

```
fun lazy {Three} {Delay 1000} 3 end
```

Calculating {Three}+0 returns 3 after a 1000 millisecond delay. This is as expected, since the addition needs the result of {Three}. Now calculate {Three}+0 three times in succession. *Each* calculation waits 1000 milliseconds. How can this be, since Three is supposed to be lazy. Shouldn't its result be calculated only once?

11. **Laziness and concurrency I.** This exercise looks closer at the concurrent behavior of lazy execution. Execute the following:

```
fun lazy {MakeX} {Browse x} {Delay 3000} 1 end
fun lazy {MakeY} {Browse y} {Delay 6000} 2 end
fun lazy {MakeZ} {Browse z} {Delay 9000} 3 end
```

```
X={MakeX}
Y={MakeY}
Z={MakeZ}
```

```
{Browse (X+Y)+Z}
```

This displays `x` and `y` immediately, `z` after 6 seconds, and the result 6 after 15 seconds. Explain this behavior. What happens if  $(x+y)+z$  is replaced by  $x+(y+z)$  or by `thread x+y end + z`? Which form gives the final result the quickest? How would you program the addition of  $n$  integers  $i_1, \dots, i_n$ , given that integer  $i_j$  only appears after  $t_j$  milliseconds, so that the final result appears the quickest?

12. **Laziness and concurrency II.** Let us compare the kind of incrementality we get from laziness and from concurrency. Section 4.3.1 gives a producer/consumer example using concurrency. Section 4.5.3 gives the same producer/consumer example using laziness. In both cases, it is possible for the output stream to appear incrementally. What is the difference? What happens if you use both concurrency and laziness in the producer/consumer example?

13. **Laziness and monolithic functions.** Consider the following two definitions of lazy list reversal:

```
fun lazy {Reverse1 S}
  fun {Rev S R}
    case S of nil then R
    [] X|S2 then {Rev S2 X|R} end
  end
in {Rev S nil} end

fun lazy {Reverse2 S}
  fun lazy {Rev S R}
```

```

      case S of nil then R
      [] X|S2 then {Rev S2 X|R} end
    end
  in {Rev S nil} end

```

What is the difference in behavior between `{Reverse1 [a b c]}` and `{Reverse2 [a b c]}`? Do the two definitions calculate the same result? Do they have the same lazy behavior? Explain your answer in each case. Finally, compare the execution efficiency of the two definitions. Which definition would you use in a lazy program?

14. **Laziness and iterative computation.** In the declarative model, one advantage of dataflow variables is that the straightforward definition of `Append` is iterative. For this exercise, consider the straightforward lazy version of `Append` without dataflow variables, as defined in Section 4.5.7. Is it iterative? Why or why not?
15. **Performance of laziness.** For this exercise, take some declarative programs you have written and make them lazy by declaring all routines as lazy. Use lazy versions of all built-in operations, for example addition becomes `Add`, which is defined as `fun lazy {Add X Y} X+Y end`. Compare the behavior of the original eager programs with the new lazy ones. What is the difference in efficiency? Some functional languages, such as Haskell and Miranda, implicitly consider all functions as lazy. To achieve reasonable performance, these languages do *strictness analysis*, which tries to find as many functions as possible that can safely be compiled as eager functions.
16. **By-need execution.** Define an operation that requests the calculation of `x` but that does not wait.
17. **Hamming problem.** The Hamming problem of Section 4.5.6 is actually a special case of the original problem, which asks for the first  $n$  integers of the form  $p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$  with  $a_1, a_2, \dots, a_k \geq 0$  using the first  $k$  primes  $p_1, \dots, p_k$ . For this exercise, write a program that solves this problem for any  $n$  when given  $k$ .
18. **Concurrency and exceptions.** Consider the following control abstraction that implements `try-finally`:

```

proc {TryFinally S1 S2}
  B Y in
    try {S1} B=false catch X then B=true Y=X end
    {S2}
    if B then raise Y end end
  end
end

```

Using the abstract machine semantics as a guide, determine the different possible results of the following program:

```

local U=1 V=2 in
  {TryFinally
    proc {$}
      thread
        {TryFinally proc {$} U=V end
          proc {$} {Browse bing} end}
      end
    end
    proc {$} {Browse bong} end}
  end

```

How many different results are possible? How many different executions are possible?

19. **Limitations of declarative concurrency.** Section 4.7 states that declarative concurrency cannot model client/server applications, because the server cannot read commands from more than one client. Yet, the declarative `Merge` function of Section 4.5.6 reads from three input streams to generate one output stream. How can this be?
20. (*advanced exercise*) **Worst-case bounds with laziness.** Section 4.5.8 explains how to design a queue with worst-case time bound of  $O(\log n)$ . The logarithm appears because the variable `F` can have logarithmically many suspensions attached to it. Let us see how this happens. Consider an empty queue to which we repeatedly add new elements. The tuple  $(|F|, |R|)$  starts out as  $(0, 0)$ . It then becomes  $(0, 1)$ , which immediately initiates a lazy computation that will eventually make it become  $(1, 0)$ . (Note that `F` remains unbound and has one suspension attached.) When two more elements are added, the tuple becomes  $(1, 2)$ , and a second lazy computation is initiated that will eventually make it become  $(3, 0)$ . Each time that `R` is reversed and appended to `F`, one new suspension is created on `F`. The size of `R` that triggers the lazy computation doubles with each iteration. The doubling is what causes the logarithmic bound. For this exercise, let us investigate how to write a queue with a constant worst-case time bound. One approach that works is to use the idea of *schedule*, as defined in [138].
21. (*advanced exercise*) **List comprehensions.** Define a linguistic abstraction for list comprehensions (both lazy and eager) and add it to the Mozart system. Use the `gump` parser-generator tool documented in [104].
22. (*research project*) **Controlling concurrency.** The declarative concurrent model gives three primitive operations that affect execution order without changing the results of a computation: sequential composition (total order, supply-driven), lazy execution (total order, demand-driven), and concurrency (partial order, determined by data dependencies). These operations can be used to “tune” the order in which a program accepts input and

gives results, for example to be more or less incremental. This is a good example of separation of concerns. For this exercise, investigate this topic further and answer the following questions. Are these three operations complete? That is, can *all* possible partial execution orders be specified with them? What is the relationship with reduction strategies in the  $\lambda$  calculus (e.g., applicative order reduction, normal order reduction)? Are dataflow or single-assignment variables essential?

23. (*research project*) **Parallel implementation of functional languages.** Section 4.9.2 explains that non-strict evaluation allows to take advantage of speculative execution when implementing a parallel functional language. However, using non-strict evaluation makes it difficult to use explicit state. For this exercise, study this trade-off. Can a parallel functional language take advantage of both speculative execution and explicit state? Design, implement, and evaluate a language to verify your ideas.



# Chapter 5

## Message-Passing Concurrency

“Only then did Atreyu notice that the monster was not a single, solid body, but was made up of innumerable small steel-blue insects which buzzed like angry hornets. It was their compact swarm that kept taking different shapes.”

– The Neverending Story, *Michael Ende* (1929–1995)

In the last chapter we saw how to program with stream objects, which is both declarative and concurrent. But it has the limitation that it cannot handle observable nondeterminism. For example, we wrote a digital logic simulator in which each stream object knows exactly which object will send it the next message. We cannot program a client/server where the server does not know which client will send it the next message.

We can remove this limitation by extending the model with an asynchronous communication channel. Then any client can send messages to the channel and the server can read them from the channel. We use a simple kind of channel called a *port* that has an associated stream. Sending a message to the port causes the message to appear on the port’s stream.

The extended model is called the *message-passing concurrent model*. Since this model is nondeterministic, it is no longer declarative. A client/server program can give different results on different executions because the order of client sends is not determined.

A useful programming style for this model is to associate a port to each stream object. The object reads all its messages from the port, and sends messages to other stream objects through their ports. This style keeps most of the advantages of the declarative model. Each stream object is defined by a recursive procedure that is declarative.

Another programming style is to use the model directly, programming with ports, dataflow variables, threads, and procedures. This style can be useful for building concurrency abstractions, but it is not recommended for large programs because it is harder to reason about.

## Structure of the chapter

The chapter consists of the following parts:

- Section 5.1 defines the message-passing concurrent model. It defines the port concept and the kernel language. It also defines port objects, which combine ports with a thread.
- Section 5.2 introduces the concept of port objects, which we get by combining ports with stream objects.
- Section 5.3 shows how to do simple kinds of message protocols with port objects.
- Section 5.4 shows how to design programs with concurrent components. It uses port objects to build a lift control system.
- Section 5.5 shows how to use the message-passing model directly, without using the port object abstraction. This can be more complex than using port objects, but it is sometimes useful.
- Section 5.6 gives an introduction to Erlang, a programming language based on port objects. Erlang is designed for and used in telecommunications applications, where fine-grained concurrency and robustness are important.
- Section 5.7 explains one advanced topic: the nondeterministic concurrent model, which is intermediate in expressiveness between the declarative concurrent model and the message-passing model of this chapter.

## 5.1 The message-passing concurrent model

The message-passing concurrent model extends the declarative concurrent model by adding ports. Table 5.1 shows the kernel language. Ports are a kind of communication channel. Ports are no longer declarative since they allow observable nondeterminism: many threads can send a message on a port and their order is not determined. However, the part of the computation that does not use ports can still be declarative. This means that with care we can still use many of the reasoning techniques of the declarative concurrent model.

### 5.1.1 Ports

A *port* is an ADT that has two operations, namely creating a channel and sending to it:

- `{NewPort S P}`: create a new port with entry point `P` and stream `S`.

$\langle s \rangle ::=$	
<b>skip</b>	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
<b>local</b> $\langle x \rangle$ <b>in</b> $\langle s \rangle$ <b>end</b>	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
<b>if</b> $\langle x \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Conditional
<b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{pattern} \rangle$ <b>then</b> $\langle s \rangle_1$ <b>else</b> $\langle s \rangle_2$ <b>end</b>	Pattern matching
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
<b>thread</b> $\langle s \rangle$ <b>end</b>	Thread creation
$\{ \text{NewPort } \langle y \rangle \langle x \rangle \}$	<b>Port creation</b>
$\{ \text{Send } \langle x \rangle \langle y \rangle \}$	<b>Port send</b>

Table 5.1: The kernel language with message-passing concurrency

- $\{ \text{Send } P \ x \}$ : append  $x$  to the stream corresponding to the entry point  $P$ . Successive sends from the same thread appear on the stream in the same order in which they were executed. This property implies that a port is an asynchronous FIFO communication channel.

For example:

```
declare S P in
{NewPort S P}
{Browse S}
{Send P a}
{Send P b}
```

This displays the stream  $a|b|_-$ . Doing more sends will extend the stream. Say the current end of the stream is  $S$ . Doing the send  $\{ \text{Send } P \ a \}$  will bind  $S$  to  $a|S1$ , and  $S1$  becomes the new end of the stream. Internally, the port always remembers the current end of the stream. The end of the stream is a read-only variable. This means that a port is a secure ADT.

By *asynchronous* we mean that a thread that sends a message does not wait for any reply. As soon as the message is in the communication channel, the object can continue with other activities. This means that the communication channel can contain many pending messages, which are waiting to be handled. By *FIFO* we mean that messages sent from any one object arrive in the same order that they are sent. This is important for coordination among the threads.

### 5.1.2 Semantics of ports

The semantics of ports is quite straightforward. To define it, we first extend the execution state of the declarative model by adding a mutable store. Figure 5.1



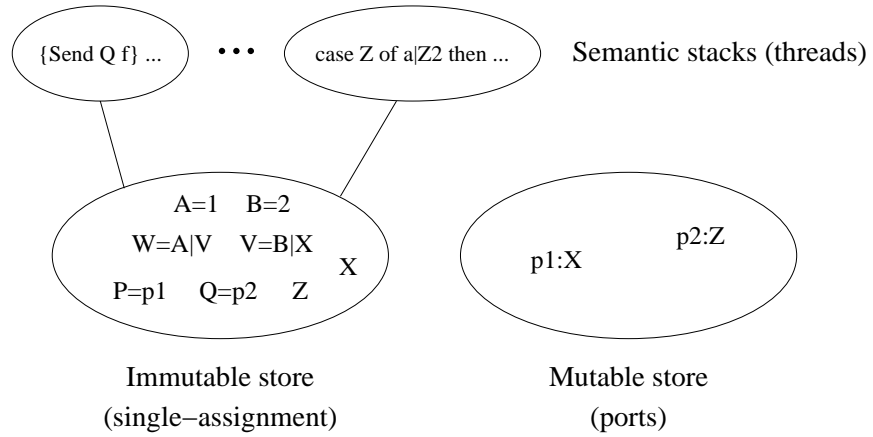


Figure 5.1: The message-passing concurrent model

shows the mutable store. Then we define the operations `NewPort` and `Send` in terms of the mutable store.

### Extension of execution state

Next to the single-assignment store  $\sigma$  (and the trigger store  $\tau$ , if laziness is important) we add a new store  $\mu$  called the *mutable store*. This store contains ports, which are pairs of the form  $x : y$ , where  $x$  and  $y$  are variables of the single-assignment store. The mutable store is initially empty. The semantics guarantees that  $x$  is always bound to a name value that represents a port and that  $y$  is unbound. We use name values to identify ports because name values are unique unforgeable constants. The execution state becomes a triple  $(MST, \sigma, \mu)$  (or a quadruple  $(MST, \sigma, \mu, \tau)$  if the trigger store is considered).

### The `NewPort` operation

The semantic statement  $(\{\text{NewPort } \langle x \rangle \langle y \rangle\}, E)$  does the following:

- Create a fresh port name  $n$ .
- Bind  $E(\langle y \rangle)$  and  $n$  in the store.
- If the binding is successful, then add the pair  $E(\langle y \rangle) : E(\langle x \rangle)$  to the mutable store  $\mu$ .
- If the binding fails, then raise an error condition.

### The `Send` operation

The semantic statement  $(\{\text{Send } \langle x \rangle \langle y \rangle\}, E)$  does the following:

- If the activation condition is true ( $E(\langle x \rangle)$  is determined), then do the following actions:
  - If  $E(\langle x \rangle)$  is not bound to the name of a port, then raise an error condition.
  - If the mutable store contains  $E(\langle x \rangle) : z$  then do the following actions:
    - \* Create a new variable  $z'$  in the store.
    - \* Update the mutable store to be  $E(\langle x \rangle) : z'$ .
    - \* Create a new list pair  $E(\langle y \rangle) | z'$  and bind  $z$  with it in the store.
- If the activation condition is false, then suspend execution.

This semantics is slightly simplified with respect to the complete port semantics. In a correct port, the end of the stream should always be a read-only view. This requires a straightforward extension to the `NewPort` and `Send` semantics. We leave this as an exercise for the reader.

### Memory management

Two modifications to memory management are needed because of the mutable store:

- Extending the definition of reachability: A variable  $y$  is reachable if the mutable store contains  $x : y$  and  $x$  is reachable.
- Reclaiming ports: If a variable  $x$  becomes unreachable, and the mutable store contains the pair  $x : y$ , then remove this pair.

## 5.2 Port objects

A port object is a combination of one or more ports and a stream object. This extends stream objects in two ways. First, many-to-one communication is possible: many threads can reference a given port object and send to it independently. This is not possible with a stream object because it has to know from where its next message will come from. Second, port objects can be embedded inside data structures (including messages). This is not possible with a stream object because it is referenced by a stream that can be extended by just one thread.

The concept of port object has many popular variations. Sometimes the word “agent” is used to cover a similar idea: an active entity with which one can exchange messages. The Erlang system has the “process” concept, which is like a port object except that it adds an attached mailbox that allows to filter incoming messages by pattern matching. Another often-used term is “active object”. It is similar to a port object except that it is defined in an object-oriented way, by a class (as we shall see in Chapter 7). In this chapter we use only port objects.

In the message-passing model, a program consists of a set of port objects sending and receiving messages. Port objects can create new port objects. Port objects can send messages containing references to other port objects. This means that the set of port objects forms a graph that can evolve during execution.

Port objects can be used to model distributed systems, where a *distributed system* is a set of computers that can communicate with each other through a network. Each computer is modeled as one or more port objects. A distributed algorithm is simply an algorithm between port objects.

A port object has the following structure:

```
declare P1 P2 ... Pn in
local S1 S2 ... Sn in
    {NewPort S1 P1}
    {NewPort S2 P2}
    ...
    {NewPort Sn Pn}
    thread {RP S1 S2 ... Sn} end
end
```

The thread contains a recursive procedure RP that reads the port streams and performs some action for each message received. Sending a message to the port object is just sending a message to one of the ports. Here is an example port object with one port that displays all the messages it receives:

```
declare P in
local S in
    {NewPort S P}
    thread {ForAll S proc {$ M} {Browse M} end} end
end
```

With the **for** loop syntax, this can be written more concisely as:

```
declare P in
local S in
    {NewPort S P}
    thread for M in S do {Browse M} end end
end
```

Doing {Send P hi} will eventually display hi. We can compare this with the stream objects of Chapter 4. The difference is that port objects allow *many-to-one* communication, i.e., any thread that references the port can send a message to the port object at any time. The object does not know from which thread the next message will come. This is in contrast to stream objects, where the object always knows from which thread the next message will come.

### 5.2.1 The NewPortObject abstraction

We can define an abstraction to make it easier to program with port objects. Let us define an abstraction in the case that the port object has just one port. To

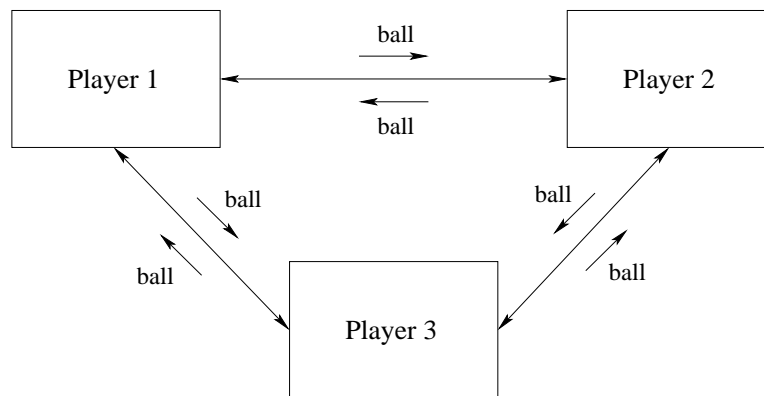


Figure 5.2: Three port objects playing ball

define the port object, we only have to give the initial state `Init` and the state transition function `Fun`. This function is of type  $\langle \text{fun } \{ \$ T_s T_m \} : T_s \rangle$  where  $T_s$  is the state type and  $T_m$  is the message type.

```

fun {NewPortObject Init Fun}
  proc {MsgLoop S1 State}
    case S1 of Msg|S2 then
      {MsgLoop S2 {Fun Msg State}}
    [] nil then skip end
  end
  Sin
in
  thread {MsgLoop Sin Init} end
  {NewPort Sin}
end

```

Some port objects are purely reactive, i.e., they have no internal state. The abstraction becomes simpler for them:

```

fun {NewPortObject2 Proc}
  Sin in
    thread for Msg in Sin do {Proc Msg} end end
  {NewPort Sin}
end

```

There is no state transition function, but simply a procedure that is invoked for each message.

### 5.2.2 An example

There are three players standing in a circle, tossing a ball amongst themselves. When a player catches the ball, he picks one of the other two randomly to throw the ball to. We can model this situation with port objects. Consider three port objects, where each object has a reference to the others. There is a ball that is sent

between the objects. When a port object receives the ball, it immediately sends it to another, picked at random. Figure 5.2 shows the three objects and what messages each object can send and where. Such a diagram is called a *component diagram*. To program this, we first define a component that creates a new player:

```

fun {Player Others}
  {NewPortObject2
    proc {$ Msg}
      case Msg of ball then
        Ran={OS.rand} mod {Width Others} + 1
      in
        {Send Others.Ran ball}
      end
    end}
  end

```

Others is a tuple that contains the other players. Now we can set up the game:

```

P1={Player others(P2 P3)}
P2={Player others(P1 P3)}
P3={Player others(P1 P2)}

```

In this program, Player is a component and P1, P2, P3 are its instances. To start the game, we toss a ball to one of the players:

```

{Send P1 ball}

```

This starts a furiously fast game of tossing the ball. To slow it down, we can add a {Delay 1000} in each player.

### 5.2.3 Reasoning with port objects

Consider a program that consists of port objects which send each other messages. Proving that the program is correct consists of two parts: proving that each port object is correct (when considered by itself) and proving that the port objects work together correctly. The first step is to show that each port object is correct. Each port object defines an ADT. The ADT should have an invariant assertion, i.e., an assertion that is true whenever an ADT operation has completed and before the next operation has started. To show that the ADT is correct, it is enough to show that the assertion is an invariant. We showed how to do this for the declarative model in Chapter 3. Since the inside of a port object is declarative (it is a recursive function reading a stream), we can use the techniques we showed there.

Because the port object has just one thread, the ADT's operations are executed sequentially within it. This means we can use mathematical induction to show that the assertion is an invariant. We have to prove two things:

- When the port object is first created, the assertion is satisfied.

- If the assertion is satisfied before a message is handled, then the assertion is satisfied after the message is handled.

The existence of the invariant shows that the port object itself is correct. The next step is to show that the program using the port objects is correct. This requires a whole different set of techniques.

A program in the message-passing model is a set of port objects that send each other messages. To show that this is correct, we have to determine what the possible sequences of messages are that each port object can receive. To determine this, we start by classifying all the events in the system (there are three kinds: message sends, message receives, and internal events of a port object). We can then define causality between events (whether an event happens before another). Considering the system of port objects as a state transition system, we can then reason about the whole program. Explaining this in detail is beyond the scope of this chapter. We refer interested readers to books on distributed algorithms, such as Lynch [115] or Tel [189].

## 5.3 Simple message protocols

Port objects work together by exchanging messages in coordinated ways. It is interesting to study what kinds of coordination are important. This leads us to define a *protocol* as a sequence of messages between two or more parties that can be understood at a higher level of abstraction than just its individual messages. Let us take a closer look at message protocols and see how to realize them with port objects.

Most well-known protocols are rather complicated ones such as the Internet protocols (TCP/IP, HTTP, FTP, etc.) or LAN (Local Area Network) protocols such as Ethernet, DHCP (Dynamic Host Connection Protocol), and so forth [107]. In this section we show some of simpler protocols and how to implement them using port objects. All the examples use `NewPortObject2` to create port objects.

Figure 5.3 shows the message diagrams of many of the simple protocols (we leave the other diagrams up to the reader!). These diagrams show the messages passed between a client (denoted `C`) and a server (denoted `S`). Time flows downwards. The figure is careful to distinguish idle threads (which are available to service requests) from suspended threads (which are not available).

### 5.3.1 RMI (Remote Method Invocation)

Perhaps the most popular of the simple protocols is the RMI. It allows an object to call another object in a different operating system process, either on the same machine or on another machine connected by a network [119]. Historically, the RMI is a descendant of the RPC (Remote Procedure Call), which was invented in the early 1980's, before object-oriented programming became popular [18]. The terminology RMI became popular once objects started replacing procedures as

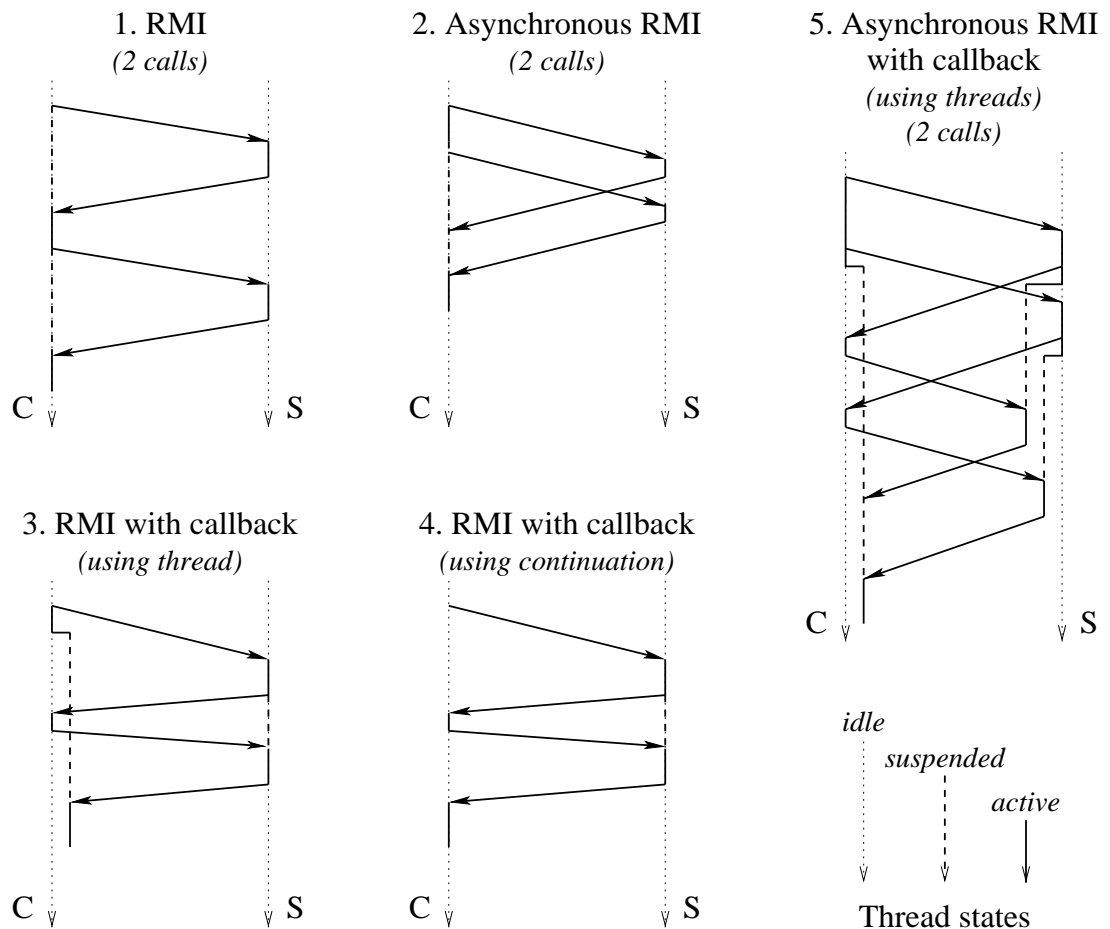


Figure 5.3: Message diagrams of simple protocols

the remote entities to be called. We apply the term RMI somewhat loosely to port objects, even though they do not have methods in the sense of object-oriented programming (see Chapter 7 for more on methods). For now, we assume that a “method” is simply what a port object does when it receives a particular message.

From the programmer’s viewpoint, the RMI and RPC protocols are quite simple: a client sends a request to a server and then waits for the server to send back a reply. (This viewpoint abstracts from implementation details such as how data structures are passed from one address space to another.) Let us give an example. We first define the server as a port object:

```

proc {ServerProc Msg}
  case Msg
  of calc(X Y) then
    Y=X*X+2.0*X+2.0
  end
end
Server={NewPortObject2 ServerProc}

```

This particular server has no internal state. The second argument `Y` of `calc` is bound by the server. We assume the server does a complex calculation, which we model by the polynomial  $x^2 + 2.0 \cdot x + 2.0$ . We define the client:

```

proc {ClientProc Msg}
  case Msg
  of work(Y) then
    Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Wait Y1}
      {Send Server calc(20.0 Y2)}
      {Wait Y2}
      Y=Y1+Y2
    end
  end
  Client={NewPortObject2 ClientProc}
  {Browse {Send Client work($)}}

```

Note that we are using a nesting marker “\$”. We recall that the last line is equivalent to:

```

local X in {Send Client work(X)} {Browse X} end

```

Nesting markers are a convenient way to turn statements into expressions. There is an interesting difference between the client and server definitions. The client definition references the server directly but the server definition does not know its clients. The server gets a client reference indirectly, through the argument `Y`. This is a dataflow variable that is bound to the answer by the server. The client waits until receiving the reply before continuing.

In this example, all messages are executed sequentially by the server. In our experience, this is the best way to implement RMI. It is simple to program with and reason about. Some RMI implementations do things somewhat differently. They allow multiple calls from different clients to be processed concurrently. This is done by allowing multiple threads at the server-side to accept requests for the same object. The server no longer serves requests sequentially. This is much harder to program with: it requires the server to protect its internal state data. We will examine this case later, in Chapter 8. When programming in a language that provides RMI or RPC, such as C or Java, it is important to know whether or not messages are executed sequentially by the server.

In this example, the client and server are both written in the same language and both execute in the same operating system process. This is true for all programs of this chapter. When the processes are not the same, we speak of a *distributed system*. This is explained in Chapter 11. This is possible, e.g., in Java RMI, where both processes run Java. The programming techniques of this chapter still hold for this case, with some modifications due to the nature of distributed systems.

It can happen that the client and server are written in different languages, but that we still want them to communicate. There exist standards for this, e.g.,



the CORBA architecture. This is useful for letting programs communicate even if their original design did not plan for it.

### 5.3.2 Asynchronous RMI

Another useful protocol is the asynchronous RMI. This is similar to RMI, except that the client continues execution immediately after sending the request. The client is informed when the reply arrives. With this protocol, two requests can be done in rapid succession. If communications between client and server are slow, then this will give a large performance advantage over RMI. In RMI, we can only send the second request after the first is completed, i.e., after one round trip from client to server.

```

proc {ClientProc Msg}
  case Msg
  of work(?Y) then
    Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Send Server calc(20.0 Y2)}
      Y=Y1+Y2
    end
  end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}

```

The message sends overlap. The client waits for both results Y1 and Y2 before doing the addition Y1+Y2.

Note that the server sees no difference with standard RMI. It still receives messages one by one and executes them sequentially. Requests are handled by the server in the same order as they are sent and the replies arrive in that order as well. We say that the requests and replies are sent in First-In-First-Out (FIFO) order.

### 5.3.3 RMI with callback (using thread)

The RMI with callback is like an RMI except that the server needs to call the client in order to fulfill the request. Let us see an example. Here is a server that does a callback to find the value of a special parameter called `delta`, which is known only by the client:

```

proc {ServerProc Msg}
  case Msg
  of calc(X ?Y Client) then
    X1 D in
      {Send Client delta(D)}
      X1=X+D
      Y=X1*X1+2.0*X1+2.0
    end
  end

```

```

    end
end
Server={NewPortObject2 ServerProc}

```

The server knows the client reference because it is an argument of the `calc` message. We leave out the `{Wait D}` since it is implicit in the addition `X+D`. Here is a client that calls the server in the same way as for RMI:

```

proc {ClientProc Msg}
  case Msg
  of work(?Z) then
    Y in
      {Send Server calc(10.0 Y Client)}
      Z=Y+100.0
    [] delta(?D) then
      D=1.0
    end
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}

```

(As before, the `wait` is implicit.) Unfortunately, this solution does not work. It deadlocks during the call `{Send Client work(Z)}`. Do you see why? Draw a message diagram to see why.<sup>1</sup> This shows that a simple RMI is not the right concept for doing callbacks.

The solution to this problem is for the client call not to wait for the reply. The client must continue immediately after making its call, so that it is ready to accept the callback. When the reply comes eventually, the client must handle it correctly. Here is one way to write a correct client:

```

proc {ClientProc Msg}
  case Msg
  of work(?Z) then
    Y in
      {Send Server calc(10.0 Y Client)}
      thread Z=Y+100.0 end
    [] delta(?D) then
      D=1.0
    end
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}

```

Instead of waiting for the server to bind `Y`, the client creates a new thread to do the waiting. The new thread's body is the work to do when `Y` is bound. When the reply comes eventually, the new thread does the work and binds `Z`.

---

<sup>1</sup>It is because the client suspends when it calls the server, so that the server cannot call the client.

It is interesting to see what happens when we call this client from outside. For example, let us do the call `{Send Client work(Z)}`. When this call returns, `Z` will usually not be bound yet. Usually this is not a problem, since the operation that uses `Z` will block until `Z` is bound. If this is undesirable, then the client call can itself be treated like an RMI:

```
{Send Client work(Z)}
{Wait Z}
```

This *lifts* the synchronization from the client to the application that uses the client. This is the right way to handle the problem. The problem with the original, buggy solution is that the synchronization is done in the wrong place.

### 5.3.4 RMI with callback (using record continuation)

The solution of the previous example creates a new thread for each client call. This assumes that threads are inexpensive. How do we solve the problem if we are not allowed to create a new thread? The solution is for the client to pass a *continuation* to the server. After the server is done, it passes the continuation back to the client so that the client can continue. In that way, the client never waits and deadlock is avoided. Here is the server definition:

```
proc {ServerProc Msg}
  case Msg
  of calc(X Client Cont) then
    X1 D Y in
      {Send Client delta(D)}
      X1=X+D
      Y=X1*X1+2.0*X1+2.0
      {Send Client Cont#Y}
    end
  end
end
Server={NewPortObject2 ServerProc}
```

After finishing its own work, the server passes `Cont#Y` back to the client. It adds `Y` to the continuation since `Y` is needed by the client!

```
proc {ClientProc Msg}
  case Msg
  of work(?Z) then
    {Send Server calc(10.0 Client cont(Z))}
    [] cont(Z)#Y then
      Z=Y+100.0
    [] delta(?D) then
      D=1.0
    end
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

The part of work after the server call is put into a new method, `cont`. The client passes the server the continuation `cont(Z)`. The server calculates `Y` and then lets the client continue its work by passing it `cont(Z)#Y`.

When the client is called from outside, the continuation-based solution to callbacks behaves in the same way as the thread-based solution. Namely, `z` will usually not be bound yet when the client call returns. We handle this in the same way as the thread-based solution, by lifting the synchronization from the client to its caller.

### 5.3.5 RMI with callback (using procedure continuation)

The previous example can be generalized in a powerful way by passing a procedure instead of a record. We change the client as follows (the server is unchanged):

```

proc {ClientProc Msg}
  case Msg
  of work(?Z) then
    C=proc{ $ Y } Z=Y+100.0 end
  in
    {Send Server calc(10.0 Client cont(C))}
  [] cont(C)#Y then
    {C Y}
  [] delta(?D) then
    D=1.0
  end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}

```

The continuation contains the work that the client has to do after the server call returns. Since the continuation is a procedure value, it is self-contained: it can be executed by anyone without knowing what is inside.

### 5.3.6 Error reporting

All the protocols we covered so far assume that the server will always do its job correctly. What should we do if this is not the case, that is, if the server can occasionally make an error? For example, it might be due to a network problem between the client and server, or the server process is no longer running. In any case, the client should be notified that an error has occurred. The natural way to notify the client is by raising an exception. Here is how we can modify the server to do this:

```

proc {ServerProc Msg}
  case Msg
  of sqrt(X Y E) then
    try

```

```

        Y={Sqrt X}
        E=normal
    catch Exc then
        E=exception(Exc)
    end
end
end
end
Server={NewPortObject2 ServerProc}

```

The extra argument *E* signals whether execution was normal or not. The server calculates square roots. If the argument is negative, *Sqrt* raises an exception, which is caught and passed to the client.

This server can be called by both synchronous and asynchronous protocols. In a synchronous protocol, the client can call it as follows:

```

{Send Server sqrt(X Y E)}
case E of exception(Exc) then raise Exc end end

```

The **case** statement blocks the client until *E* is bound. In this way, the client synchronizes on one of two things happening: a normal result or an exception. If an exception was raised at the server, then the exception is raised again at the client. This guarantees that *Y* is not used unless it is bound to a normal result. In an asynchronous protocol there is no guarantee. It is the client's responsibility to check *E* before using *Y*.

This example makes the basic assumption that the server can catch the exception and pass it back to the client. What happens when the server fails or the communication link between the client and server is cut or too slow for the client to wait? These cases will be handled in Chapter 11.

### 5.3.7 Asynchronous RMI with callback

Protocols can be combined to make more sophisticated ones. For example, we might want to do two asynchronous RMIs where each RMI does a callback. Here is the server:

```

proc {ServerProc Msg}
    case Msg
    of calc(X ?Y Client) then
        X1 D in
            {Send Client delta(D)}
        thread
            X1=X+D
            Y=X1*X1+2.0*X1+2.0
        end
    end
end
end

```

Here is the client:

```

proc {ClientProc Msg}
  case Msg
  of work(?Y) then
    Y1 Y2 in
      {Send Server calc(10.0 Y1 Client)}
      {Send Server calc(20.0 Y2 Client)}
      thread Y=Y1+Y2 end
    [] delta(?D) then
      D=1.0
    end
  end

```

What is the message diagram for the call {Send Client work(Y)}? What would happen if the server did not create a thread for doing the work after the callback?

### 5.3.8 Double callbacks

Sometimes the server does a first callback to the client, which itself does a second callback to the server. To handle this, both the client and the server must continue immediately and not wait until the result comes back. Here is the server:

```

proc {ServerProc Msg}
  case Msg
  of calc(X ?Y Client) then
    X1 D in
      {Send Client delta(D)}
      thread
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
      end
    [] serverdelta(?S) then
      S=0.01
    end
  end

```

Here is the client:

```

proc {ClientProc Msg}
  case Msg
  of work(Z) then
    Y in
      {Send Server calc(10.0 Y Client)}
      thread Z=Y+100.0 end
    [] delta(?D) then S in
      {Send Server serverdelta(S)}
      thread D=1.0+S end
    end
  end

```

Calling `{Send Client work(Z)}` calls the server, which calls the client method `delta(D)`, which itself calls the server method `serverdelta(S)`. A question for an alert reader: why is the last statement `D=1.0+S` also put in a thread?<sup>2</sup>

## 5.4 Program design for concurrency

This section gives an introduction to component-based programming with concurrent components.

In Section 4.3.5 we saw how to do digital logic design using the declarative concurrent model. We defined a logic gate as the basic circuit component and showed how to compose them to get bigger and bigger circuits. Each circuit had inputs and outputs, which were modeled as streams.

This section continues that discussion in a more general setting. We put it in the larger context of component-based programming. Because of message-passing concurrency we no longer have the limitations of the synchronous “lock-step” execution of Chapter 4.

We first introduce the basic concepts of concurrent modeling. Then we give a practical example, a lift control system. We show how to design and implement this system using high-level component diagrams and state diagrams. We start by explaining these concepts.

### 5.4.1 Programming with concurrent components

To design a concurrent application, the first step is to model it as a set of concurrent activities that interact in well-defined ways. Each concurrent activity is modeled by exactly one concurrent component. A concurrent component is sometimes known as an “agent”. Agents can be reactive (have no internal state) or have internal state. The science of programming with agents is sometimes known as *multi-agent systems*, often abbreviated as MAS. Many different protocols of varying complexities have been devised in MAS. This section only briefly touches on these protocols. In component-based programming, agents are usually considered as quite simple entities with little intelligence built-in. In the artificial intelligence community, agents are usually considered as doing some kind of reasoning.

Let us define a simple model for programming with concurrent components. The model has primitive components and ways to combine components. The primitive components are used to create port objects.

#### A concurrent component

Let us define a simple model for component-based programming that is based on port objects and executes with concurrent message-passing. In this model, a

---

<sup>2</sup>Strictly speaking, it is not needed in this example. But in general, the client does not know whether the server will do another callback!

*concurrent component* is a procedure with inputs and outputs. When invoked, the procedure creates a *component instance*, which is a port object. An input is a port whose stream is read by the component. An output is a port to which the component can send.

Procedures are the right concept to model concurrent components since they allow to compose components and to provide arbitrary numbers of inputs and outputs. Inputs and outputs can be local, with visibility restricted to inside the component.

### Interface

A concurrent component interacts with its environment through its interface. The interface consists of the set of its inputs and outputs, which are collectively known as its *wires*. A wire connects one or more outputs to one or more inputs. The message-passing model of this chapter provides two basic kinds of wires: one-shot and many-shot. One-shot wires are implemented by dataflow variables. They are used for values that do not change or for one-time messages (like acknowledgements). Only one message can be passed and only one output can be connected to a given input. Many-shot wires are implemented by ports. They are used for message streams. Any number of messages can be passed and any number of outputs can write to a given input.

The declarative concurrent model of Chapter 4 also has one-shot and many-shot wires, but the latter are restricted in that only one output can write to a given input.

### Basic operations

There are four basic operations in component-based programming:

- *Instantiation*: creating an instance of a component. By default, each instance is independent of each other instance. In some cases, instances might all have a dependency on a shared instance.
- *Composition*: build a new component out of other components. The latter can be called *subcomponents* to emphasize their relationship with the new component. We assume that the default is that the components we wish to compose have *no* dependencies. This means that they are concurrent! Perhaps surprisingly, compound components in a sequential system have dependencies even if they share no arguments. This follows because execution is sequential.
- *Linking*: combining component instances by connecting inputs and outputs. Different kinds of links: one-shot, many-shot, inputs that can be connected to one output only or to many outputs, outputs that can be connected to one input only or to many inputs. Usually, one-shot links go from one output to many inputs. All inputs see the same value when it is available.



Many-shot links go from many outputs to many inputs. All inputs see the same stream of input values.

- *Restriction*: restricting visibility of inputs or outputs to within a compound component. Restriction means to limit some of the interface wires of the subcomponents to the interior of the new component, i.e., they do not appear in the new component's interface.

Let us give an example to illustrate these concepts. In Section 4.3.5 we showed how to model digital logic circuits as components. We defined procedures `AndG`, `OrG`, `NotG`, and `DelayG` to implement logic gates. Executing one of these procedures creates a component instance. These instances are stream objects, but they could have been port objects. (A simple exercise is to generalize the logic gates to become port objects.) We defined a latch as a compound component as follows in terms of gates:

```

proc {Latch C DI ?DO}
    X Y Z F
in
    {DelayG DO F}
    {AndG F C X}
    {NotG C Z}
    {AndG Z DI Y}
    {OrG X Y DO}
end

```

The latch component has five subcomponents. These are linked together by connecting outputs and inputs. For example, the output `X` of the first `And` gate is given as input to the `Or` gate. Only the wires `DI` and `DO` are visible to the outside of the latch. The wires `X`, `Y`, `Z`, and `F` are restricted to the inside of the component.

### 5.4.2 Design methodology

Designing a concurrent program is more difficult than designing a sequential program, because there are usually many more potential interactions between the different parts. To have confidence that the concurrent program is correct, we need to follow a sequence of unambiguous design rules. From our experience, the design rules of this section give good results if they are followed with some rigor.

- *Informal specification*. Write down a possibly informal, but precise specification of what the system should do.
- *Components*. Enumerate all the different forms of concurrent activity in the specification. Each activity will become one component. Draw a block diagram of the system that shows all component instances.

- *Message protocols.* Decide on what messages the components will send and design the message protocols between them. Draw the component diagram with all message protocols.
- *State diagrams.* For each concurrent entity, write down its state diagram. For each state, verify that all the appropriate messages are received and sent with the right conditions and actions.
- *Implement and schedule.* Code the system in your favorite programming language. Decide on the scheduling algorithm to be used for implementing the concurrency between the components.
- *Test and iterate.* Test the system and reiterate until it satisfies the initial specification.

We will use these rules for designing the lift control system that is presented later on.

### 5.4.3 List operations as concurrency patterns

Programming with concurrent components results in many message protocols. Some simple protocols are illustrated in Section 5.3. Much more complicated protocols are possible. Because message-passing concurrency is so close to declarative concurrency, many of these can be programmed as simple list operations.

All the standard list operations (e.g., of the `List` module) can be interpreted as concurrency patterns. We will see that this is a powerful way to write concurrent programs. For example, the standard `Map` function can be used as a pattern that broadcasts queries and collects their replies in a list. Consider a list `PL` of ports, each of which is the input port of a port object. We would like to send the message `query(foo Ans)` to each port object, which will eventually bind `Ans` to the answer. By using `Map` we can send all the messages and collect the answers in a single line:

```
AL={Map PL fun {$ P} Ans in {Send P query(foo Ans)} Ans end}
```

The queries are sent asynchronously and the answers will eventually appear in the list `AL`. We can simplify the notation even more by using the `$` nesting marker with the `Send`. This completely avoids mentioning the variable `Ans`:

```
AL={Map PL fun {$ P} {Send P query(foo $)} end}
```

We can calculate with `AL` as if the answers are already there; the calculation will automatically wait if it needs an answer that is not there. For example, if the answers are positive integers, we can calculate their maximum by doing the same call as in a sequential program:

```
M={FoldL AL Max 0}
```

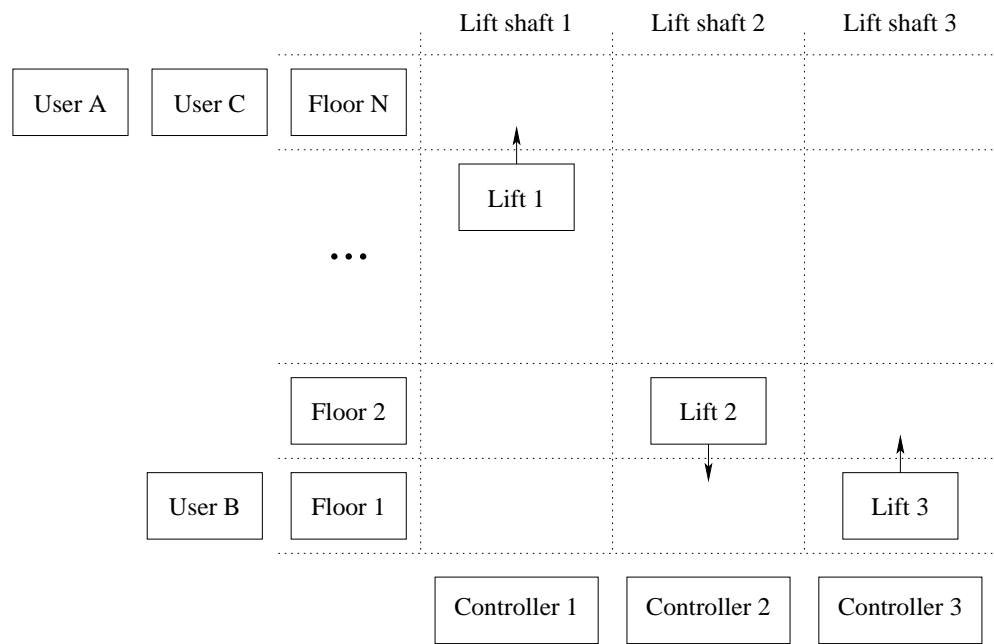


Figure 5.4: Schematic overview of a building with lifts

#### 5.4.4 Lift control system

Lifts are a part of our everyday life.<sup>3</sup> Yet, have you ever wondered how they work? How do lifts communicate with floors and how does a lift decide which floor to go to? There are many ways to program a lift control system.

In this section we will design a simple lift control system as a concurrent program. Our first design will be quite simple. Nevertheless, as you will see, the concurrent program that results will still be fairly complex. Therefore we take care to follow the design methodology given earlier.

We will model the operation of the hypothetical lift control system of a building, with a fixed number of lifts, a fixed number of floors between which lifts travel, and users. Figure 5.4 gives an abstract view of what our building looks like. There are floors, lifts, controllers for lift movement, and users that come and go. We will model what happens when a user calls a lift to go to another floor. Our model will focus on concurrency and timing, to show correctly how the concurrent activities interact in time. But we will put in enough detail to get a running program.

The first task is the specification. In this case, we will be satisfied with a partial specification of the problem. There are a set of floors and a set of lifts. Each floor has a call button that users can press. The call button does not specify an up or down direction. The floor randomly chooses the lift that will service its request. Each lift has a series of call(I) buttons numbered for all floors I, to tell

<sup>3</sup>Lifts are useful for those who live in flats, in the same way that elevators are useful for those who live in apartments.

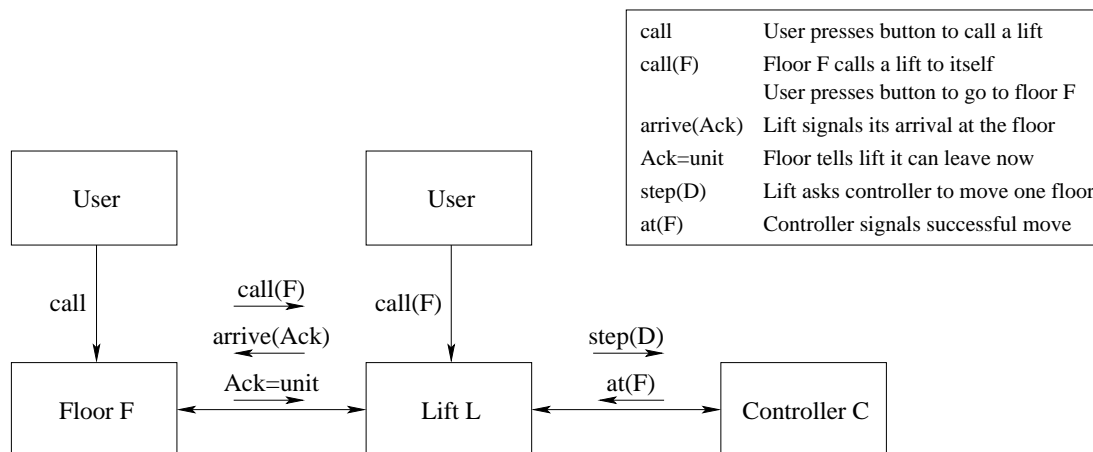


Figure 5.5: Component diagram of the lift control system

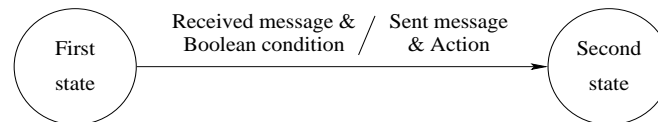


Figure 5.6: Notation for state diagrams

it to stop at a given floor. Each lift has a schedule, which is the list of floors that it will visit in order.

The scheduling algorithm we will use is called FCFS (First-Come-First-Served): a new floor is always added at the end of the schedule. This is also known as FIFO (First-In-First-Out) scheduling. Both the call and call(I) buttons do FCFS. When a lift arrives at a scheduled floor, the doors open and stay open for a fixed time before closing. Moving lifts take a fixed time to go from one floor to the next.

The lift control system is designed as a set of interacting concurrent components. Figure 5.5 shows the block diagram of their interactions. Each rectangle represents an instance of a concurrent component. In our design, there are four kinds of components, namely floors, lifts, controllers, and timers. All component instances are port objects. Controllers are used to handle lift motion. Timers handle the real-time aspect of the system.

Because of FCFS scheduling, lifts will often move much farther than necessary. If a lift is already at a floor, then calling that floor again may call another lift. If a lift is on its way from one floor to another, then calling an intermediate floor will not cause the lift to stop there. We can avoid these problems by making the scheduler more intelligent. Once we have determined the structure of the whole application, it will become clear how to do this and other improvements.

## State transition diagrams

A good way to design a port object is to start by enumerating the states it can be in and the messages it can send and receive. This makes it easy to check that all messages are properly handled in all states. We will go over the state diagrams of each component. First we introduce the notation for state transition diagrams (sometimes called state diagrams for short).

A state transition diagram is a finite state automaton. It consists of a finite set of states and a set of transitions between states. At each instant, it is in a particular state. It starts in an initial state and evolves by doing transitions. A *transition* is an atomic operation that does the following. The transition is enabled when the appropriate message is received and a boolean condition on it and the state is true. The transition can then send a message and change the state. Figure 5.6 shows the graphical notation. Each circle represents a state. Arrows between circles represent transitions.

Messages can be sent in two ways: to a port or by binding a dataflow variable. Messages can be received on the port's stream or by waiting for the binding. Dataflow variables are used as a lightweight channel on which only one message can be sent (a “one-shot wire”). To model time delays, we use a timer protocol: the caller `Pid` sends the message `starttimer(N Pid)` to a timer agent to request a delay of `N` milliseconds. The caller then continues immediately. When time is up, the timer agent sends a message `stoptimer` back to the caller. (The timer protocol is similar to the `{Delay N}` operation, reformulated in the style of concurrent components.)

## Implementation

We present the implementation of the lift control system by showing each part separately, namely the controller, the floor, and the lift. We will define functions to create them:

- `{Floor Num Init Lifts}` returns a floor `Fid` with number `Num`, initial state `Init`, and lifts `Lifts`.
- `{Lift Num Init Cid Floors}` returns a lift `Lid` with number `Num`, initial state `Init`, controller `Cid`, and floors `Floors`.
- `{Controller Init}` returns a controller `Cid`.

For each function, we explain how it works and give the state diagram and the source code. We then create a building with a lift control system and show how the components interact.

**The controller** The controller is the easiest to explain. It has two states, motor stopped and motor running. At the motor stopped state the controller can receive a `step(Dest)` from the lift, where `Dest` is the destination floor number.

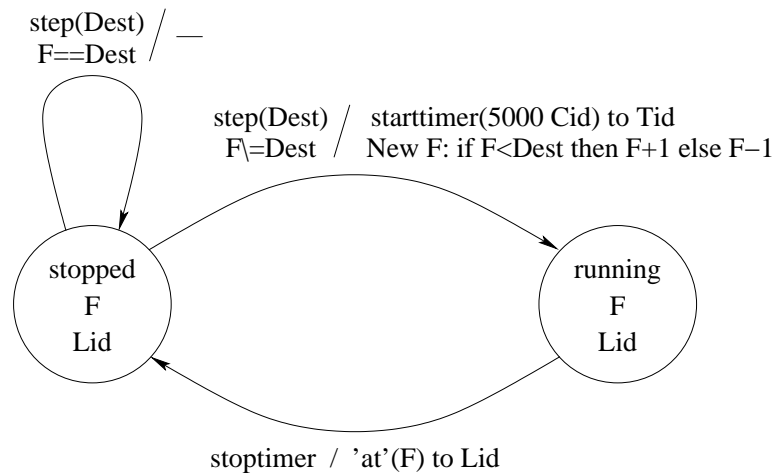


Figure 5.7: State diagram of a lift controller

The controller then goes to the motor running state. Depending on the direction, the controller moves up or down one floor. Using the timer protocol, the motor running state automatically makes a transition to the motor stopped state after a fixed time. This is the time needed to move from one floor to the next (either up or down). In the example, we assume this time to be 5000 ms. The timer protocol models a real implementation which would have a sensor at each floor. When the lift has arrived at floor **F**, the controller sends the message `'at'(F)` to the lift. Figure 5.7 gives the state diagram of controller **Cid**.

The source code of the timer and the controller is given in Figure 5.8. It is interesting to compare the controller code with the state diagram. The timer defined here is used also in the floor component.

Attentive readers will notice that the controller actually has more than two states, since strictly speaking the floor number is also part of the state. To keep the state diagram simple, we *parameterize* the motor stopped and motor running states by the floor number. Representing several states as one state with variables inside is a kind of syntactic sugar for state diagrams. It lets us represent very big diagrams in a compact way. We will use this technique also for the floor and lift state diagrams.

**The floor** Floors are more complicated because they can be in one of three states: no lift called, lift called but not yet arrived, and lift arrived and doors open. Figure 5.9 gives the state diagram of floor **Fid**. Each floor can receive a `call` message from a user, an `arrive(Ack)` message from a lift, and an internal timer message. The floor can send a `call(F)` message to a lift.

The source code of the floor is shown in Figure 5.10. It uses the random number function `OS.rand` to pick a lift at random. It uses `Browse` to display when a lift is called and when the doors open and close. The total time needed for opening and closing the doors is assumed to be 5000 ms.