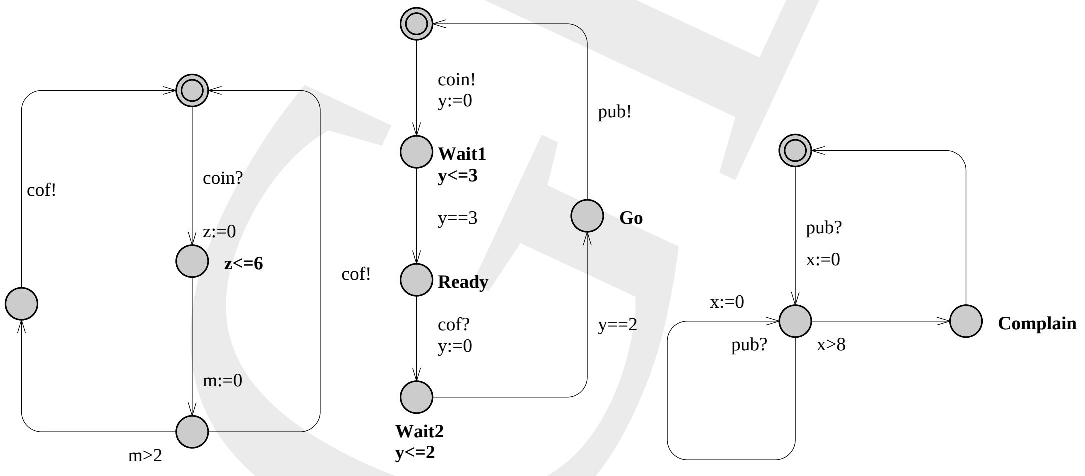
6.2.2 Coffee machine example in Uppaal

The problem is to model the behaviour of a system with three components, a coffee *Machine*, a *Person* and an *Observer*. The person repeatedly tries to insert a coin, tries to extract coffee after which (s)he will make a publication. Between each action the person requires a suitable time-delay before being ready to participate in the next one. After receiving a coin the machine should take some time for brewing the coffee. The machine should time-out if the brewed coffee has not been taken before a certain upper time-limit. The observer should complain if at any time more than 8 time-units elapses between two consecutive publications.



(a) Machine (b) Person (c) Observer

Figure 6.2: The three automata

The automata are shown in Figure 8.3, and (partially) model the specified system. Why partially? In the specification there is a worrying phrase: “The Machine should time-out if the brewed coffee has not been taken before a certain upper time-limit”. This phrase is worrying because it is an under-specification of the system. For example: “What does the machine do if it times out?”. If it times out and then dumps the coffee, the system will deadlock, as the Person automata must pay and then drink. So - rather than modifying the specified Person automaton, the machine specified here times out and <u>then</u> synchronizes on the dispensing of coffee.

- **Machine:** The coffee machine accepts a coin and then delays for some time (above it is 6 time units). It then sets a timeout timer, and either (to the right) dispenses coffee, or (to the left) times out and then dispenses coffee. The extra state on the left is because Uppaal does not allow both guards and synchronizing elements to appear on the same transition.
- **Observer:** The observer has an 8 time unit timeout. If the publications keep coming in more often than 8 time units, then the system stays in the middle state. However, if the timer times out, we visit (briefly) the Complain state.
- **Person:** The person was already specified, and it just puts in a coin and then drinks coffee before publishing.

In UPPAAL, the path operators $\Diamond$ and $\Box$ are written as `<>` and `[]`, so to test the model, the temporal query `E<> Observer.Complain` is used, which corresponds to the CTL formula `EF Observer.Complain`, specifying that:

- for at least one computation path, at some time state `Observer.Complain` is reached.

In addition the system is tested with `A[] not deadlock`. The results of the testing are as follows:

- System is deadlock free
- `Observer.Complain` is reached if the coffee timeout is 7 or more
- `Observer.Complain` is never reached if the coffee timeout is 6 or less

The last two tests were done by trial and error - setting the value in the coffee machine model to different values, and rerunning the model checker.

6.2.3 A simple protocol example in Uppaal

The problem is to model a simple protocol, with a communication *Medium*, a *Sender*, and a *Receiver*. The sender sends messages of a fixed length `length`, which is the time between the beginning and the end of a message. The medium has a transmission delay `delay`.

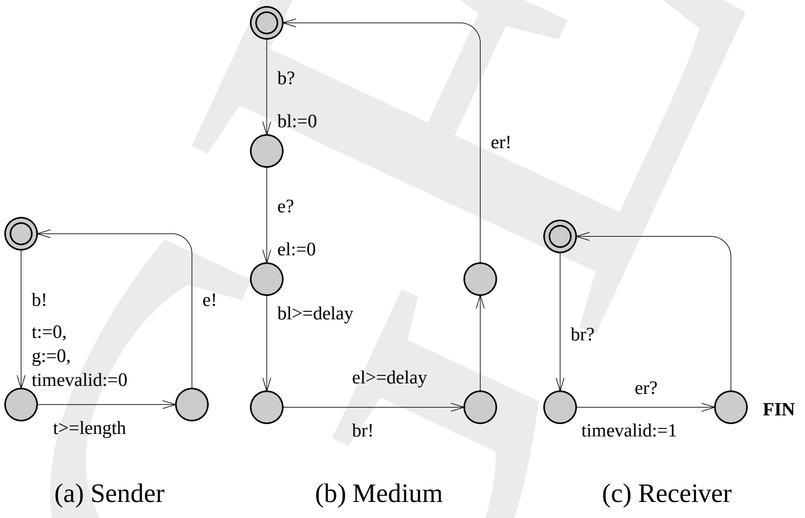


Figure 6.3: The three automata

- **Sender:** The sender just synchronizes on the beginning and end of the message, ensuring a time of length between the two synchronizations. The `timevalid` and `g` variables are global variables used to time the total transit time of the message.
- **Receiver:** The receiver just synchronizes on the beginning and end of the message after it arrives from the medium, setting `timevalid` as the `FIN` state is entered.
- **Medium:** The medium uses two local clocks, `bl` and `el`, to delay a message enroute to the receiver.

The first step is to model the system, assuming `length<delay`. The model in Figure 6.3 shows this model.

A quick test with `A[] not deadlock` shows that it is deadlock free. To find out the total time between begin send and end receive, a global clock variable *g* is reset by the sender at the beginning of a message, and its value in state `Receiver.FIN` tells us the total time between the beginning of sending the message and the end of receiving the message. To test this a global variable `timevalid` was added to the system, and if the receiver is in the `Receiver.FIN` state, and the time is valid, then we can run various tests - for example the query

```
E<> (Receiver.FIN and timevalid==1 and g<maxtime)
```

(where `maxtime` is `length+delay`) is always unsatisfied, which tells us there is no time sequence shorter than `length+delay`. The query

```
A[] (Receiver.FIN and timevalid==1 imply g>=maxtime)
```

is satisfied, which tells us that the time will always be greater than or equal to `length+delay`.

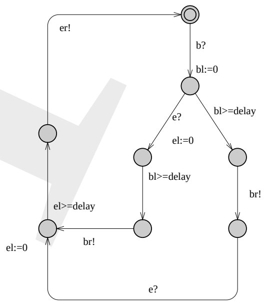


Figure 6.4: The new specification of the medium

The preceding model could only handle systems in which the length of the message was less than the medium delay time. We can extend the medium to also handle messages with `length>=delay`. The only thing that needs changing is the definition of medium, given in Figure 6.4. The two queries before both still produce the expected results, no matter what the relationship between length and delay:

```
E<> (Receiver.FIN and timevalid==1 and g<maxtime)
A[] (Receiver.FIN and timevalid==1 imply g>=maxtime)
```

6.3 Summary of topics

In this section, we introduced the following topics:

Theoretical foundations. *Formal foundations for TCTL, syntax and semantics.*
Model checking algorithms. *Algorithms for checking timed CTL systems.*
Examples. *Two worked Uppaal examples.*