# Engineering Self-Adaptive Systems: Preparatory Task

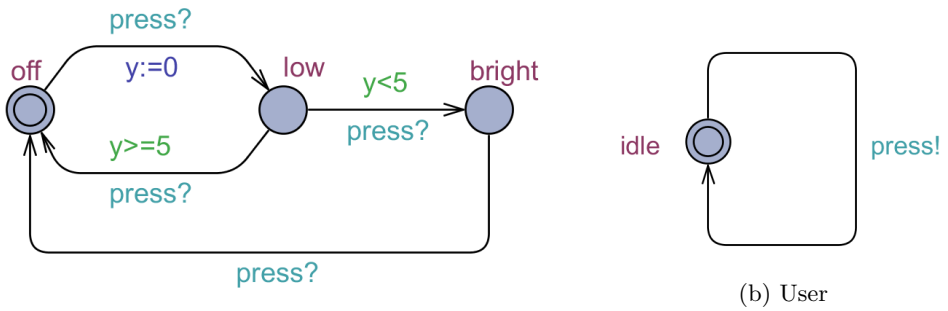
**Stefan Seifried**
**Matr.Nr.: 00925401**

# Uppaal – Lessons Learned

Uppaal [1] is an integrated tool environment for modeling, simulation, and verification of systems that can be modeled as networks of timed automata. A timed automaton is a finite-state machine extended with dense clock variables (i.e., real numbers) progressing synchronously. The state of a system is then defined by the current state of all automata, the values of the clocks, and the values of variables (e.g., integer or boolean variables). Transitions between different states either are independent or simultaneously with another automaton through the means of synchronization channels. A synchronization channel (e.g., `x`) is always comprised of a sender and a receiver where the state transition of the sender (e.g., `x!`) introduces a state transition at the receiver (e.g., `x?`). The use of transitions between different states can also be limited by the use of guard statements (e.g., guard `x>0`). These statements serve as invariants (on clock or ordinary variables) and decide whether a transition may be taken or not (i.e., transition is enabled). Further, clocks and variables may be modified upon a transition (assignments).
The locations in Uppaal, allow for more fine-grained tuning of their temporal behavior. Despite the explicitly flagged *initial* location, there are three different types of locations: *normal*, *urgent*, and *committed*. *Normal locations* may be used with or without invariants. Meaning that unless there is a restriction enforced by an invariant or a trigger by a synchronization channel, the process may stay in this location indefinitely. *Urgent locations* are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant x¡=0 on the location. Hence, time is not allowed to pass when the system is in an urgent location. *Committed locations* are even more restrictive on the execution than *urgent locations*. A state is committed if any of the locations in the state is *committed*. A *committed* state cannot delay, and the next transition must involve an outgoing edge of at least one of the *committed locations* (See Section 2.1 in [1]).

# Simple lamp

Figure 1 shows two automata modeling the simple lamp system as described in [1]. The *simple lamp* example consists of two independent processes, the *User*, and the *Lamp*. The User can control the lamp with button presses to change the state of the Lamp to either *off*, *low* or *bright*. The *Lamps bright* state can only be reached by two consecutive presses within the duration of 5-time units. The full source code of the implementation can be found at GitHub [2].




(a) Lamp

(b) User

Figure 1: The simple lamp example

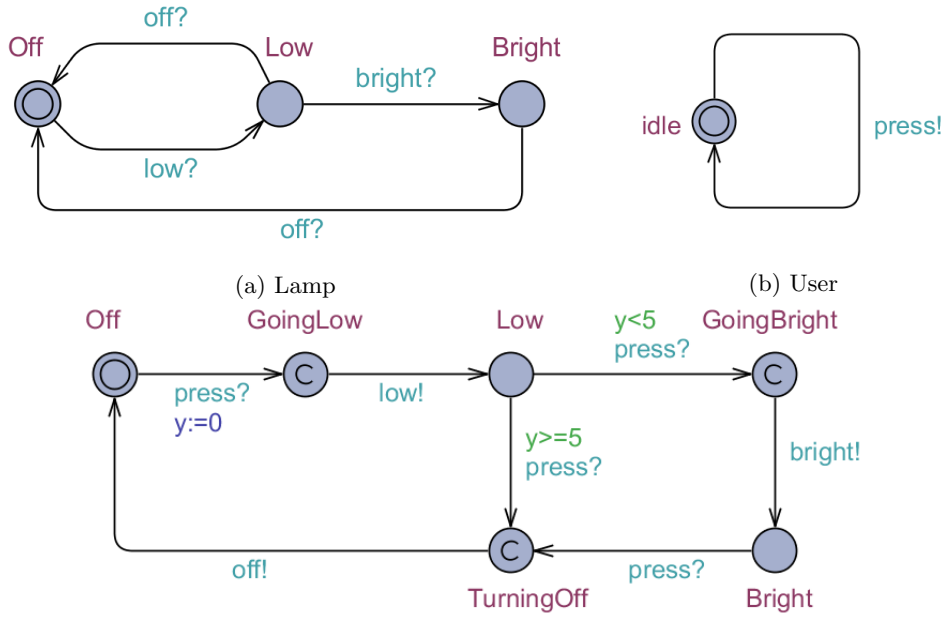
[1] `http://www.uppaal.org/`
[2] `https://github.com/daFritz84/Engineering-Self-Adaptive-Systems`

### Task – Step 6: Prove deadlock freedom

Task 6 stated that the deadlock freedom of the simple lamp example should be proven utilizing the default query provided by Uppaal. This query, `A[] not deadlock` can be found either in the simple tutorial [2], the detailed tutorial [1], or by looking for *deadlock* in the Uppaal help file. In Uppaal, a `deadlock` is a special state formula, which 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 (see detailed tutorial, section 2.2 [1]). To prove the deadlock freedom of our automata, the `deadlock` state formula must not hold for any reachable state. Therefore, Uppaal offers `A[]` $\phi$ ([1]) path formula, where the state $\phi$ must hold for all paths. Combining the state and path formula, the resulting query is created `A[] not deadlock` which ultimately means: For all paths the *deadlock* property does not hold.

## Simple lamp with lamp controller

Figure 2 depicts the three automata modeling the enhancements to the original simple lamp example. The requirements in Task - Step 7 stated that the control of the lamp should now be handled by an intermediary process, the *lamp controller*. Hence, three new synchronization channels were introduced to trigger transitions to the different locations of the lamp process, namely *off*, *low*, and *bright*. Therefore, the lamp automaton is simplified, since it now lacks the transitions necessary for the double-press mechanism. The lamp controller mostly follows the model from the original simple lamp example. A striking repeated modeling pattern of the lamp controller automata is its use of committed locations. These locations had to be inserted to separate the *press* synchronization channel with the triggered synchronization channel (i.e., *off*, *low*, or *bright*). Again the full source code of the example can be found on GitHub [3].




(a) Lamp

(b) User

(c) LampController

Figure 2: The simple lamp with lamp controller example

Another part of the assignment – step 7 was the creation of two queries which prove the reachability of the locations low and bright. Reachability in Uppaal is defined as that there exists a path in an automata where

[3] `https://github.com/daFritz84/Engineering-Self-Adaptive-Systems`

the state formula $\phi$ is possibly satisfied [1]. This task is realized by the following two queries:

`E<> Lamp.Low` There exists a path where `Lamp.Low` eventually holds. Where `Lamp.Low` only holds if in fact the corresponding location in the *lamp* automata has been reached.

`E<> Lamp.Bright` There exists a path where `Lamp.Bright` eventually holds. Where `Lamp.Bright` only holds if in fact the corresponding location in the *lamp* automata has been reached.

# References

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen, *A Tutorial on Uppaal 4.0*, 2006.

[2] *Uppaal 4.0 : Small Tutorial*, 2009