- after 3 time units the transition can not be taken anymore (satisfied),
- when the transition is taken, the x is between 2 and 3 (satisfied),
- deadlock is not possible (not satisfied).

Queries and the model are available here: model, queries. As we expected, there is a chance of occurring a deadlock, as the verification refuted that a deadlock is not possible. The provided queries file also contain a query asking for property whether there is a reachable state with deadlock ($E \diamond deadlock$), which is satisfied.

3.1 Petterson's mutual exclusion algorithm

The following example is a case study to Uppaal from [2], for demonstration how we can derive a model as an automaton from a program/algorithm and check properties related to it. We can abstract the algorithm to four states that come directly from the algorithm, similar to goto labels. Using these, both processes can be written as depicted in Figure 6.

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

Figure 6. Petterson's mutual exclusion algorithm written in C. Taken from [2].

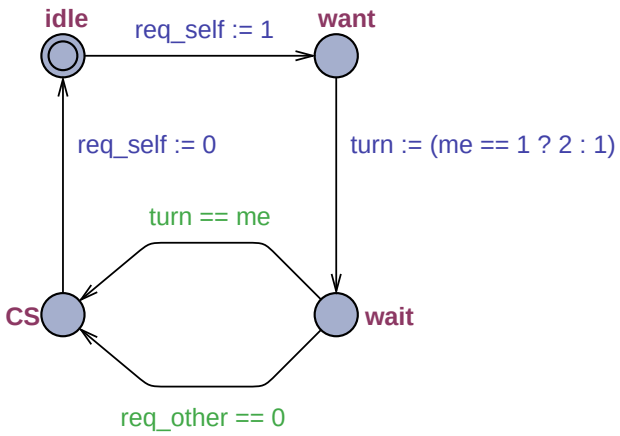


Figure 7. Petterson's mutual exclusion as timed automata in Uppaal. Taken from [2].

To construct the corresponding automata I followed the instructions which lead to implement the model using just one Uppaal *template* for both processes (because the protocol is symmetric). The resulting automaton, which we will examine is shown in the Figure 7.

Now we can move on and simulate the automata using Uppaal *Simulator*. There we can simulate the system by choosing interactively the transitions. We can also try to reach the critical section in both processes at the same time, and see that this state is not reachable. However, by using only simulation we can't be sure about this, hence a better idea is to use the *Verifier*.

There I checked few properties showing that protocol is working correctly. These properties was verified by queries, whether:

- process 1 may reach the critical section (satisfied),
- process 2 may reach the critical section (satisfied),
- deadlock is not possible (satisfied),
- mutual exclusion property holds (there is no reachable state where both processes are in the critical section, satisfied).

All the properties were satisfied, so we can be sure that modelled mutual exclusion protocol is working correctly. My source files to this example are available here: model, queries.

In the following sections we will focus on demonstration of the examples that are part of the Uppaal package (directory `uppaal/demo`).

3.2 2doors problem

The first of the demo examples will be 2doors problem. I found the assignment of this problem only in one place on the internet: lecture[3] from BRICS (Basic Research in Computer Science[4]). The basic information about the problem are the following:

- A room has 2 doors which can not be open at the same time.
- A door starts to open if its button is pushed.
- The door opens for 6 seconds, thereafter it stays open for at least 4 seconds, but no more than 8 seconds.
- The door takes 6 seconds to close and it stays closed for at least 5 seconds.

[3] Modelling and Analysing RealTime Systems using Uppaal
[4] BRICS

Problem was modelled and the model verified by checking the given properties with the following results:

- The system is deadlock-free (satisfied),
- Both doors can open (satisfied),
- Liveness property: whenever a button is pushed, the corresponding door will eventually open (satisfied),
- Mutex property: both doors are never open at the same time (satisfied),
- Bounded Liveness property: a door will open within 31 seconds (satisfied),

Although the model and queries files are part of the Uppaal package, I made them available online too: model, queries.

3.3 Bridge and torch problem

This is one of a well known logic puzzle that deals with four people (in our case), a bridge and a torch. Definition of the problem from the *Declarations* section: Four vikings are about to cross a damaged bridge in the middle of the night. The bridge can only carry two of the vikings at the time and to find the way over the bridge the vikings need to bring a torch. The vikings need 5, 10, 20 and 25 minutes (one-way) respectively to cross the bridge.

Created model contains (Viking 1..4 and Torch) in total. Viking processes models where the Vikings currently are, and the Torch process tells us how many torches are available. The question was, does a schedule exist which gets all four vikings over the bridge within 60 minutes? Before answering that, we have to verify that the system works as was initiated with checking the following properties:

- The system is deadlock-free (satisfied).
- All Vikings can cross the bridge (satisfied).
- It is not possible that slowest Viking crossed the bridge, and the global time clock is lower than his time for crossing (satisfied).
- If the slowest Viking crossed the bridge, the global time clock must be greater or equal to his time for crossing (satisfied).

To answer our question, we have to set the Diagnostic trace option in Uppaal to fastest, and then in *Verifier* check the property that all Vikings crossed the bridge. After the check, the Simulator section contains the fastest trace that leads to the state where the Vikings are safely on the other side of the bridge. This trace ends in the state, where the global time clock is greater or equal to 60. Therefore we can say, that there exist a schedule which gets all four vikings over the bridge within 60 minutes.

Again, source files are available to look at here: model, queries.

3.4 Fischer's Protocol

This is a well-known mutual exclusion protocol designed for n processes. It is a timed protocol, where the concurrent processes check for both a delay and their turn to enter the critical section using a shared variable `id`. The model of the Fisher's Protocol can be seen in Figure 8.

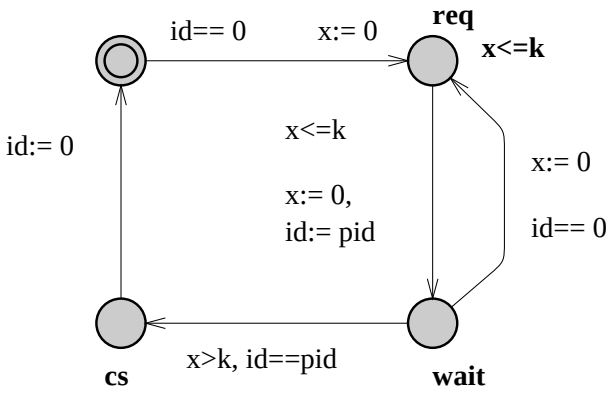


Figure 8. Template of Fischer's protocol. The parameter of the template is `const pid` (process id). Each process has its own clock and delay, so the template has the local declarations `clock x` and `const k 2` (delay). Figure taken from [1].

To confirm that the model works properly, I verified following properties:

- The system is deadlock-free (satisfied).
- Whenever a process requests access to the critical section it will eventually enter the wait state. (satisfied).
- Mutual exclusion (satisfied).
- Whenever a process requests access to the critical section it will eventually enter the critical section (not satisfied).

The last property is violated, because the process is allowed to stay in `wait` forever, thus there is a way to avoid the critical section. Sources of this example are available here: model, queries.

3.5 Other demonstration examples.

These seven demonstrated examples are not all I have tried in Uppaal. For example, there are two more demo examples (interrupt and train gate) in the demo directory of the tool, which I decided not to include in order to preserve a reasonable length.

4. Experiments

When I thought about what examples I could try in Uppaal, my first idea leads to the second assignment from the curse Model-Based Analysis (MBA) 2019/2020, which was focused on timed automata. With the experience in modelling of examples from the previous section I modelled two timed automata from this assignment and verify some interesting properties.

4.1 Timed automata with deadlock

This is the first timed automaton I chose from the assignment, because I think it is perfect to start with, since it is not so complicated and, moreover, it is suitable for demonstration of how Uppaal works. We can see the automaton in Figure 9. As one can see, in this example we use two clocks: x, y, some guards and invariant in location `A`. Simulation did not bring to us any useful information so let see what we can verify. Let us name this timed automaton simply `Process`. Firstly, we may verify if each of the automaton locations is reachable, e.g. `E<> Process.A` for location `A`. These queries were satisfied, obviously, but the the verifying of property `A[] not deadlock` was not satisfied.[5]

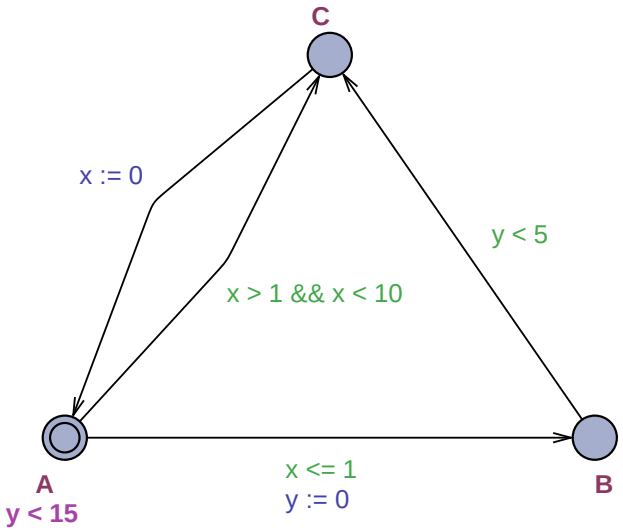


Figure 9. Timed automaton containing deadlock. Extracted from Uppaal. Source of the model can be found here.

The deadlock can occur in several states, for example:

- `Process.C and y >= 15`
- `Process.B and y > 5`
- `Process.A and x > 10`

All these states are reachable and were verified, that they imply deadlock state. All verified properties are included in the queries file here.

[5]Notice that Uppal uses <> with the semantic of diamond symbol ◇ and [] as square symbol □ in TCTL.

4.2 Timed automata of coffee machine

This example also comes from the second MBA curse assignment and contains a simple automaton that models a coffee machine. The model in Uppaal is shown in Figure 10. Before the verification part, let us name this timed automaton simply `Process` again. Complete list of verified properties can be found here, in this report I decided to mention at least some of them:

- All the locations of `Process` are reachable, i.e. properties like `E<> Process.A` ...are satisfied.
- If the automaton gets to the location `D`, it cannot recover back to the initial state, (i.e. `A[] Process.D imply not Process.A` is satisfied).
- After inserting the coins, we does not have to get the coffee, i.e. the coffee machine will get to the error state (`E<> Process.B imply Process.D` is satisfied).
- There exist a path where `Process.A` always holds, i.e. the machine will never be used (`E[] Process.A` is satisfied).[6]

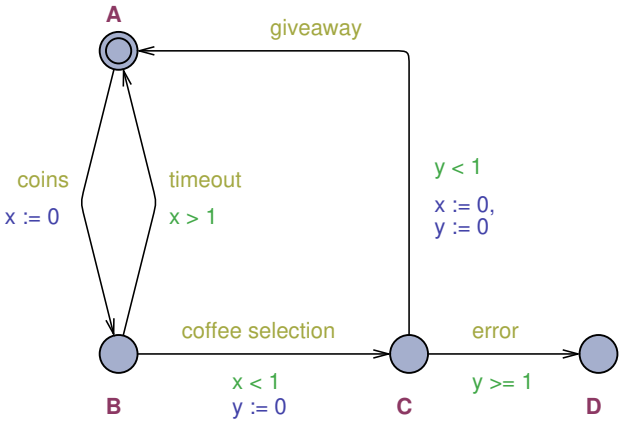


Figure 10. Timed automaton of coffee machine. Extracted from Uppaal. Source of the model can be found here.

4.3 Double deck lift

This is example I made to model the behaviour of lift with three floors, where one of them is the ground floor. The model consist of three templates: cabin, door, and controller from which five automata are created in total (cabin, controller and three doors).

Door template (Figure 11) only inform the system whether the certain doors are open or closed. Cabin (Figure 12) template is used for elevator movement control, and its guards and clocks are used to ensure that for movement between the two floors is necessary

[6]In Uppaal, $E\Box\varphi$ says that there should exist a maximal path such that φ is always true. A maximal path is a path that is either infinite or where the last state has no outgoing transitions.

a minimum of six and three time units, respectively. Controller (Figure 13) serves as a layer connecting the processes of the door and the cabin. It is used to synchronise with them and thus forms the main control part of the model. The source file of the created model containing all templates is available here.

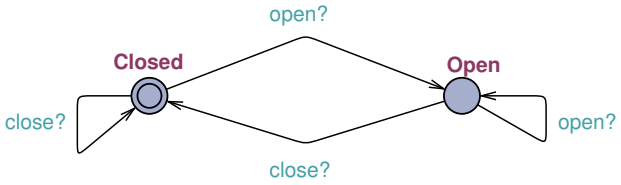


Figure 11. Template of door. Extracted from Uppaal.

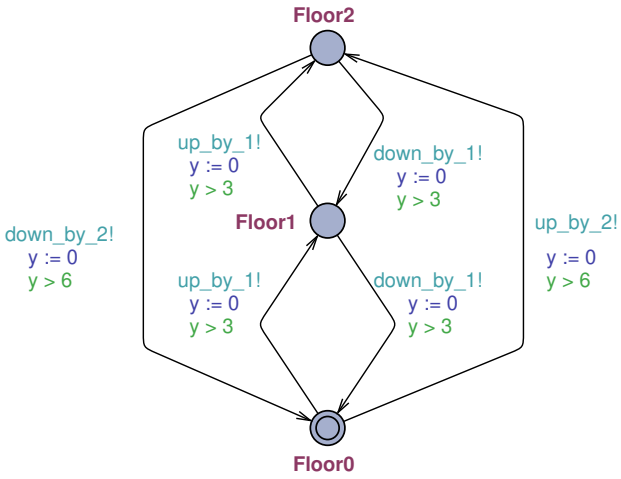


Figure 12. Template of cabin. Extracted from Uppaal.

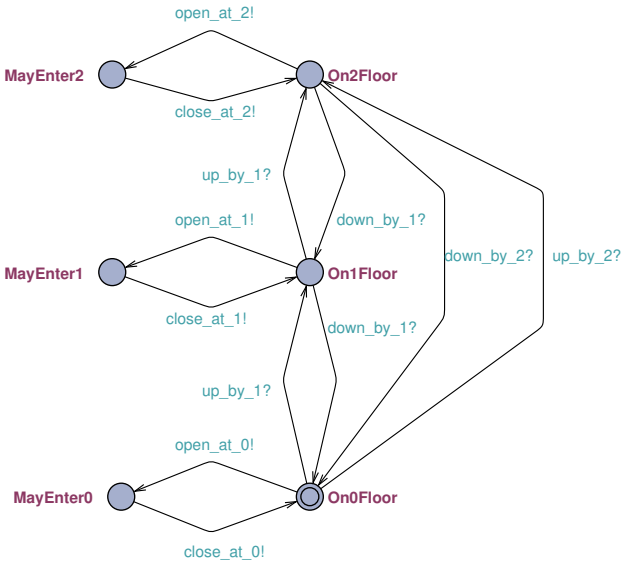


Figure 13. Template of controller. Extracted from Uppaal.

At first I verified that the system is deadlock-free, hence when modelling I came across the fact that a deadlock can be caused by a small accident, e.g. greater-than symbol instead on less-then in the guard. This system is deadlock free and all the locations of the automata are reachable, which was verified by checking `E<>Template.location` like properties. Another verified property was that it is not possible for any two doors to be open at the same time. This is actually the same property as that the doors can be either all closed or at most one open. Both properties were satisfied and are included in queries file.

What I did not mentioned yet, is that the model has one global clock `time`. We can use it to verify how much time do we need to get to second floor. This information we can obtain by setting the Diagnostic Trace in Uppaal options to fastest and checking `E<>Door2.Open` property. In the Simulator tab we can see the generated trace and that we need more than 6 time units to open the door on the second floor from the beginning. We can also check property `E<> Door2.Open and time <= 6` to see it is not satisfied, but `E<> Door2.Open and time > 6` is. The same type of property can be used for any doors of the system.

4.4 Another interesting examples.

While doing the research, I came across many articles and other sources of examples, where can be nicely utilised the functionalities of the Uppaal tool, e.g. this lecture[7] from National University of Singapore curse, examples in papers like [4], or this web-page[8] which is tackling whit concurrency problems (The Producer/-Consumer Problem, Dining Philosophers, . . .) using semaphores in Uppaal. However, many of them are fairly comprehensive, hence they were not suitable for this report. One can find all the examples mentioned in this report individually or zipped in examples.zip.

5. Conclusion

In this report, we focused on the Uppaal model-checker. This tool is specified for the verification of systems that can be modelled as networks of timed automata, extended by additional features. We started with a description of the tool, what it consists of, we approached how its model-checking algorithm works and explained the basic functions and possibilities of its use. Subsequently, we tried the tool on several demo examples and we also tried to create our own models, simulate them and verify their properties. Uppaal is free for academic use, any other use requires a commercial license. This tool has been used in many industrial case studies and helps to teach about verification of real-time systems at universities all around the world.

[7] Verification of Real Time Systems - CS5270, lecture 11
[8] Semaphores in Uppaal

References

[1] BEHRMANN, G., DAVID, A. and LARSEN, K. G. A tutorial on UPPAAL 4.0. 2006.

[2] DAVID, A., AMNELL, T. and STIGGE, M. Uppaal 4.0 : Small Tutorial. 2009.

[3] DAVID, A., LARSEN, K. G., LEGAY, A., MIKUČIONIS, M. and POULSEN, D. B. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*. Springer. 2015, vol. 17, no. 4, p. 397–415.

[4] KATOEN, J.-P. *Concepts, algorithms, and tools for model checking*. IMMD Erlangen, 1999.

[5] LARSEN, K. G., PETTERSSON, P. and YI, W. UPPAAL in a nutshell. *International journal on software tools for technology transfer*. Springer. 1997, vol. 1, 1-2, p. 134–152.