INGI1131 Practical Exercises
Lab 10: Lift Control System

This lab session is dedicated to the study of state diagrams for concurrent systems and the lift control system. The knowledge that you will acquire will help you to design the multi-agent system for the project.

1. Figure 1 shows the notation for state diagrams where components receive and send messages, changing from one state to another. Consider a dishwasher that is programmed as a port object. Every time the dishwasher receives an input (a dish, a glass, etc.), it analyses whether it is full or not. When it is full, it does not accepts any other input and waits for a message to start cleaning the dishes. In order to know when the cleaning task is done, it sends a message to a timer that send a message back after reaching the time out. After that, the machine must come back to the empty state.

   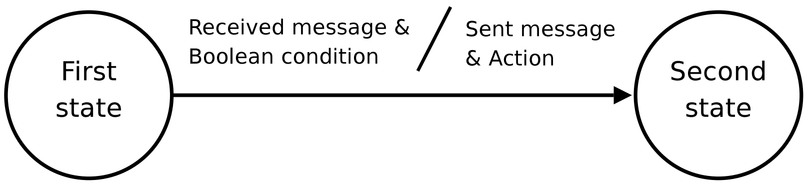

   Figure 1: Notation for state diagrams

   Model the state diagram of the dishwasher using the notation described on Figure 1.

2. Consider the implementation of the Lift Control System that is given on:
   iCampus → INGI1131 → Documents and Links → Labs → **lift.oz**. Suppose we want to redefine Building like this:

```
proc {Building FN LN ?Floors ?Lifts} C in
   {Controller C}
   Lifts={MakeTuple lifts LN}
   for I in 1..LN do
      Lifts.I={Lift I state(1 nil false) C Floors}
   end
   Floors={MakeTuple floors FN}
   for I in 1..FN do
      Floors.I={Floor I state(false) Lifts}
   end
end
```

   (i.e. we are associating the same controller to all the lifts of the building). What would be the impact of such a change in the Lift control system?

3. Suppose that, instead of using a port object for a controller, we modeled it as a simple abstraction:

```
proc {Controller Msg}
   case Msg
   of step(Lid Pos Dest) then
      if Pos<Dest then
     {Delay 1000} {Send Lid 'at'(Pos+1)}
      elseif Pos>Dest then
     {Delay 1000} {Send Lid 'at'(Pos-1)}
      end
   end
end
```

   a Make the corresponding changes in the source in order to fit with this new definition of Controller.

   b What would be the impact of such a change in the Lift control system?

   c Does the state diagram of any component change?

4. The lift of this building (Réaumur) is much simpler than the one we have programmed on this course. What is its state diagram? Do some experiments with it in order to infere its state diagram. For instance, test what happen if you press the button on a floor while the door is still open. What happen if you call the lift that is already busy?

5. Suppose we want to have autonomous lifts that don't make use of any controller. Each lift, autonomously, decides where to go. Under this new design, there is no need for outgoing 'step' messages nor incoming 'at' messages. How would you implement such a Lift control system without losing the properties of the current system? For instance, note that, under the current implementation, the lift is able to receive call messages while deciding where to go. Draw the state diagram.

6. Improve the scheduling of the given system. For example, assume the lift is moving from floor 1 to floor 5. Calling floor 3 should cause the lift to stop on its way up, instead of the naive solution where it first goes to floor 5 and then down to floor 3. With this improved scheduler, we can extend the call button to separately call up-going and down-going lifts. How would you implement this? Once again: it is a very good idea to analyse the state diagrams instead of going directly to the code.

7. Improve the given system to pick up the lift that is closest instead of a random lift. Build a concurrent negotiator, that does not cause the lift to wait while a negotiation is happening. We create a new port object for each negotiation. The negotiator and the lift object can be considered together, as a single compound component. Give also the state diagram for the negotiator object, the new floor object, and the new lift object.