```
fun {Timer}
   {NewPortObject2
    proc {$ Msg}
       case Msg of starttimer(T Pid) then
          thread {Delay T} {Send Pid stoptimer} end
       end
    end}
end

fun {Controller Init}
   Tid={Timer}
   Cid={NewPortObject Init
        fun {$ Msg state(Motor F Lid)}
           case Motor
           of running then
              case Msg
              of stoptimer then
                 {Send Lid ´at´(F)}
                 state(stopped F Lid)
              end
           [] stopped then
              case Msg
              of step(Dest) then
                 if F==Dest then
                    state(stopped F Lid)
                 elseif F<Dest then
                    {Send Tid starttimer(5000 Cid)}
                    state(running F+1 Lid)
                 else % F>Dest
                    {Send Tid starttimer(5000 Cid)}
                    state(running F-1 Lid)
                 end
              end
           end
        end}
in Cid end
```

Figure 5.8: Implementation of the timer and controller components

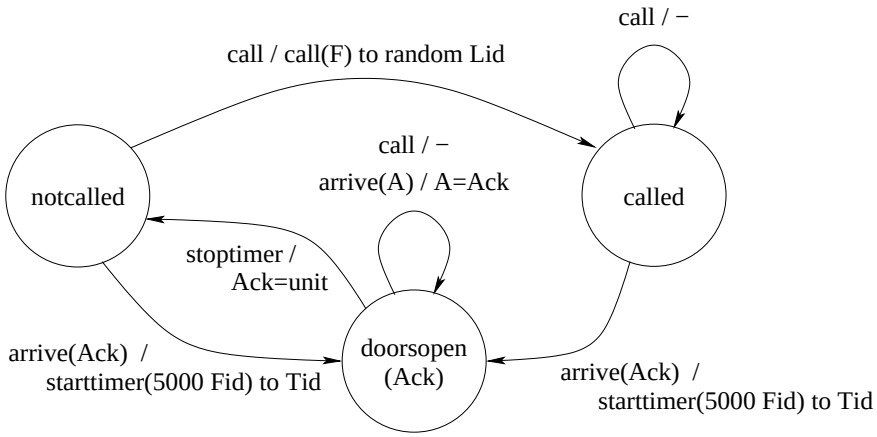

Figure 5.9: State diagram of a floor

The lift Lifts are the most complicated of all. Figure 5.11 gives the state diagram of lift Lid. Each lift can be in one of four states: empty schedule and lift stopped (idle), nonempty schedule and lift moving past a given floor, waiting for doors when moving past a scheduled floor, and waiting for doors when idle at a called floor. The way to understand this figure is to trace through some execution scenarios. For example, here is a simple scenario. A user presses the call button at floor 1. The floor then sends `call(1)` to a lift. The lift receives this and sends `step(1)` to the controller. Say the lift is currently at floor 3. The controller sends `´at´(2)` to the lift, which then sends `step(1)` to the controller again. The controller sends `´at´(1)` to the lift, which then sends `arrive(Ack)` to floor 1 and waits until the floor acknowledges that it can leave.

Each lift can receive a `call(N)` message and an `´at´(N)` message. The lift can send an `arrive(Ack)` message to a floor and a `step(Dest)` message to its controller. After sending the `arrive(Ack)` message, the lift waits until the floor acknowledges that the door actions have finished. The acknowledgement is done by using the dataflow variable `Ack` as a one-shot wire. The floor sends an acknowledgement by binding `Ack=unit` and the lift waits with `{Wait Ack}`.

The source code of the lift component is shown in Figure 5.12. It uses a series of **`if`** statements to implement the conditions for the different transitions. It uses `Browse` to display when a lift will go to a called floor and when the lift arrives at a called floor. The function `{ScheduleLast L N}` implements the scheduler: it adds `N` to the end of the schedule `L` and returns the new schedule.

The building We have now specified the complete system. It is instructive to trace through the execution by hand, following the flow of control in the floors, lifts, controllers, and timers. For example, say that there are 10 floors and 2 lifts. Both lifts are on floor 1 and floors 9 and 10 each call a lift. What are the possible executions of the system? Let us define a compound component that creates a building with `FN` floors and `LN` lifts:

```
fun {Floor Num Init Lifts}
   Tid={Timer}
   Fid={NewPortObject Init
        fun {$ Msg state(Called)}
           case Called
           of notcalled then Lran in
              case Msg
              of arrive(Ack) then
                 {Browse ´Lift at floor ´#Num#´: open doors´}
                 {Send Tid starttimer(5000 Fid)}
                 state(doorsopen(Ack))
              [] call then
                 {Browse ´Floor ´#Num#´ calls a lift!´}
                 Lran=Lifts.(1+{OS.rand} mod {Width Lifts})
                 {Send Lran call(Num)}
                 state(called)
              end
           [] called then
              case Msg
              of arrive(Ack) then
                 {Browse ´Lift at floor ´#Num#´: open doors´}
                 {Send Tid starttimer(5000 Fid)}
                 state(doorsopen(Ack))
              [] call then
                 state(called)
              end
           [] doorsopen(Ack) then
              case Msg
              of stoptimer then
                 {Browse ´Lift at floor ´#Num#´: close doors´}
                 Ack=unit
                 state(notcalled)
              [] arrive(A) then
                 A=Ack
                 state(doorsopen(Ack))
              [] call then
                 state(doorsopen(Ack))
              end
           end
        end}
in Fid end
```

Figure 5.10: Implementation of the floor component

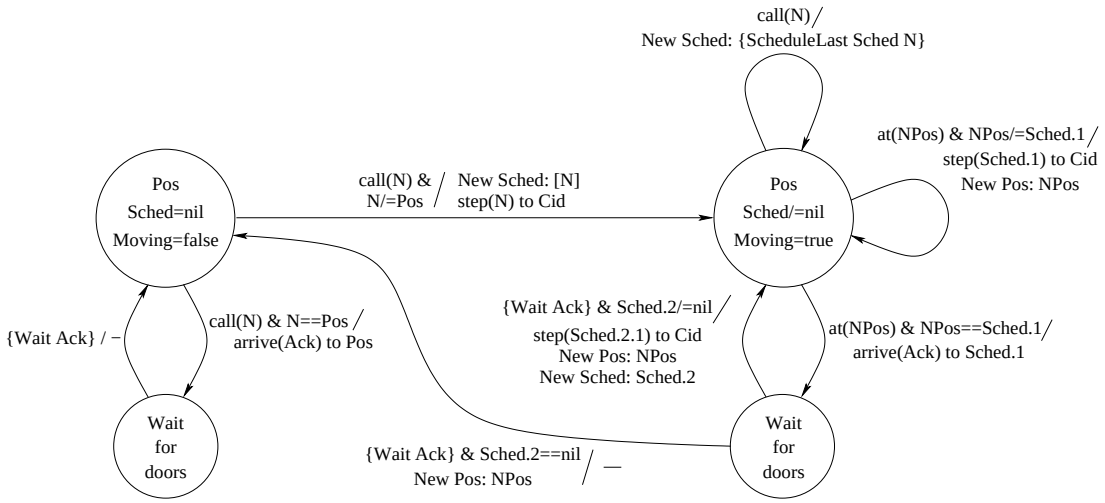

Figure 5.11: State diagram of a lift

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

This uses `MakeTuple` to create a new tuple containing unbound variables. Each component instance will run in its own thread. Here is a sample execution:

```
declare F L in
{Building 20 2 F L}
{Send F.20 call}
{Send F.4 call}
{Send F.10 call}
{Send L.1 call(4)}
```

This makes the lifts move around in a building with 20 floors and 2 lifts.

Reasoning about the lift control system To show that the lift works correctly, we can reason about its invariant properties. For example, an `´at´(_)` message can only be received when `Sched\=nil`. This is a simple invariant that can be proved easily from the fact that `´at´` and `step` messages occur in pairs. It is easy to see by inspection that a `step` message is always done when the lift goes into a state where `Sched\=nil`, and that the only transition out of this

```
fun {ScheduleLast L N}
   if L\=nil andthen {List.last L}==N then L
   else {Append L [N]} end
end

fun {Lift Num Init Cid Floors}
   {NewPortObject Init
    fun {$ Msg state(Pos Sched Moving)}
       case Msg
       of call(N) then
          {Browse ´Lift ´#Num#´ needed at floor ´#N}
          if N==Pos andthen {Not Moving} then
             {Wait {Send Floors.Pos arrive($)}}
             state(Pos Sched false)
          else Sched2 in
             Sched2={ScheduleLast Sched N}
             if {Not Moving} then
                {Send Cid step(N)} end
             state(Pos Sched2 true)
          end
       [] ´at´(NewPos) then
          {Browse ´Lift ´#Num#´ at floor ´#NewPos}
          case Sched
          of S|Sched2 then
             if NewPos==S then
                {Wait {Send Floors.S arrive($)}}
                if Sched2==nil then
                   state(NewPos nil false)
                else
                   {Send Cid step(Sched2.1)}
                   state(NewPos Sched2 true)
                end
             else
                {Send Cid step(S)}
                state(NewPos Sched Moving)
             end
          end
       end
    end}
end
```

Figure 5.12: Implementation of the lift component

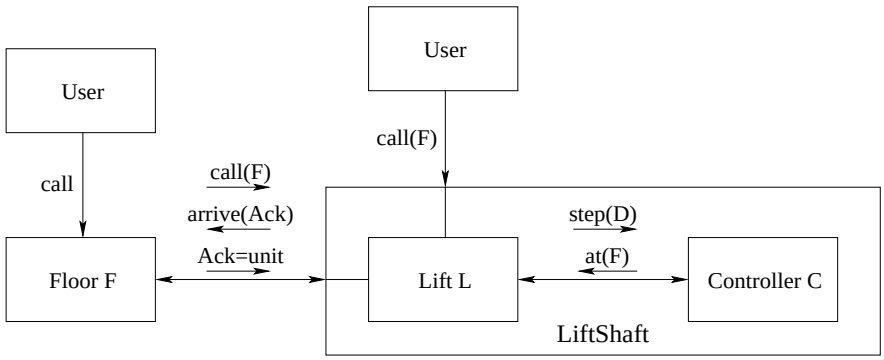

Figure 5.13: Hierarchical component diagram of the lift control system

state (triggered by a `call` message) preserves the invariant. Another invariant is that successive elements of a schedule are always different (can you prove this?).

5.4.5 Improvements to the lift control system

The lift control system of the previous section is somewhat naive. In this section we will indicate five ways in which it can be improved: by using component composition to make it hierarchical, by improving how it opens and closes doors, by using negotiation to find the best lift to call, by improving scheduling to reduce the amount of lift motion, and by handling faults (lifts that stop working). We leave the last three improvements as exercises for the reader.

Hierarchical organization

Looking at the component diagram of Figure 5.5, we see that each controller talks only with its corresponding lift. This is visible also in the definition of `Building`. This means that we can improve the organization by combining controller and lift into a compound component, which we call a *lift shaft*. Figure 5.13 shows the updated component diagram with a lift shaft. We implement this by defining the component `LiftShaft` as follows:

```
fun {LiftShaft I state(F S M) Floors}
   Cid={Controller state(stopped F Lid)}
   Lid={Lift I state(F S M) Cid Floors}
in Lid end
```

Then the `Building` procedure can be simplified:

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

```
      for I in 1..FN do
         Floors.I={Floor I state(notcalled) Lifts}
      end
   end
```

The encapsulation provided by `LiftShaft` improves the modularity of the program. We can change the internal organization of a lift shaft without changing its interface.

Improved door management

Our system opens all doors at a floor when the first lift arrives and closes them a fixed time later. So what happens if a lift arrives at a floor when the doors are already open? The doors may be just about to close. This behavior is unacceptable for a real lift. We need to improve our lift control system so that each lift has its own set of doors.

Improved negotiation

We can improve our lift control system so that the floor picks the closest lift instead of a random lift. The idea is for the floor to send messages to all lifts asking them to give an estimate of the time it would take to reach the floor. The floor can then pick the lift with the least time. This is an example of a simple negotiation protocol.

Improved scheduling

We can improve the lift scheduling. For example, assume the lift is moving from floor 1 to floor 5 and is currently at floor 2. 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. The improved algorithm moves in one direction until there are no more floors to stop at and then changes direction. Variations on this algorithm, which is called the *elevator algorithm* for obvious reasons, are used to schedule the head movement of a hard disk. With this scheduler we can have two call buttons to call upgoing and downgoing lifts separately.

Fault tolerance

What happens if part of the system stops working? For example, a lift can be out of order, either because of maintenance, because it has broken down, or simply because someone is blocking open the doors at a particular floor. Floors can also be "out of order", e.g., a lift may be forbidden to stop at a floor for some reason. We can extend the lift control system to handle these cases. The basic ideas are explained in the Exercises.

5.5 Using the message-passing concurrent model directly

The message-passing model can be used in other ways rather than just programming with port objects. One way is to program directly with threads, procedures, ports, and dataflow variables. Another way is to use other abstractions. This section gives some examples.

5.5.1 Port objects that share one thread

It is possible to run many port objects on just one thread, if the thread serializes all their messages. This can be more efficient than using one thread per port object. According to David Wood of Symbian Ltd., this solution was used in the operating system of the Psion Series 3 palmtop computers, where memory is at a premium [210]. Execution is efficient since no thread scheduling has to be done. Objects can access shared data without any particular precautions since all the objects run in the same thread. The main disadvantage is that synchronization is harder. Execution cannot wait inside an object for a calculation done in another object. Attempting this will block the program. This means that programs must be written in a particular style. State must be either global or stored in the message arguments, not in the objects. Messages are a kind of continuation, i.e., there is no return. Each object execution finishes by sending a message.

Figure 5.14 defines the abstraction `NewPortObjects`. It sets up the single thread and returns two procedures, `AddPortObject` and `Call`:

- `{AddPortObject PO Proc}` adds a new port object with name `PO` to the thread. The name should be a literal or a number. Any number of new port objects can be added to the thread.
- `{Call PO Msg}` asynchronously sends the message `Msg` to the port object `PO`. All message executions of all port objects are executed in the single thread. Exceptions raised during message execution are simply ignored.

Note that the abstraction stores the port objects' procedures in a record and uses `AdjoinAt` to extend this record when a new port object is added.

Figure 5.15 gives a screenshot of a small concurrent program, 'Ping-Pong', which uses port objects that share one thread. Figure 5.16 gives the full source code of 'Ping-Pong'. It uses `NewProgWindow`, the simple progress monitor defined in Chapter 10. Two objects are created initially, `pingobj` and `pongobj`. Each object understands two messages, `ping(N)` and `pong(N)`. The `pingobj` object asynchronously sends a `pong(N)` message to the `pongobj` object and vice versa. Each message executes by displaying a text and then continuing execution by sending a message to the other object. The integer argument `N` counts messages by being incremented at each call. Execution is started with the initial call `{Call pingobj ping(0)}`.

```
proc {NewPortObjects ?AddPortObject ?Call}
   Sin P={NewPort Sin}

   proc {MsgLoop S1 Procs}
      case S1
      of msg(I M)|S2 then
         try {Procs.I M} catch _ then skip end
         {MsgLoop S2 Procs}
      [] add(I Proc Sync)|S2 then Procs2 in
         Procs2={AdjoinAt Procs I Proc}
         Sync=unit
         {MsgLoop S2 Procs2}
      [] nil then skip end
   end
in
   proc {AddPortObject I Proc}
   Sync in
      {Send P add(I Proc Sync)}
      {Wait Sync}
   end

   proc {Call I M}
      {Send P msg(I M)}
   end

   thread {MsgLoop Sin procs} end
end
```

Figure 5.14: Defining port objects that share one thread

Figure 5.15: Screenshot of the 'Ping-Pong' program

```
declare AddPortObject Call
{NewPortObjects AddPortObject Call}

InfoMsg={NewProgWindow "See ping-pong"}

fun {PingPongProc Other}
   proc {$ Msg}
      case Msg
      of ping(N) then
         {InfoMsg "ping("#N#")"}
         {Call Other pong(N+1)}
      [] pong(N) then
         {InfoMsg "pong("#N#")"}
         {Call Other ping(N+1)}
      end
   end
end

{AddPortObject pingobj {PingPongProc pongobj}}
{AddPortObject pongobj {PingPongProc pingobj}}
{Call pingobj ping(0)}
```

Figure 5.16: The 'Ping-Pong' program: using port objects that share one thread

When the program starts, it creates a window that displays a term of the form `ping(123)` or `pong(123)`, where the integer gives the message count. This monitors execution progress. When the checkbutton is enabled, then each term is displayed for 50 ms. When the checkbutton is disabled, then the messages are passed internally at a much faster rate, limited only by the speed of the Mozart run-time system.[4]

5.5.2 A concurrent queue with ports

The program shown in Figure 5.17 defines a thread that acts as a FIFO queue. The function `NewQueue` returns a new queue `Q`, which is a record `queue(put:PutProc get:GetProc)` that contains two procedures, one for inserting an element in the queue and one for fetching an element from the queue. The queue is implemented with two ports. The use of dataflow variables makes the queue insensitive to the relative arrival order of `Q.get` and `Q.put` requests. For example, the `Q.get` requests can arrive even when the queue is empty. To insert an element `X`, call `{Q.put X}`. To fetch an element in `Y`, call `{Q.get Y}`.

The program in Figure 5.17 is almost correct, but it does not work because port streams are read-only variables. To see this, try the following sequence of

[4]With Mozart 1.3.0 on a 1 GHz PowerPC processor (PowerBook G4), the rate is about 300000 asynchronous method calls per second.

```
fun {NewQueue}
   Given GivePort={NewPort Given}
   Taken TakePort={NewPort Taken}
in
   Given=Taken
   queue(put:proc {$ X} {Send GivePort X} end
         get:proc {$ X} {Send TakePort X} end)
end
```

Figure 5.17: Queue (naive version with ports)

statements:

```
declare Q in
thread Q={NewQueue} end
{Q.put 1}
{Browse {Q.get $}}
{Browse {Q.get $}}
{Browse {Q.get $}}
{Q.put 2}
{Q.put 3}
```

The problem is that `Given=Taken` tries to impose equality between two read-only variables, i.e., bind them. But a read-only variable can only be read and not bound. So the thread defining the queue will suspend in the statement `Given=Taken`. We can fix the problem by defining a procedure `Match` and running it in its own thread, as shown in Figure 5.18. You can verify that the above sequence of statements now works.

Let us look closer to see why the correct version works. Doing a series of put operations:

```
{Q.put I0} {Q.put I1} ... {Q.put In}
```

incrementally adds the elements `I0`, `I1`, ..., `In`, to the stream `Given`, resulting in:

```
I0|I1|...|In|F1
```

where `F1` is a read-only variable. In the same way, doing a series of get operations:

```
{Q.get X0} {Q.get X1} ... {Q.get Xn}
```

adds the elements `X0`, `X1`, ..., `Xn` to the stream `Taken`, resulting in:

```
X0|X1|...|Xn|F2
```

where `F2` is another read-only variable. The call `{Match Given Taken}` binds the `Xi`'s to `Ii`'s and blocks again for `F1=F2`.

This concurrent queue is completely *symmetric* with respect to inserting and retrieving elements. That is, `Q.put` and `Q.get` are defined in exactly the same way. Furthermore, because they use dataflow variables to reference queue elements, these operations never block. This gives the queue the remarkable prop-

```
fun {NewQueue}
   Given GivePort={NewPort Given}
   Taken TakePort={NewPort Taken}
   proc {Match Xs Ys}
      case Xs # Ys
      of (X|Xr) # (Y|Yr) then
         X=Y {Match Xr Yr}
      [] nil # nil then skip
      end
   end
in
   thread {Match Given Taken} end
   queue(put:proc {$ X} {Send GivePort X} end
         get:proc {$ X} {Send TakePort X} end)
end
```

Figure 5.18: Queue (correct version with ports)

erty that it can be used to insert and retrieve elements *before* the elements are known. For example, if you do a `{Q.get X}` when there are no elements in the queue, then an unbound variable is returned in `X`. The next element that is inserted will be bound to `X`. To do a blocking retrieval, i.e., one that waits when there are no elements in the queue, the call to `Q.get` should be followed by a `Wait`:

```
{Q.get X}
{Wait X}
```

Similarly, if you do `{Q.put X}` when `X` is unbound, i.e., when there is no element to insert, then the unbound variable `X` will be put in the queue. Binding `X` will make the element known. To do an insert only when the element is known, the call to `Q.put` should be preceded by a `Wait`:

```
{Wait X}
{Q.put X}
```

We have captured the essential asymmetry between put and get: it is in the `Wait` operation. Another way to see this is that put and get reserve *places* in the queue. The reservation can be done independent of whether the values of the elements are known or not.

Attentive readers will see that there is an even simpler solution to the problem of Figure 5.17. The procedure `Match` is not really necessary. It is enough to run `Given=Taken` in its own thread. This is because the unification algorithm does exactly what `Match` does.[5]

[5]This FIFO queue design was first given by Denys Duchier.

5.5.3 A thread abstraction with termination detection

"Ladies and gentlemen, we will be arriving shortly in Brussels Midi station, where this train terminates."
– Announcement, Thalys high-speed train, Paris-Brussels line, January 2002

Thread creation with **thread** ⟨stmt⟩ **end** can itself create new threads during the execution of ⟨stmt⟩. We would like to detect when all these new threads have terminated. This does not seem easy: new threads may themselves create new threads, and so forth. A termination detection algorithm like the one of Section 4.4.3 is needed. The algorithm of that section requires explicitly passing variables between threads. We require a solution that is encapsulated, i.e., it does not have this awkwardness. To be precise, we require a procedure `NewThread` with the following properties:

- The call `{NewThread P SubThread}` creates a new thread that executes the zero-argument procedure `P`. It also returns a one-argument procedure `SubThread`.
- During the execution of `P`, new threads can be created by calling `{SubThread P1}`, where the zero-argument procedure `P1` is the thread body. We call these *subthreads*. `SubThread` can be called recursively, that is, inside threads created with `SubThread`.
- The `NewThread` call returns after the new thread and all subthreads have terminated.

That is, there are three ways to create a new thread:

```
thread ⟨stmt⟩ end
{NewThread proc {$} ⟨stmt⟩ end SubThread}
{SubThread proc {$} ⟨stmt⟩ end}
```

They have identical behavior except for `NewThread`, which has a different termination behavior. `NewThread` can be defined using the message-passing model as shown in Figure 5.19. This definition uses a port. When a subthread is created, then 1 is sent to the port. When a subthread terminates, then -1 is sent. The procedure `ZeroExit` accumulates a running total of these numbers. If the total ever reaches zero, then all subthreads have terminated and `ZeroExit` returns.

We can prove that this definition is correct by using invariant assertions. Consider the following assertion: "the sum of the elements on the port's stream is greater than or equal to the number of active threads." When the sum is zero, this implies that the number of active threads is zero as well. We can use induction to show that the assertion is true at every part of every possible execution, starting from the call to `NewThread`. It is clearly true when `Newthread` starts since both numbers are zero. During an execution, there are four relevant actions: sending

```
local
   proc {ZeroExit N Is}
      case Is of I|Ir then
         if N+I\=0 then {ZeroExit N+I Ir} end
      end
   end
in
   proc {NewThread P ?SubThread}
      Is Pt={NewPort Is}
   in
      proc {SubThread P}
         {Send Pt 1}
         thread
            {P} {Send Pt ~1}
         end
      end
      {SubThread P}
      {ZeroExit 0 Is}
   end
end
```

Figure 5.19: A thread abstraction with termination detection

+1, sending -1, starting a thread, and terminating a thread. By inspection of the program, each of these actions keeps the assertion true. (We can assume without loss of generality that thread termination occurs just before sending -1, since the thread then no longer executes any part of the user program.)

This definition of `NewThread` has two restrictions. First, `P` and `P1` should always call `SubThread` to create subthreads, never any other operation (such as **`thread ... end`** or a `SubThread` created elsewhere). Second, `SubThread` should not be called anywhere else in the program. The definition can be extended to relax these restrictions or to check them. We leave these tasks as exercises for the reader.

An issue about port send semantics

We know that the `Send` operation is *asynchronous*, that is, it completes immediately. The termination detection algorithm relies on another property of `Send`: that `{Send Pt 1}` (in the parent thread) arrives *before* `{Send Pt ~1}` (in the child thread). Can we assume that sends in different threads behave in this way? Yes we can, *if* we are sure the `Send` operation reserves a slot in the port stream. Look back to the semantics we have defined for ports in the beginning of the chapter: the `Send` operation does indeed put its argument in the port stream.

```
proc {ConcFilter L F ?L2}
   Send Close
in
   {NewPortClose L2 Send Close}
   {Barrier
      {Map L
         fun {$ X}
            proc {$}
               if {F X} then {Send X} end
            end
         end}}
   {Close}
end
```

Figure 5.20: A concurrent filter without sequential dependencies

We call this the *slot-reserving* semantics of `Send`.[6]

Unfortunately, this semantics is not the right one in general. We really want an *eventual slot-reserving* semantics, where the `Send` operation might not immediately reserve a slot but we are sure that it will eventually. Why is this semantics "right"? It is because it is the natural behavior of a distributed system, where a program is spread out over more than one process and processes can be on different machines. A `Send` can execute on a different process than where the port stream is constructed. Doing a `Send` does not immediately reserve a slot because the slot might be on a different machine (remember that the `Send` should complete *immediately*)! All we can say is that doing a `Send` will *eventually* reserve a slot.

With the "right" semantics for `Send`, our termination detection algorithm is incorrect since `{Send Pt ~1}` might arrive before `{Send Pt 1}`. We can fix the problem by defining a slot-reserving port in terms of an eventual slot-reserving port:

```
proc {NewSPort ?S ?SSend}
S1 P={NewPort S1} in
  proc {SSend M} X in {Send P M#X} {Wait X} end
  thread S={Map S1 fun {$ M#X} X=unit M end} end
end
```

`NewSPort` behaves like `NewPort`. If `NewPort` defines an eventual slot-reserving port, then `NewSPort` will define a slot-reserving port. Using `NewSPort` in the termination detection algorithm will ensure that it is correct in case we use the "right" port semantics.

[6] This is sometimes called a *synchronous* `Send`, because it only completes when the message is delivered to the stream. We will avoid this term because the concept of "delivery" is not clear. For example, we might want to talk about delivering a message to an application process instead of a stream.

5.5.4 Eliminating sequential dependencies

Let us examine how to remove useless sequential dependencies between different parts of a program. We take as example the procedure `{Filter L F L2}`, which takes a list `L` and a one-argument boolean function `F`. It outputs a list `L2` that contains the elements `X` of `L` for which `{F X}` is true. This is a library function (it is part of the `List` module) that can be defined declaratively as follows:

```
fun {Filter L F}
   case L
   of nil then nil
   [] X|L2 then
      if {F X} then X|{Filter L2 F} else {Filter L2 F} end
   end
end
```

or equivalently, using the loop syntax:

```
fun {Filter L F}
   for X in L collect:C do
      if {F X} then {C X} end
   end
end
```

This definition is efficient, but it introduces sequential dependencies: `{F X}` can be calculated only after it has been calculated for all elements of `L` before `X`. These dependencies are introduced because all calculations are done sequentially in the same thread. But these dependencies are not really necessary. For example, in the call:

```
{Filter [A 5 1 B 4 0 6] fun {$ X} X>2 end Out}
```

it is possible to deduce immediately that `5`, `4`, and `6` will be in the output, without waiting for `A` and `B` to be bound. Later on, if some other thread does `A=10`, then `10` could be added to the result immediately.

We can write a new version of `Filter` that avoids these dependencies. It constructs its output incrementally, as the input information arrives. We use two building blocks:

- *Concurrent composition* (see Section 4.4.3). The procedure `Barrier` implements concurrent composition: it creates a concurrent task for each list element and waits until all are finished.

- *Asynchronous channels* (ports, see earlier in this chapter). The procedure `NewPortClose` implements a port with a send and a close operation. Its definition is given in the supplements file on the book's Web site. The close operation terminates the port's stream with `nil`.

Figure 5.20 gives the definition. It first creates a port whose stream is the output list. Then `Barrier` is called with a list of procedures, each of which adds `X` to

the output list if `{F X}` is true. Finally, when all list elements are taken care of, the output list is ended by closing the port.

Is `ConcFilter` declarative? As it is written, certainly not, since the output list can appear in any order (an observable nondeterminism). It can be made declarative by hiding this nondeterminism, for example by sorting the output list. There is another way, using the properties of ADTs. If the rest of the program does not depend on the order (e.g., the list is a representation of a set data structure), then `ConcFilter` can be treated as if it were declarative. This is easy to see: if the list were in fact hidden *inside* a set ADT, then `ConcFilter` would be deterministic and hence declarative.

5.6 The Erlang language

The Erlang language was developed by Ericsson for telecommunications applications, in particular, for telephony [9, 206]. Its implementation, the Ericsson OTP (Open Telecom Platform), features fine-grained concurrency (efficient threads), extreme reliability (high performance software fault tolerance), and hot code replacement ability (update software while the system is running). It is a high-level language that hides the internal representation of data and does automatic memory management. It has been used successfully in several Ericsson products.

5.6.1 Computation model

The Erlang computation model has an elegant layered structure. We first explain the model and then we show how it is extended for distribution and fault tolerance.

The Erlang computation model consists of entities called processes, similar to port objects, that communicate through message passing. The language can be divided into two layers:

- **Functional core**. Port objects are programmed in a dynamically-typed strict functional language. Each port object contains one thread that runs a recursive function whose arguments are the thread's state. Functions can be passed in messages.

- **Message passing extension**. Threads communicate by sending messages to other threads asynchronously in FIFO order. Each thread has a unique identifier, its PID, which is a constant that identifies the receiving thread, but can also be embedded in data structures and messages. Messages are values in the functional core. They are put in the receiving thread's *mailbox*. Receiving can be blocking or nonblocking. The receiving thread uses pattern matching to wait for and then remove messages that have a given form from its mailbox, without disturbing the other messages. This means that messages are not necessarily treated in the order that they are sent.

A port object in Erlang consists of a thread associated with one mailbox. This is called a *process* in Erlang terminology. A process that spawns a new process specifies which function should be initially executed inside it.

Extensions for distribution and fault tolerance

The centralized model is extended for distribution and fault tolerance:

- **Transparent distribution**. Processes can be on the same machine or on different machines. A single machine environment is called a *node* in Erlang terminology. In a program, communication between local or remote processes is written in exactly the same way. The PID encapsulates the destination and allows the run-time system to decide whether to do a local or remote operation. Processes are stationary; this means that once a process is created in a node it remains there for its entire lifetime. Sending a message to a remote process requires exactly one network operation, i.e., no intermediate nodes are involved. Processes can also be created at remote nodes. Programs are network transparent, i.e., they give the same result no matter on which nodes the processes are placed. Programs are network aware since the programmer has complete control of process placement and can optimize it according to the network characteristics.

- **Failure detection**. A process can be set up to detect faults in another process. In Erlang terminology this is called *linking* the two processes. When the second process fails, a message is sent to the first, which can receive it. This failure detection ability allows many fault-tolerance mechanisms to be programmed entirely in Erlang.

- **Persistence**. The Erlang run-time system comes with a database, called Mnesia, that helps to build highly available applications.

We can summarize by saying that Erlang's computation model (port objects without mutable state) is strongly optimized for building fault-tolerant distributed systems. The Mnesia database compensates for the lack of a general mutable store. A typical example of a product built using Erlang is Ericsson's AXD301 ATM switch, which provides telephony over an ATM network. The AXD301 handles 30-40 million calls per week with a reliability of 99.9999999% (about 30 ms downtime per year) and contains 1.7 million lines of Erlang [8].

5.6.2 Introduction to Erlang programming

To give a taste of Erlang, we give some small Erlang programs and show how to do the same thing in the computation models of this book. The programs are mostly taken from the Erlang book [9]. We show how to write functions and concurrent programs with message passing. For more information on Erlang programming, we highly recommend the Erlang book.

A simple function

The core of Erlang is a strict functional language with dynamic typing. Here is a simple definition of the factorial function:

```
factorial(0) -> 1;
factorial(N) when N>0 -> N*factorial(N-1).
```

This example introduces the basic syntactic conventions of Erlang. Function names are in lowercase and variable identifiers are capitalized. Variable identifiers are bound to values when defined, which means that Erlang has a value store. An identifier's binding cannot be changed; it is single assignment, just as in the declarative model. These conventions are inherited from Prolog, in which the first Erlang implementation (an interpreter) was written.

Erlang functions are defined by clauses; each clause has a head (with a pattern and optional guard) and a body. The patterns are checked in order starting with the first clause. If a pattern matches, its variables are bound and the clause body is executed. The optional guard is a boolean function that has to return `true`. All the variable identifiers in the pattern must be different. If a pattern does not match, then the next clause is tried. We can translate the factorial as follows in the declarative model:

```
fun {Factorial N}
   case N
   of 0 then 1
   [] N andthen N>0 then N*{Factorial N-1}
   end
end
```

The **`case`** statement does pattern matching exactly as in Erlang, with a different syntax.

Pattern matching with tuples

Here is a function that does pattern matching with tuples:

```
area({square, Side}) ->
    Side*Side;
area({rectangle, X, Y}) ->
    X*Y;
area({circle, Radius}) ->
    3.14159*Radius*Radius;
area({triangle, A, B, C}) ->
    S=(A+B+C)/2;
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

This uses the square root function `sqrt` defined in the module `math`. This function calculates the area of a plane shape. It represents the shape by means of a tuple

that identifies the shape and gives its size. Tuples in Erlang are written with curly braces: `{square, Side}` would be written as `square(Side)` in the declarative model. In the declarative model, the function can be written as follows:

```
fun {Area T}
   case T
   of square(Side) then Side*Side
   [] rectangle(X Y) then X*Y
   [] circle(Radius) then 3.14159*Radius*Radius
   [] triangle(A B C) then S=(A+B+C)/2.0 in
      {Sqrt S*(S-A)*(S-B)*(S-C)}
   end
end
```

Concurrency and message passing

In Erlang, threads are created together with a mailbox that can be used to send messages to the thread. This combination is called a *process*. There are three primitives:

- The `spawn` operation (written as `spawn(M,F,A)`) creates a new process and returns a value (called "process identifier") that can be used to send messages to it. The arguments of `spawn` give the initial function call that starts the process, identified by module `M`, function name `F`, and argument list `A`.

- The `send` operation (written as `Pid!Msg`) asynchronously sends the message `Msg` to the process, which is identified by its process identifier `Pid`. The messages are put in the mailbox, which is a kind of process queue.

- The `receive` operation receives a message from inside the process. It uses pattern matching to pick a message from the mailbox.

Let us take the `area` function and put it inside a process. This makes it into a server that can be called from any other process.

```
-module(areaserver).
-export([start/0, loop/0]).

start() -> spawn(areaserver, loop, []).

loop() ->
    receive
       {From, Shape} ->
           From!area(Shape),
           loop()
    end.
```

This defines the two operations `start` and `loop` in the new module `areaserver`. These two operations are exported outside the module. We need to define them in a module because the `spawn` operation requires the module name as an argument. The `loop` operation repeatedly reads a message (a two-argument tuple `{From, Shape}`) and responds to it by calling `area` and sending the reply to the process `From`. Now let us start a new server and call it:

```
Pid=areaserver:start(),
Pid!{self(), {square, 3.4}},
receive
   Ans -> ...
end,
```

Here `self()` is a language operation that returns the process identifier of the current process. This allows the server to return a reply. Let us write this in the concurrent stateful model:

```
fun {Start}
S AreaServer={NewPort S} in
   thread
      for msg(Ans Shape) in S do
         Ans={Area Shape}
      end
   end
   AreaServer
end
```

Let us again start a new server and call it:

```
Pid={Start}
local Ans in
   {Send Pid msg(Ans square(3.4))}
   {Wait Ans}
   ...
end
```

This example uses the dataflow variable `Ans` to get the reply. This mimics the send to `From` done by Erlang. To do *exactly* what Erlang does, we need to translate the `receive` operation into a computation model of the book. This is a little more complicated. It is explained in the next section.

5.6.3 The `receive` operation

Much of the unique flavor and expressiveness of concurrent programming in Erlang is due to the mailboxes and how they are managed. Messages are taken out of a mailbox with the `receive` operation. It uses pattern matching to pick out a desired message, leaving the other messages unchanged. Using `receive` gives particularly compact, readable, and efficient code. In this section, we implement

`receive` as a linguistic abstraction. We show how to translate it into the computation models of this book. There are two reasons for giving the translation. First, it gives a precise semantics for `receive`, which aids the understanding of Erlang. Second, it shows how to do Erlang-style programming in Oz.

Because of Erlang's functional core, `receive` is an expression that returns a value. The `receive` expression has the following general form [9]:

```
receive
   Pattern1 [when Guard1] -> Body1;
   ...
   PatternN [when GuardN] -> BodyN;
[ after Expr -> BodyT; ]
end
```

The guards (`when` clauses) and the time out (`after` clause) are optional. This expression blocks until a message matching one of the patterns arrives in the current thread's mailbox. It then removes this message, binds the corresponding variables in the pattern, and executes the body. Patterns are very similar to patterns in the **`case`** statement of this book: they introduce new single-assignment variables whose scope ranges over the corresponding body. For example, the Erlang pattern {`rectangle, [X,Y]`} corresponds to the pattern `rectangle([X Y])`. Identifiers starting with lowercase letters correspond to atoms and identifiers starting with capital letters correspond to variables, like the notation of this book. Compound terms are enclosed in braces { and } and correspond to tuples.

The optional `after` clause defines a time out; if no matching message arrives after a number of milliseconds given by evaluating the expression `Expr`, then the time-out body is executed. If zero milliseconds are specified, then the `after` clause is executed immediately if there are no messages in the mailbox.

General remarks

Each Erlang process is translated into one thread with one port. Sending to the process means sending to the port. This adds the message to the port's stream, which represents the mailbox contents. All forms of `receive`, when they complete, either take exactly one message out of the mailbox or leave the mailbox unchanged. We model this by giving each translation of `receive` an input stream and an output stream. All translations have two arguments, `Sin` and `Sout`, that reference the input stream and the output stream. These streams do not appear in the Erlang syntax. After executing a `receive`, there are two possibilities for the value of the output stream. Either it is the same as the input stream or it has one less message than the input stream. The latter occurs if the message matches a pattern.

We distinguish three different forms of `receive` that result in different translations. In each form the translation can be directly inserted in a program and it will behave like the respective `receive`. The first form is translated using the

```
T(receive ... end Sin Sout) ≡
   local
      fun {Loop S T#E Sout}
         case S of M|S1 then
            case M
            of T(Pattern1) then E=S1 T(Body1 T Sout)
            ...
            [] T(PatternN) then E=S1 T(BodyN T Sout)
            else E1 in E=M|E1 {Loop S1 T#E1 Sout}
            end
         end
      end T
   in
      {Loop Sin T#T Sout}
   end
```

Figure 5.21: Translation of `receive` without time out

declarative model. The second form has a time out; it uses the nondeterministic concurrent model (see Section 8.2). The third form is a special case of the second where the delay is zero, which makes the translation much simpler.

First form (without time out)

The first form of the `receive` expression is as follows:

```
receive
   Pattern1 -> Body1;
   ...
   PatternN -> BodyN;
end
```

The `receive` blocks until a message arrives that matches one of the patterns. The patterns are checked in order from `Pattern1` to `PatternN`. We leave out the guards to avoid cluttering up the code. Adding them is straightforward. A pattern can be any partial value; in particular an unbound variable will always cause a match. Messages that do not match are put in the output stream and do not cause the `receive` to complete.

Figure 5.21 gives the translation of the first form, which we will write as T(`receive ... end Sin Sout`). The output stream contains the messages that remain after the `receive` expression has removed the ones it needs. Note that the translation T(`Body T Sout`) of a body that does *not* contain a `receive` expression must bind `Sout=T`.

The `Loop` function is used to manage out-of-order reception: if a message `M` is received that does not match any pattern, then it is put in the output stream and

```
T(receive ... end Sin Sout) ≡
   local
      Cancel={Alarm T(Expr)}
      fun {Loop S T#E Sout}
         if {WaitTwo S Cancel}==1 then
            case S of M|S1 then
               case M
               of T(Pattern1) then E=S1 T(Body1 T Sout)
               ...
               [] T(PatternN) then E=S1 T(BodyN T Sout)
               else E1 in E=M|E1 {Loop S1 T#E1 Sout} end
            end
         else E=S T(BodyT T Sout)
      end T
   in
      {Loop Sin T#T Sout}
   end
```

Figure 5.22: Translation of `receive` with time out

`Loop` is called recursively. `Loop` uses a difference list to manage the case when a `receive` expression contains a `receive` expression.

Second form (with time out)

The second form of the `receive` expression is as follows:

```
receive
   Pattern1 -> Body1;
   ...
   PatternN -> BodyN;
after Expr -> BodyT;
end
```

When the `receive` is entered, `Expr` is evaluated first, giving the integer n. If no match is done after n milliseconds, then the time-out action is executed. If a match is done before n milliseconds, then it is handled as if there were no time out. Figure 5.22 gives the translation.

The translation uses a timer interrupt implemented by `Alarm` and `WaitTwo`. `{Alarm N}`, explained in Section 4.6, is guaranteed to wait for at least n milliseconds and then bind the unbound variable `Cancel` to **`unit`**. `{WaitTwo S Cancel}`, which is defined in the supplements file on the book's Web site, waits simultaneously for one of two events: a message (`S` is bound) and a time out (`Cancel` is bound). It can return 1 if its first argument is bound and 2 if its second argument is bound.

```
T(receive ... end Sin Sout) ≡
   if {IsDet Sin} then
      case Sin of M|S1 then
         case M
         of T(Pattern1) then T(Body1 S1 Sout)
         ...
         [] T(PatternN) then T(BodyN) S1 Sout)
         else T(BodyT Sin Sout) end
      end
   else Sout=Sin end
```

Figure 5.23: Translation of `receive` with zero time out

The Erlang semantics is slightly more complicated than what is defined in Figure 5.22. It guarantees that the mailbox is checked at least once, even if the time out is zero or has expired by the time the mailbox is checked. We can implement this guarantee by stipulating that `WaitTwo` favors its first argument, i.e., that it always returns 1 if its first argument is determined. The Erlang semantics also guarantees that the `receive` is exited quickly after the time out expires. While this is easily guaranteed by an actual implementation, it is not guaranteed by Figure 5.22 since `Loop` could go on forever if messages arrive quicker than the loop iterates. We leave it to the reader to modify Figure 5.22 to add this guarantee.

Third form (with zero time out)

The third form of the `receive` expression is like the second form except that the time-out delay is zero. With zero delay the `receive` is nonblocking. A simpler translation is possible when compared to the case of nonzero time out. Figure 5.23 gives the translation. Using `IsDet`, it first checks whether there is a message that matches any of the patterns. `{IsDet S}`, explained in Section 4.9.3, checks immediately whether `S` is bound or not and returns **`true`** or **`false`**. If there is no message that matches (for example, if the mail box is empty) then the default action `BodyT` is done.

5.7 Advanced topics

5.7.1 The nondeterministic concurrent model

This section explains the nondeterministic concurrent model, which is intermediate in expressiveness between the declarative concurrent model and the message-passing concurrent model. It is less expressive than the message-passing model but in return it has a logical semantics (see Chapter 9).

The nondeterministic concurrent model is the model used by concurrent logic

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Procedure application
thread ⟨s⟩ **end**	Thread creation
{WaitTwo ⟨x⟩ ⟨y⟩ ⟨z⟩}	**Nondeterministic choice**

Table 5.2: The nondeterministic concurrent kernel language

programming [177]. It is sometimes called the *process model* of logic programming, since it models predicates as concurrent computations. It is interesting both for historical reasons and for the insight it gives into practical concurrent programming. We first introduce the nondeterministic concurrent model and show how it solves the stream communication problem of Section 4.7.3. We then show how to implement nondeterministic choice in the declarative concurrent model with exceptions, showing that the latter is at least as expressive as the nondeterministic model.

Table 5.2 gives the kernel language of the nondeterministic concurrent model. It adds just one operation to the declarative concurrent model: a nondeterministic choice that waits for either of two events and nondeterministically returns when one has happened with an indication of which one.

Limitation of the declarative concurrent model

In Section 4.7.3 we saw a fundamental limitation of the declarative concurrent model: stream objects must access input streams in a fixed pattern. Two streams cannot independently feed the same stream object. How can we solve this problem? Consider the case of two client objects and a server object. We can try to solve it by putting a new stream object, a *stream merger*, in between the two clients and the server. The stream merger has two input streams and one output stream. All the messages appearing on each of the input streams will be put on the output stream. Figure 5.24 illustrates the solution. This *seems* to solve our problem: each client sends messages to the stream merger, and the stream merger forwards them to the server. The stream merger is defined as follows:

```
fun {StreamMerger OutS1 OutS2}
   case OutS1#OutS2
   of (M|NewS1)#OutS2 then
       M|{StreamMerger NewS1 OutS2}
```

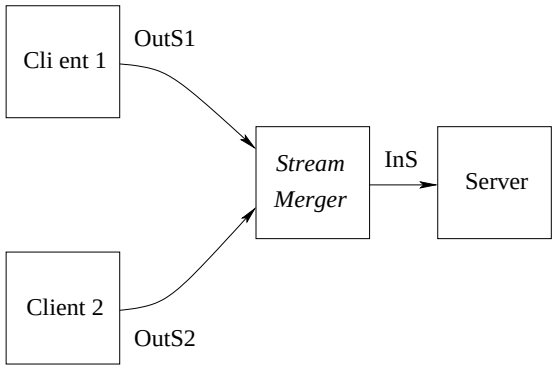

Figure 5.24: Connecting two clients using a stream merger

```
      [] OutS1#(M|NewS2) then
          M|{StreamMerger OutS1 NewS2}
      [] nil#OutS2 then
          OutS2
      [] OutS1#nil then
          OutS1
      end
   end
```

The stream merger is executed in its own thread. This definition handles the case of termination, i.e., when either or both clients terminate. Yet, this solution has a basic difficulty: it does not work! Why not? Think carefully before reading the answer in the footnote.[7]

Adding nondeterministic choice

But this abortive solution has the germs of a working solution. The problem is that the **case** statement only waits on *one* condition at a time. A possible solution is therefore to extend the declarative concurrent model with an operation that allows to wait concurrently on *more than one* condition. We call this operation *nondeterministic choice*. One of the simplest ways is to add an operation that waits concurrently on two dataflow variables being bound. We call this operation `WaitTwo` because it generalizes `Wait`. The function call `{WaitTwo A B}` returns when either `A` or `B` is bound. It returns either 1 or 2. It can return 1 when `A` is bound and 2 when `B` is bound. A simple Mozart definition is given in the supplements file on the book's Web site. The declarative concurrent model extended with `WaitTwo` is called the *nondeterministic concurrent model*.

[7] It is because the **case** statement tests only *one* pattern at a time, and only goes to the next when the previous ones fail. While it is waiting on stream `OutS1`, it cannot accept an input from stream `OutS2`, and vice versa.

Concurrent logic programming

The nondeterministic concurrent model is the basic model of concurrent logic programming, as pioneered by IC-Prolog, Parlog, Concurrent Prolog, FCP (Flat Concurrent Prolog), GHC (Guarded Horn Clauses), and Flat GHC [35, 36, 34, 175, 176, 191]. It is the principal computation model that was used by the Japanese Fifth Generation Project and many other substantial projects in the 1980's [177, 57, 190]. In the nondeterministic concurrent model, it is possible to write a stream merger. Its definition looks as follows:

```
fun {StreamMerger OutS1 OutS2}
   F={WaitTwo OutS1 OutS2}
in
   case F#OutS1#OutS2
   of 1#(M|NewS1)#OutS2 then
       M|{StreamMerger OutS2 NewS1}
   [] 2#OutS1#(M|NewS2) then
       M|{StreamMerger NewS2 OutS1}
   [] 1#nil#OutS2 then
       OutS2
   [] 2#OutS1#nil then
       OutS1
   end
end
```

This style of programming is exactly what concurrent logic programming does. A typical syntax for this definition in a Prolog-like concurrent logic language would be as follows:

```
streamMerger([M|NewS1], OutS2, InS) :- true |
        InS=[M|NewS],
        streamMerger(OutS2, NewS1, NewS).
streamMerger(OutS1, [M|NewS2], InS) :- true |
        InS=[M|NewS],
        streamMerger(NewS2, OutS1, NewS).
streamMerger([], OutS2, InS) :- true |
        InS=OutS2.
streamMerger(OutS1, [], InS) :- true |
        InS=OutS1.
```

This definition consists of four clauses, each of which defines one nondeterministic choice. Keep in mind that syntactically Prolog uses `[]` for `nil` and `[H|T]` for `H|T`. Each clause consists of a guard and a body. The vertical bar `|` separates the guard from the body. A guard does only tests, blocking if a test cannot be decided. A guard must be true for a clause to be choosable. The body is executed only if the clause is chosen. The body can bind output variables.

The stream merger first calls `WaitTwo` to decide *which* stream to listen to. Only after `WaitTwo` returns does it enter the **`case`** statement. Because of the

argument `F`, alternatives that do not apply are skipped. Note that the recursive calls *reverse* the two stream arguments. This helps guarantee fairness between both streams in systems where the `WaitTwo` statement favors one or the other (which is often the case in an implementation). A message appearing on an input stream will eventually appear on the output stream, independent of what happens in the other input stream.

Is it practical?

What can we say about practical programming in this model? Assume that new clients arrive during execution. Each client wants to communicate with the server. This means that a new stream merger must be created for each client! The final result is a tree of stream mergers feeding the server. Is this a practical solution? It has two problems:

- It is inefficient. Each stream merger executes in its own thread. The tree of stream mergers is extended at run time each time a new object references the server. Furthermore, the tree is not necessarily balanced. It would take extra work to balance it.

- It lacks expressiveness. It is not possible to reference the server directly. For example, it is not possible to put a server reference in a data structure. The only way we have to reference the server is by referencing one of its streams. We can put this in a data structure, but only *one* client can use this reference. (Remember that declarative data structures cannot be modified.)

How can we solve these two problems? The first problem could hypothetically be solved by a very smart compiler that recognizes the tree of stream mergers and replaces it by a direct many-to-one communication in the implementation. However, after two decades of research in this area, such a compiler does not exist [190]. Some systems solve the problem in another way: by adding an abstraction for multi-way merge whose implementation is done outside the model. This amounts to extending the model with ports. The second problem can be partially solved (see Exercises), but the solution is still cumbersome.

We seem to have found an inherent limitation of the nondeterministic concurrent model. Upon closer examination, the problem seems to be that there is no notion of explicit state in the model, where explicit state associates a name with a store reference. Both the name and the store reference are immutable; only their association can be changed. There are many equivalent ways to introduce explicit state. One way is by adding the concept of *cell*, as will be shown in Chapter 6. Another way is by adding the concept of *port*, as we did in this chapter. Ports and cells are equivalent in a concurrent language: there are simple implementations of each in terms of the other.

```
fun {WaitTwo A B}
X in
   thread {Wait A} try X=1 catch _ then skip end end
   thread {Wait B} try X=2 catch _ then skip end end
   X
end
```

Figure 5.25: Symmetric nondeterministic choice (using exceptions)

```
fun {WaitTwo A B}
U in
   thread {Wait A} U=unit end
   thread {Wait B} U=unit end
   {Wait U}
   if {IsDet A} then 1 else 2 end
end
```

Figure 5.26: Asymmetric nondeterministic choice (using `IsDet`)

Implementing nondeterministic choice

The `WaitTwo` operation can be defined in the declarative concurrent model if exceptions are added.[8] Figure 5.25 gives a simple definition. This returns 1 or 2, depending on whether `A` is bound or `B` is bound. This definition is symmetric; it does not favor either `A` or `B`. We can write an asymmetric version that favors `A` by using `IsDet`, as shown in Figure 5.26.[9]

5.8 Exercises

1. **Port objects that share one thread**. Section 5.5.1 gives a small program, 'Ping-Pong', that has two port objects. Each object executes a method and then asynchronously calls the other. When one initial message is inserted into the system, this causes an infinite ping-pong of messages to bounce between the objects. What happens if *two* (or more) initial messages are inserted? For example, what happens if these two initial calls are done:

   ```
   {Call Ping ping(0)}
   {Call Pong pong(10000000)}
   ```

[8] For practical use, however, we recommend the definition given in the supplements file on the book's Web site.

[9] Both definitions have the minor flaw that they can leave threads "hanging around" forever if one variable is never bound. The definitions can be corrected to terminate any hanging threads. We leave these corrections as exercises for the reader.