Homework 5: Message Passing

Due: problems 1–6, 8–9, 11–12, Monday, April 14, 2008 at 11pm; problems 7 and 10, Wednesday, April 16, 2008 at 11pm.

In this homework you will learn about the message passing model and basic techniques of programming in that model. The programming techniques include using port objects to create agents (state machines)[Concepts] [UseModels]. A few problems also make comparisons with the other models we have studied, and also with message passing features in other languages [EvaluateModels] [MapToLanguages].

Your code should be written in the message passing model, so must not use cells and assignment in your Oz solutions. Furthermore, note that the message passing model does *not* include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions.

You should use helping functions whenever you find that useful. Unless we specifically say how you are to solve a problem, feel free to use any functions from the Oz library (base environment), especially functions like `Map` and `FoldR`.

For all Oz programing tasks, you must run your code using the Mozart/Oz system. For these you must also provide evidence that your program is correct (for example, test cases). Oz code with tests for various problems is available in a zip file, which you can download from the course resources web page. For testing, you may want to use tests based on our code in the course library file `TestingNoStop.oz`.

Turn in (on WebCT) your code and, if necessary, output of your testing for all questions that require code. Please upload text files with your code that have the suffix `.oz` (or `.java` as appropriate), and text files with suffix `.txt` that contain the output of your testing. Please use the name of the main function as the name of the file. (In any case, don't put any spaces in your file names!)

If we provide tests and your code passes all of them, you can just indicate with a comment in the code that your code passes all the tests. Otherwise it is necessary for you to provide us test code and/or test output. If the code does not pass some tests, indicate in your code with a comment what tests did not pass, and try to say why, as this enhances communication and makes commenting on the code easier and more specific to your problem than just leaving the buggy code without comments.

If you're not sure how to use our testing code, ask us for help.

Your code should compile with Oz, if it doesn't you probably should keep working on it. If you don't have time, at least tell us that you didn't get it to compile.

Be sure to clearly label what problem each area of code solves with a comment.

Don't hesitate to contact the staff if you are stuck at some point.

Read Chapter 5 of the textbook [VH04]. (See the syllabus for optional readings.)

Message Passing Semantics and Expressiveness

1. (30 points) [Concepts] [UseModels]

   Using Oz's message passing model, implement a data abstraction `Box`, by writing the following functions and procedures.

   ```
   NewBox: <fun <$ Value>: Box>
   BoxExchange: <proc <$ Box Value Value>>
   BoxAssign: <proc <$ Box Value>>
   BoxAccess: <fun <$ Box>: Value>
   ```

   A Box is like a Cell, in that it holds a value (of any type). The function call `{NewBox X}` returns a new Box containing the value `X`. The procedure call `{BoxExchange B Old New}` atomically binds `Old` to the value in box `B` and makes `New` be the new value contained in Box `B`. The procedure call `{BoxAssign B V}` makes the value `V` be the new value of Box `B`. The function call `{BoxAccess B}` returns the value contained in Box `B`.

   Your code should pass the tests shown in Figure 1 on the following page

```
% $Id: BoxTest.oz,v 1.1 2007/11/09 21:32:07 leavens Exp leavens $
\insert 'Box.oz'
\insert 'TestingNoStop.oz'
declare B1 B2 V1old V2old
{StartTesting 'Box operations'}
B1 = {NewBox 1}
B2 = {NewBox 2}
{BoxExchange B1 V1old 7}
{BoxExchange B2 V2old 99}
{Test V1old '==' 1}
{Test V2old '==' 2}
declare V1x V2x
{BoxExchange B1 V1x 88}
{BoxExchange B2 V2x 333}
{Test V2x '==' 99}
{Test V1x '==' 7}
{Test {BoxAccess B1} '==' 88}
{Test {BoxAccess B2} '==' 333}
{Test {BoxAccess B2} '==' 333}
{Test {BoxAccess B1} '==' 88}
{BoxAssign B1 4}
{Test {BoxAccess B1} '==' 4}
{Test {BoxAccess B2} '==' 333}
{BoxAssign B1 asymbolliteral}
{Test {BoxAccess B1} '==' asymbolliteral}
declare
X=1 Y=2 Z=3
B={NewBox Z}
{StartTesting 'Some equations'}
{Test {BoxAccess {NewBox X}} '==' X}
{BoxAssign B Y}
{Test {BoxAccess B} '==' Y}
```

Figure 1: Testing code for Problem 1 on the previous page.

You should use the `NewPortObject.oz` file, given in the book and supplied with the test cases for this homework, in your solution. (Hint: represent a Box with a port object, have `NewBox` return a port, and have the other functions send messages to the port. The state of the port object will be the Box's value.)

You are *not* allowed to use cells in your solution!

2. (20 points) [Concepts] [UseModels]

   Using Cells, but without using the message passing primitives `NewPort` and `Send`, define in Oz an ADT `PortAsCell`, which acts like the built-in port type, but is represented as a Cell. (For more about the imperative model and cells, look back at Section 1.12 of the textbook or forward to chapter 6.) The `PortAsCell` ADT has two operations:

   ```
   MyNewPort: <fun <$ Stream>: PortAsCell>
   MySend: <proc <$ PortAsCell Value>>,
   ```

   which are intended to act like `NewPort` and `Send`. That is, the function `MyNewPort` takes an undetermined store variable, and returns a `PortAsCell`, which is a Cell that we want to act like a Port. The procedure `MySend` takes such a `PortAsCell` and a Value and adds the Value to the corresponding stream. In other words, the idea behind the ADT `PortAsCell` is that it should act like the built-in Port ADT of Oz, but be represented using Cells.

   For this problem, don't worry about the improper uses of the stream argument to `MyNewPort` (see Problem 11 on page 11). Also for the moment, don't worry about potential race conditions when multiple threads are used.

   Hint: look at the semantics for `NewPort` and `Send` in section 5.1 of the textbook, which shows how these work in terms of the mutable memory. Mutable memory is like a collection of cells. Think of the port name as being represented by the cell's identity. The cell holds the undetermined store variable that is the end of the list. Note how the semantics of `Send` manipulates the mutable memory and works with new undetermined store variables.

   Your code should pass the tests shown in Figure 2 on the next page.

   You are *not* allowed to use `Send`, `NewPort`, or functions that call them (such as `NewPortObject`) in your solution.

3. [Concepts] [EvaluateModels]

   This problem concerns the expressive power of the message passing model in comparison to the imperative model.

   (a) (5 points) Can the Box ADT of Problem 1 on page 1 do everything that a Cell can do in Oz? Briefly explain.

   (b) (5 points) Can the PortAsCell ADT of Problem 2 do everything that a Cell can do in a sequential Oz program? Briefly explain.

   (c) (10 points) Does your implementation of the PortAsCell ADT of Problem 2 have any potential race conditions when used in an Oz program with multiple threads? That is, is it possible that `MyNewPort` or `MySend` act differently than `NewPort` and `Send` when there are multiple threads? Briefly explain your answer.

   (d) (5 points) Is there any significant difference in expressive power between adding Cells or adding `NewPort` and `Send` to the kernel language?

```
% $Id: PortAsCellTest.oz,v 1.2 2007/11/18 14:09:59 leavens Exp leavens $
\insert 'PortAsCell.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'MyNewPort'}
% Simulating basic semantics of NewPort and Send
declare Strm Port in
Port = {MyNewPort Strm}
{StartTesting 'MySend'}
{MySend Port 3}
{MySend Port 4}
% Must use List.take, otherwise Test suspends...
{Test {List.take Strm 2} '==' [3 4]}
{MySend Port 5}
{MySend Port 6}
{Test {List.take Strm 4} '==' [3 4 5 6]}

{StartTesting 'MyNewPort second part'}
declare S2 P2 U1 U2 in
P2 = {MyNewPort S2}
{StartTesting 'MySend second part'}
{MySend P2 7}
{MySend P2 unit}
{MySend P2 true}
{MySend P2 U1}
{MySend P2 hmmm(x:U2)}
U1 = 4020
{Test {List.take S2 5} '==' [7 unit true 4020 hmmm(x:U2)]}
{Test {List.take Strm 4} '==' [3 4 5 6]}
```

Figure 2: Testing code for Problem 2 on the preceding page.

Message Passing Programming

4. (10 points) [UseModels]

   Using Oz's message passing model, write a function `NewGauge` that takes no arguments and returns a port object. The returned port object should store an integer value in its state and can respond to the messages `up`, `down`, and `fetch(Var)`. Initially the value stored in the port object is 0. The message `up` increments this value and `down` decrements it. The `fetch(Var)` message binds `Var` to the current value. Figure 3 contains some examples.

   You should use `NewPortObject` in your solution (see the textbook, and the `NewPortObject.oz` file supplied with the test cases for this homework).

```
\insert 'Gauge.oz'
\insert 'TestingNoStop.oz'
declare
MyGauge = {NewGauge}
{Test local R in {Send MyGauge fetch(R)} R end '==' 0}
{Send MyGauge up}
{Test {Send MyGauge fetch($)} '==' 1}
{Send MyGauge up}
{Test {Send MyGauge fetch($)} '==' 2}
{Send MyGauge down}
{Test {Send MyGauge fetch($)} '==' 1}
{Send MyGauge up}
{Send MyGauge up}
{Test {Send MyGauge fetch($)} '==' 3}

MyGauge2 = {NewGauge}
{Test {Send MyGauge2 fetch($)} '==' 0}
{Send MyGauge2 down}
{Test {Send MyGauge2 fetch($)} '==' ~1}
{Send MyGauge2 down}
{Test {Send MyGauge2 fetch($)} '==' ~2}
for I in 1..12 do {Send MyGauge2 up} end
{Test {Send MyGauge2 fetch($)} '==' 10}
```

Figure 3: Testing code for Problem 4.

5. (20 points) [UseModels]

   Using Oz's message passing model, write a function `NewAuctioneer` that takes no arguments and returns a port object that acts as an auctioneer (an agent that is selling an item in an auction).

   This port object understands several messages. The `bid(Amt DidIWin)` message places a bid on the item, where `Amt` is a non-negative integer (the number of dollars bid) and `DidIWin` is an undetermined store variable.

   The `stop` message ends the auction. When the `stop` message is received, the `DidIWin` variable in the first `bid` message received whose `Amt` was the largest is bound to `true`, and all other `DidIWin` variables in other `bid` messages are bound to `false`. (Note that there may be no bids. In case of a tie, the first `bid` message received by the port with the highest amount wins. If any `bid` message is received after the `stop` message, it does not win. You can assume that only one `stop` message is ever sent to the port object.)

   Figure 4 on the following page contains some examples.

   You should use `NewPortObject` in your solution (see the textbook, and the `NewPortObject.oz` file supplied with the test cases for this homework).

   Hint: you may want to use helping functions or procedures.

```
\insert 'Auctioneer.oz'
\insert 'TestingNoStop.oz'
declare
MyAer = {NewAuctioneer}
local Status1 Status2 Status3 Status4 in
   {Send MyAer bid(17 Status1)}
   {Send MyAer bid(8 Status2)}
   {Send MyAer bid(27 Status3)}
   {Send MyAer bid(24 Status4)}
   {Send MyAer stop}
   {Test Status1 '==' false}
   {Test Status2 '==' false}
   {Test Status3 '==' true}
   {Test Status4 '==' false}
end

MyAer2 = {NewAuctioneer}
local Status1 Status2 Status3 Status4
   Status5 Status6 Status7
in
   {Send MyAer2 bid(100 Status1)}
   {Send MyAer2 bid(15 Status2)}
   {Send MyAer2 bid(99 Status3)}
   {Send MyAer2 bid(99 Status4)}
   {Send MyAer2 bid(100 Status5)}
   {Send MyAer2 bid(100 Status6)}
   {Send MyAer2 bid(100 Status7)}
   {Send MyAer2 stop}
   % first of the highest bids wins
   {Test Status1 '==' true}
   {Test Status2 '==' false}
   {Test Status3 '==' false}
   {Test Status4 '==' false}
   {Test Status5 '==' false}
   {Test Status6 '==' false}
   {Test Status7 '==' false}
end

% Stopping with no bids
MyAer3 = {NewAuctioneer}
{Send MyAer3 stop}
% Bidding after an auction is over
```

Figure 4: Testing for Problem 5 on the previous page.

6. (40 points) [UseModels]

   Using Oz's message passing model, write a function `NewCatalog` that takes no arguments and returns a port object that acts as a catalog.

   The returned port object understands several messages. The `add(Key Value)` message adds an association between the `Key` and the `Value`; the added association replaces any previous association in the catalog whose key is `Key`. The `lookup(Key WhatValue)` message takes key value, `Key`, and an undetermined store variable, `WhatValue`; if `Key` is associated with a value, that value is unified with `WhatValue`; if `Key` is not associated with a value, then the port object remembers the lookup request and if an `add` message of the form `add(`K V`)` arrives, where K `==` `Key`, then the port object binds `WhatValue` to V.

   Figure 5 on the following page contains some examples.

   You should use `NewPortObject` in your solution.

   Hint: you may want to use helping functions or procedures.

7. (80 points) [Concepts] [UseModels] [MapToLanguages]

   Do problem 7 in section 5.9 of the textbook [VH04] (Erlang's receive as a control abstraction). Put your code that solves this problem in a file `Mailbox.oz`.

   Note that, according to the book's errata, when the `D` argument to `Mailbox.receive` has the form `T#E`, then `E` should be a *zero*-argument function.

   Some testing for this is shown in Figure 6 on page 9 and Figure 7 on page 10.

   Some hints follow.

   Be sure to read section 5.7.3 in the textbook. Figure 5.21 in particular explains the semantics of `receive`, although you are not going to code that directly.

   In the problem, the name `Mailbox` is a variable identifier that is bound to a record containing three fields: `new`, `send`, and `receive`. Thus in outline your code should do something like the following.

```
declare
local
   % ...
   fun {New} ... end
   proc {MSend C M} ... end
   fun {MReceive C PGL D} ... end
in
   Mailbox = mailbox(new:New send:MSend receive:MReceive)
end
```

   Now `Mailbox.new` denotes the function `New`, since it does a record selection on the `Mailbox` record.

   In essence, your function `New` will return a port object (or something that contains a port object), which you will make by calling `NewPortObject`. Then `MSend` and `MReceive` will send messages to the port object passed as their first argument (or contained in their first argument). However, `MSend` and `MReceive` do very little themselves. In particular, `MReceive` can pass along its arguments `PGL` and `D` in a message, so that the port object (which has the appropriate state) can do the work.

   The port object should contain a list (or queue) of unprocessed messages in its state. It is helpful to think of the port object as having two states, one in which it is receiving (i.e., still processing a receive), and another in which it is not receiving. You can use a timer like the one in Figure 5.8 for timeouts, but you may need to modify it to suit this problem.

```
\insert 'Catalog.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'NewCatalog'}
declare
MyCat = {NewCatalog} % (Meow! :-)
local What BKStre What2 What3 What4 What27 What5 in
   {StartTesting 'add then lookup'}
   {Send MyCat add(true false)}
   {Test {Send MyCat lookup(true $)} '==' false}
   {Send MyCat add("EBay" ebay)}
   {Send MyCat lookup("EBay" What)}
   {Test What '==' ebay}
   {Test {Send MyCat lookup("EBay" $)} '==' ebay}
   {StartTesting 'lookup then add'}
   {Send MyCat lookup("Amazon" BKStre)}
   {Send MyCat add("Amazon" bookstore)}
   {Test BKStre '==' bookstore}
   {StartTesting 'persistence'}
   {Test {Send MyCat lookup("Amazon" $)} '==' bookstore}
   {Send MyCat lookup("EBay" What2)}
   {Test What2 '==' ebay}
   {StartTesting 'changing added value'}
   {Send MyCat add("EBay" 'auction house')}
   {Send MyCat lookup("EBay" What3)}
   {Test What3 '==' 'auction house'}
   {Send MyCat lookup(27 What27)}
   {Send MyCat lookup(4 What4)}
   {Send MyCat add(5 'first try')}
   {Test {Send MyCat lookup(5 $)} '==' 'first try'}
   {Send MyCat add(4 'not a prime')}
   {Send MyCat add(27 'good hailstone argument')}
   {Send MyCat add(5 'second try')}
   {Test {Send MyCat lookup(5 $)} '==' 'second try'}
   {Send MyCat add(5 'first odd non prime')}
   {Send MyCat lookup(5 What5)}
   {Test What4 '==' 'not a prime'}
   {Test What27 '==' 'good hailstone argument'}
   {Test What5 '==' 'first odd non prime'}
end
DotCom = {NewCatalog}
local WhatCat WhatCat2 WhatStore WhatPaper What4 WhatMore
in
   {Send DotCom add('MyCat' MyCat)}
   {Send DotCom lookup('MyCat' WhatCat)}
   {Send WhatCat lookup("Amazon" WhatStore)}
   {Test WhatStore '==' bookstore}
   {Send DotCom lookup('New York Times' WhatPaper)}
   {Send DotCom add('New York Times' 'the New York Times website')}
   {Test WhatPaper '==' 'the New York Times website'}
   {Send DotCom lookup('MyCat' WhatCat2)}
   {Send WhatCat2 lookup(4 What4)}
   {Test What4 '==' 'not a prime'}
   thread {Test WhatMore '==' 'delayed'} end
   {Send WhatCat lookup('More' WhatMore)}
   {Send WhatCat add('More' 'delayed')}
end
```

Figure 5: Testing for Problem 6 on the previous page.

```
\insert 'Mailbox.oz'
\insert 'TestingNoStop.oz'
declare
%% conditions for testing (guards), used only for testing purposes
fun {Else X} true end
fun {Equals Val} fun {$ X} X == Val end end
fun {GreaterThan Val} fun {$ X} X > Val end end
fun {LessThan Val} fun {$ X} X < Val end end
fun {And C1 C2} fun {$ X} {C1 X} andthen {C2 X} end end
fun {Or C1 C2} fun {$ X} {C1 X} orelse {C2 X} end end
fun {Negate C} fun {$ X} {Not {C X}} end end
%% bodies for when guards are true, used only for testing purposes
fun {Id X} X end
fun {Add Y} fun {$ X} X+Y end end
%% Bodies for timeouts,
%% note that this is a zero-argument procedure, contrary to the book!
fun {Timedout} timedout end

{StartTesting 'Mailbox'}
MB={Mailbox.new}
{StartTesting 'send'}
{Mailbox.send MB 4020}
{StartTesting 'receive'}
{Test {Mailbox.receive MB [Else#Id] infinity} '==' 4020}

{StartTesting 'send and receive of unbound'}
{Mailbox.send MB _}
{Test {Not {IsDet {Mailbox.receive MB [Else#Id] infinity}}} '==' true}

{StartTesting 'Multiple sends and receives'}
{Mailbox.send MB 340}
{Mailbox.send MB 99}
{Test {Mailbox.receive MB [Else#Id] infinity} '==' 340}
{Test {Mailbox.receive MB [Else#Id] infinity} '==' 99}

{StartTesting 'Receive based on guards'}
{Mailbox.send MB 11}
{Mailbox.send MB 340}
{Mailbox.send MB 99}
{Mailbox.send MB 33}
{Test {Mailbox.receive MB [{Equals 99}#Id] infinity} '==' 99}
{Test {Mailbox.receive MB [Else#Id] infinity} '==' 11}
{Test {Mailbox.receive MB [{Equals 33}#Id] infinity} '==' 33}
{Test {Mailbox.receive MB [Else#Id] infinity} '==' 340}
```

Figure 6: Testing for the textbook's problem 7 (Problem 7 on page 7), part 1.

```
{StartTesting 'Receive with multiple guards'}
{Mailbox.send MB 11}
{Mailbox.send MB 340}
{Mailbox.send MB 99}
{Mailbox.send MB 33}
{Mailbox.send MB 10}
{Mailbox.send MB 20}
{Mailbox.send MB 30}
{Test {Mailbox.receive MB [{Equals 99}#Id] infinity} '==' 99}
{Test {Mailbox.receive MB [{Equals 99}#Id Else#Id] infinity} '==' 11}
{Test {Mailbox.receive MB [{Equals 4}#Id {LessThan 34}#Id Else#Id] infinity}
 '==' 340}
{Test {Mailbox.receive MB [Else#Id Else#{Add 7}] infinity} '==' 33}
{Test {Mailbox.receive MB [{Equals 30}#Id {Equals 20}#Id {Equals 10}#{Add 7}]
       infinity}
 '==' 17}
{Test {Mailbox.receive MB [{Equals 30}#Id {Equals 20}#Id {Equals 10}#{Add 7}]
       infinity}
 '==' 20}
{Test {Mailbox.receive MB [{Equals 30}#Id {Equals 20}#Id {Equals 10}#{Add 7}]
       infinity}
 '==' 30}

% Note that receive in Erlang must be done in a process's mailbox, so,
% while they can be nested there aren't multiple receive's happening
% simultaneously.
%% But sends can come from other processes, so can be in threads.
{StartTesting 'Receive that blocks, unblocked by send'}
thread {Delay 1000} {Mailbox.send MB 99} end
thread {Delay 3000} {StartTesting 'sending 5'} {Mailbox.send MB 5} end
thread {Delay 6000} {StartTesting 'sending 72'} {Mailbox.send MB 72} end
{Test {Mailbox.receive MB [{Equals 99}#Id] infinity} '==' 99}
{Test {Mailbox.receive MB [{Equals 72}#Id] infinity} '==' 72}
{Test {Mailbox.receive MB [{Equals 5}#Id] infinity} '==' 5}

%% Note that timeout procedures (like Timedout) are zero-argument procedures
{StartTesting 'Receive with nonzero timeout'}
thread {Delay 1000} {Mailbox.send MB 99} end
thread {Delay 3000} {StartTesting 'sending 5'} {Mailbox.send MB 5} end
thread {Delay 6000} {StartTesting 'sending 72'} {Mailbox.send MB 72} end
{Test {Mailbox.receive MB [{Equals 99}#Id] 20#Timedout} '==' timedout}
{Test {Mailbox.receive MB [{Equals 72}#Id] 2000#Timedout} '==' timedout}
{Test {Mailbox.receive MB [{Equals 5}#Id] infinity} '==' 5}
{Test {Mailbox.receive MB [{Equals 72}#Id] 40000#Timedout} '==' 72}
{Test {Mailbox.receive MB [{Equals 99}#Id] infinity} '==' 99}

{StartTesting 'Receive with zero timeout'}
{Mailbox.send MB 27}
{Mailbox.send MB 42}
{Test {Mailbox.receive MB [{Equals 42}#Id] 0#Timedout} '==' 42}
{Test {Mailbox.receive MB [{Equals 27}#Id] 0#Timedout} '==' 27}
{Test {Mailbox.receive MB [{Equals 313}#Id] 0#Timedout} '==' timedout}

{StartTesting 'All done'}
```

Figure 7: Second part of testing for the textbook's problem 7 (Problem 7 on page 7).

Comparisons Among Models

8. (10 points) [EvaluateModels]

   Would it have been easier to use the message passing model to solve the square root approximation and differentiation exercises (numbers 13, 14, and 16) of homework 4? Briefly explain your answer. (You don't have to actually solve these problems with the message passing model.)

9. (10 points) [EvaluateModels]

   Suppose you are asked to program a simulation of an agent-based auction system for someone doing research in economics. This system consists of several independent agents, each of which must communicate with a central auction server to evaluate merchandise, place bids, and make payments.

   Among the programming models we studied this semester, What is the most restrictive (i.e., the least expressive or smallest) programming model that can practically be used program the overall structure of such a system? Briefly justify your answer.

10. (25 points) [EvaluateModels]

    Update your table from homework 4 that lists all the different programming techniques, the characteristics of problems that are best solved with these techniques (i.e., when to use the techniques), and the name of at least one example of that technique. In this update, take into account comments from the staff from homework 4, and add entries for the new row on "message passing."

programming technique	problem characteristics	example(s)
recursion (over grammars)		
higher-order functions		
stream programming		
lazy functions		
message passing		

Extra Credit Problems

11. (20 points; extra credit) [Concepts] [UseModels]

    Using read-only views (see Sections 3.7.5 and 13.1.14 of our textbook [VH04]), fix your solution to Problem 2 on page 3 so that your code has the same behavior for improper uses of the stream argument passed to `MyNewPort` as does Oz's built-in `NewPort` primitive.

    You must show by writing your own tests that your code has the required same behavior for such uses. Testing is up to you and an important part of this exercise.

12. (60 points; extra credit) [UseModels]

    Do part (a) of problem 3 in section 5.9 of the textbook [VH04] (Fault tolerance for the lift control system).

    You can get the code for the book's figures from the textbook's supplementary web site, which is: `http://www.info.ucl.ac.be/~pvr/ds/mitbook.html` or more directly from the book's supplementary web directory: `http://www.info.ucl.ac.be/~pvr/bookfigures/` or even more directly from WebCT.

    Note that in part (a) "when the floor is called" refers to `call` messages sent to a `Floor` port object.

    Feel free to make other changes to the code to make it more easy for you to understand and more sensible.

    Since deciding when the code works correctly is not easy, you are responsible for your own testing for this problem. In essence, you should set up a situation where a lift gets blocked on some floor, and make sure that the floor's timer is reset and that the blocked lift's schedule is distributed to other lifts and that the floors aren't calling the blocked lift. You can should add extra outputs to see whether the system is functioning as you intend.

Now for some hints.

To get started, overall it's useful to understand how the state diagrams (like Figure 5.7) relate to the code (like Figure 5.8's `Timer` class). Think about the design at the level of the state diagrams, and then make the corresponding changes to the code. You may find it useful to look at the diagrams to understand the code (and vice versa). Also don't be afraid to introduce new state (or parts of state) and new kinds of messages.

To get started coding, look at the `Floor` component in Figure 5.10. When it's in the `doorsopen` state and it gets a `call` message is when the lift is being blocked at a floor. You'll have to change the code in that spot to start with. Get it to work reset the timer, perhaps by having the floor remember how many `stoptimer` messages it should wait for before telling the lift to close the doors. Then add to the `Lift` a way to get the schedule, and then figure out what component should ask for it and redistribute the schedule to other lifts.

Be sure to mark, with comments, any changes you made to the code to solve the problem.

References

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.