Seif Haridi

Assignment 5, CS2104: Lift Simulation

Due date: 2002-11-10
Points: 8 from 100 exam points

This lab assignment uses agents for simulating a simple lift scenario. It exercises modeling with agents, concurrency, and message sending. This lab is a variant of the lift case-study of Chapter 5.

The Lift Scenario

Assume we have a building with n lifts and m floors. When a lift is called, we wish to service the request in a reasonable manner: the lift will be selected that can service the request as soon as possible.
Imagine for the time being a building with three lifts ($n = 3$) and five ($m = 5$) floors. Floors one and five have a single call button, the remaining floors have two call buttons each for *up* and *down* respectively.
A lift is requested by pressing one of the buttons. A request is served by selecting a lift that stops at the floor. In addition to that, each lift has five buttons (or, in general m buttons) inside labeled with the floor numbers. These buttons will stop the lift at the appropriate floor.
Our task is now to write a program that controls the lift system. We will construct a solution that models each lifts and floors by agents that react to incoming messages and possibly send messages to other agents. It is known to be good practice to model each problem agent by one program agent.

Scheduling Lifts

Suppose the up button is pressed on the fourth floor. We have to decide which lift should stop at the fourth floor. We will use the following algorithm:

- The agent for floor 4 sends a message to each of the lift agents asking them how long it would take them to get to floor 4. It also tells the lifts that there is a person at floor 4 who wants to go up.

- Each lift receiving this message works out how long it would take for it to get to floor 4 and replies with a message containing the time (the *waittime*).
- The floor agent chooses the lift with the minimal proposed waittime and replies with a `reserve` message. All other lifts are informed by a `reject` message.

We assume that a lift agent is locked after it has sent a time estimate and remains locked until it receives either a `reserve` or `reject` message.
This strategy works and does not lead to deadlocks if all request messages sent to lifts are received by the lifts in the same order. Why?

Modeling

We model the scenario with three different types of agents which includes of course the lift and floor agents mentioned above together with agents who model the actual lift cabin.

The Floor Agent. The only message a floor agent receives is that a button has been pressed. This is modeled by either a message `press(up)` or `press(down)` depending on which button has been pressed. When a button press is received, the best lift is selected as described above.
The state of a floor agent contains its floor number and a list of all lift agents. The state is encoded as record

```
state(floor:I lifts:Ls)
```

where `I` is the floor number and `Ls` is the list of lift agents.

The Lift Agent. A lift agent understands the following messages:

- `request(floor:I dir:D answer:A)` which is sent by the floor agent for floor `I`. `D` is either `up` or `down` depending on the button pressed at floor `I`.

  The answer `A` is an unbound variable. The lift agent binds the variable to a record `propose(time:T status:S)` where `T` is the waittime. `S` again is an unbound variable.

  The unbound variable `S` is in turn bound to either `reject` or `reserve` by the floor agent. The best lift agent gets a `reserve` while the others get a `reject`. If the agent gets a `reserve`, it must include the floor number `I` in the floors it must visit (the list of floors to be visited is called *stoplist*).

- `arrived(floor:I action:A)` is sent by the cabin agent when the cabin has reached floor `I`. The lift agent now must determine which action the cabin agent takes next by binding the variable `A` to either:

1. `stop`: the cabin will open and close the doors. This takes five time units.
2. `up`: the cabin moves up one floor. This takes one time unit.
3. `down`: the cabin moves down one floor. This takes one time unit.
4. `wait`: the cabin stays where it is for one time unit.

After the cabin has executed its action, it sends again an `arrived`-message to the lift agent.

- `stop(floor:I)` requests the lift to stop a floor `I` (coming from the buttons inside the lift).

The state of the lift agent is a record `state(floor:I stop:S)` where `I` is the floor the lift agent is currently at (this is known from messages sent by the cabin agent) and `S` is the stoplist.
The stoplist is a list of integers describing at which floors the lift agent must stop. Initially, the stoplist is empty. Floor numbers are inserted into the stoplist whenever a lift agent has received a `request` message and subsequently also has been confirmed that its offer has been the best.
Closely related to managing the stoplist is computing the waittime. The waittime is determined by where the floor would be inserted into the stoplist.

1 Agents With State

We use the `NewAgent` abstraction for agents. It implements an agent as a port to which messages can be sent and a thread that serves the incoming messages.

```
fun {NewAgent Process InitState}
   Port Stream
   proc {Serve Messages State}
      case Messages of Message|Next then
         {Serve Next {Process State Message}}
      end
   end
in
   Port={NewPort Stream}
   thread {Serve Stream InitState} end
   Port
end
```

Here the behavior of an agent is given by a binary function `Process`, taking a `State` and a `Message` as input and returning a new state.

A Simple Memory Cell Agent. For example, a simple agent for maintaining a memory cell can be programmed as follows:

```
fun {ProcessMemoryCell State Message}
   case Message
   of read(Old) then
      Old=State.value State
   [] write(New) then
      {AdjoinAt State value New}
   [] exchange(New Old) then
      Old=State.value
      {AdjoinAt State value New}
    end
end
```

Please note that the function *always* returns a state. In case the state has not been modified (as for the `read`-message), the input state is returned.
The state is implemented as a record with a single feature `value` and the field giving the current value of the memory cell.
In order to conveniently create new memory cell agents, we define

```
fun {NewMemoryCell InitValue}
   {NewAgent ProcessMemoryCell state(value:InitValue)}
end
```

Try the following example:

```
declare
M={NewMemoryCell 5} A B
{Browse A#B}
{Send M read(A)}
{Send M write(7)}
{Send M exchange(8 B)}
```

2 Inserting Floors into the Stoplist

Implement a function `{Insert Stoplist Now Dir Floor}` that returns a new stoplist. Here `Stoplist` is the current stoplist, `Now` is the floor the lift agent is currently located, `Dir` is either `up` or `down` depending on which button has been pressed at floor `Floor`.
It is important to compute a new stoplist that minimizes time until the lift arrives at the desired floor. Take the following issues into account:

- `Floor` is already contained in the stoplist.
- The lift is already at `Floor`.
- If the lift moves from floor `A` to `B` with `A<Floor<B` and `Dir` is `up`, insert between `A` and `B`.
- If the lift moves down from floor `A` to `B` with `A>Floor>B` and `Dir` is `down`, insert between `A` and `B`.

- Insert `Floor` as much to the front of the stoplist as possible while following the above rules.

For example,

```
{Insert nil 3 up 6}
```

returns `[6]`, whereas

```
{Insert [5 7 9 3] 4 up 6}
```

returns `[5 6 7 9 3]`, and

```
{Insert [5 7 9 3] 4 down 6}
```

returns `[5 7 9 6 3]`.
Store the program in the file `Insert.oz`.

Hint. In order to test your scenario you might want to consider a simple implementation of `Insert` first.

3 Computing the Waittime

Implement a function `{WaitTime Stoplist Now Dir Floor}` that returns the waittime before a request can be served. Here `Stoplist` is the current stoplist, `Now` is where the lift is currently located, `Dir` is either `up` or `down` depending on which button has been pressed at floor `Floor`.
As mentioned above, stopping the lift takes five time units whereas moving the lift by one floor takes one time unit. The waittime must be computed with the same rules as for insertion into the stoplist.
For example,

```
{WaitTime [5 7 9 3] 4 up 6}
```

returns `7`, and

```
{WaitTime[5 7 9 3] 4 down 6}
```

returns `23`.
Store the program in the file `WaitTime.oz`.

4 Pressing a Lift Button

Implement a function `{StopAt Stoplist Now Floor}` that returns a new stoplist when a button for `Floor` is pressed inside the lift. Here `Stoplist` is the current stoplist, and `Now` is the floor the lift agent is currently located.
Follow the same rules to insert `Floor` into the stoplist as for `Insert`.
Store the program in the file `StopAt`.

5 The Lift Agent

Implement a function `{LiftProcess State Message}` for the lift agent which returns a new state.
The file `Lift.oz` available from the course webpage already contains code fragments that you have to complete.

6 The Floor Agent

Implement a function `{FloorProcess State Message}` for the floor agent which returns a new state.
Use a procedure similar to `FindLowest` in Lab Assignment 2 to select the lift with the shortest waittime.
The file `Floor.oz` available from the course webpage already contains code fragments that you have to complete.

7 Putting Everything Together

Download the files `Main.oz` and `Cabin.oz` from the course webpage. The file `Main.oz` creates a complete lift simulation scenario for a given number of floors and lifts.
You can simulate pressing buttons by sending the appropriate messages to the lift or floor agents.

Acknowledgments

The problem has been taken from: Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams, *Concurrent Programming in Erlang*, second edition. Prentice Hall, London, UK, 1996.