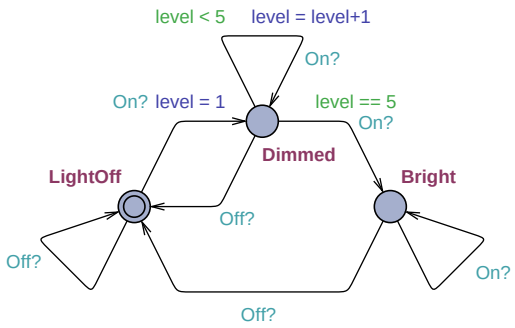
Solution to Exercise 1: Dimmed Light

A possible solution for the fine-grade dimmed light controller (without memory):



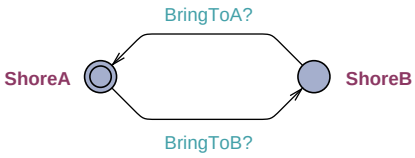
The solution may be found in the model file `DimmedLight.xml`.

Solution to Exercise 2: Goat/Wolf/Cabbage Puzzle

The system may be modelled to more or less detail. In general, however, a solution which models each component separately is preferred. In the solutions below, the phase of actually crossing the river has been abstracted from.

A simple model (1.)

Here we choose to represent each of the items, *wolf*, *goat* and *cabbage* as an automata with two locations which clearly (and visually) indicate the the current shore. These are instantiated from a common template:

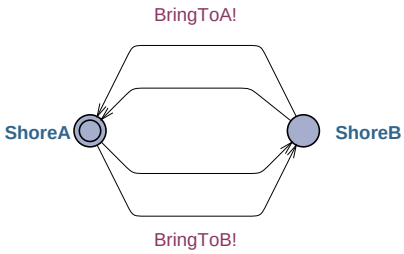


Also, the *man* is modelled to reflect the sailing between the two shores (with or without a passenger):

Finally the Passsenger template is instantiated for each item:

```
Goat    = Passenger();
Wolf    = Passenger();
Cabbage = Passenger();

system Man, Goat, Wolf, Cabbage;
```



The full system can be found as `WolfGoatCabbage0.xml`.

Notice that the passengers (as well as the man) could have represented by (a family of) binary variables indicating the current shore rather than eksplicit automata. This, however, would not have provided the same simple visual way of checking whether a system state is safe or not.

Solving the Puzzle

In the simulator, it can now be demonstrated that there is a way of travelling which brings all 3 passengers safely to shore B (the trick is to bring back the goat on the second journey):

```
[Man,Wolf,Goat,Cabbage]                    []
                          (Man,Goat) ->
[Wolf,Cabbage]                             [Man,Goat]
                      <- (Man,_)
[Man,Wolf,Cabbage]                         [Goat]
                          (Man,Wolf) ->
[Cabbage]                                  [Man,Wolf,Goat]
                      <- (Man,Goat)
[Man,Goat,Cabbage]                         [Wolf]
                          (Man,Cabbage) ->
[Goat]                                     [Man,Wolf,Cabbage]
                      <- (Man,_)
[Man,Goat]                                 [Wolf,Cabbage]
                          (Man,Goat) ->
[]                                         [Man,Wolf,Goat,Cabbage]
```

A more controlled solution (still for 1.)

The above solution lets the simulator choose among the possible synchronizations in order to determine which passenger to bring across the river. However, the man himself has no way of controlling with whom he synchronizes. To enable more control, we may let the passenger template take the BringToA and BringToB channels as parameters (these have to be passed by reference using `&`).

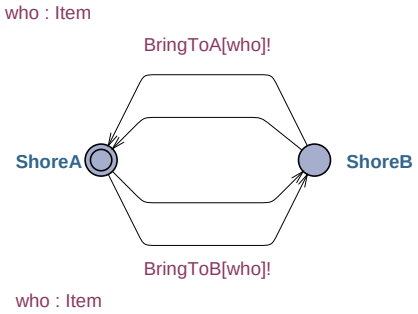
Now, each passenger is assigned a unique channel from a family of channels declared globally as:

```
/* Channels used to get Passengers (Wolf, Goat or Cabbage) across the river */
```

```
const int wolf = 0;
const int goat = 1;
const int cabbage = 2;

chan BringToA[3], BringToB[3];
```

The *man* should now explicitly take a particular passenger with him.



Here, a *selection* is used to allow for any of the three passengers to be transferred if possible. This is just a shorthand for making synchronizations on the individual channels.

Finally the system is instantied by:

```
Wolf    = Passenger(BringToA[wolf], BringToB[wolf]);
Goat    = Passenger(BringToA[goat], BringToB[goat]);
Cabbage = Passenger(BringToA[cabbage], BringToB[cabbage]);

system Man, Wolf, Goat, Cabbage;
```

This system can be found as the model file `WolfGoatCabbage1.xml`.

Note that in this system the man is still not making any qualified decisions and bad scenarios are possible.

Testing the system (2.)

To see whether the man behaves safely, we use the invariant:

```
A[] (Wolf.ShoreA == Goat.ShoreA imply Goat.ShoreA == Man.ShoreA) and
    (Cabbage.ShoreA == Goat.ShoreA imply Goat.ShoreA == Man.ShoreA)
```

which states that if there is a risk of eating on some shore, then the man is present there. As expected it is not satisfied.

This invariant together with other queries have been added in `WolfGoatCabbage2.xml`.

A safe systems (3.)

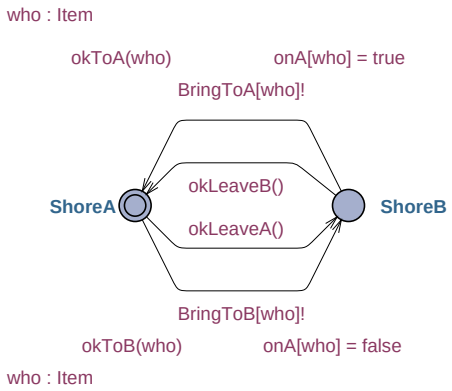
To make the system safe, the man must maintain some knowledge about the location of the items and use that for guarding his actions. Here, the man stores whether an item is on shore A or not:

```
bool onA[Item] = {true, true, true};
```

In order not to clutter the automaton with complex conditions, the guard may be given as a *local function* for the Man template written using a C-like syntax.

The condition for bringing a particular *item* from A to B is called `okToB(`*item*`)` (and vice versa). Also there are conditions for leaving a shore without passengers.

The Man template now becomes



The local functions are defined using an auxiliary notion of a *safe shore* where nobody gets eaten when the man leaves. The strategy is then to make sure that the resulting state is safe. This system can be found as `WolfGoatCabbage3.xml`.

Now, the above invariant is satisfied, and the query

```
E<> Wolf.ShoreB and Goat.ShoreB and Cabbage.ShoreB and Man.ShoreB
```

may be used to find a solution to the puzzle. The query

```
A<> Wolf.ShoreB and Goat.ShoreB and Cabbage.ShoreB and Man.ShoreB
```

is not satisfied though, as the man may go back and forth (with the goat) indefinitely.

Forcing a result (4.)

To be sure to reach the other shore the man must have a strategy that drives the situation towards the goal. It turns out that the following principles are sufficient:

- Never sail from A to B without bringing anything

- Sail the goat from A to B only in the beginning/end
- Only bring back the goat and only when the wolf and the cabbage are on different shores.

These conditions may be conjoined with the safety ones as shown in `WolfGoatCabbage4.xml`.

To be sure that any steps happens at all, however, it is necessary to force the system to take non-time actions. This is achieved by declaring both locations of the man to be urgent.