Chapter 14

Concurrent Processes

Many hands make light work.

- John Heywood.

Thus far the systems that we have considered have been simple sequential processes, and have deviated from the standard (deterministic) notion of a sequential program only by the presence of (nondeterministic) choice. Of course the real interest in the study of systems arises when we permit processes to run in parallel and interact with one another. There are a variety of ways in which one might introduce operators into the language to permit such concurrent process behaviour. In this chapter we introduce a relatively simple operator, referred to as ***synchronisation merge***, and demonstrate its use in a variety of example applications.

14.1 Synchronisation Merge

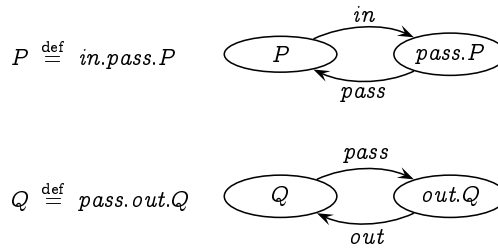
In this section, we introduce a parallel composition operator $\|$ which allows two processes E and F to execute in parallel. The precise fashion in which this concurrent execution takes place must be defined; in particular, we must clearly stipulate in what fashion such concurrent processes may interact with one another. To motivate our study, we start with a simple example.

Example 14.1

Suppose we have a very simple factory employing two workers. The first worker takes in jobs one at a time and, after carrying out some work on a job, passes it on to the second worker (assuming the second worker isn't still working on an earlier job). The second worker takes jobs one at a time from the first worker and, after carrying out some work on a job, sends it out of the factory.

The two workers can be represented by the following two simple processes P and Q:

F. Moller, G. Struth, *Modelling Computing Systems*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84800-322-4_15, © Springer-Verlag London 2013



When the two workers start their work day, the first can start working immediately by taking in the first job, represented by the transition

$$P \xrightarrow{in} pass.P$$

However, the second worker has to wait until the first worker has completed working on the first job and passes it on; that is, the transition

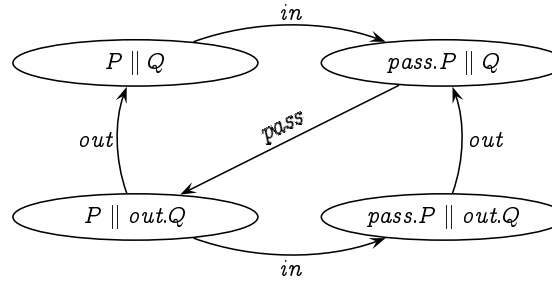
$$Q \xrightarrow{pass} out.Q$$

cannot take place in reality until the associated transition

$$pass.P \xrightarrow{pass} P$$

takes place. The two workers *synchronise* on the *pass* action; they must do this action together.

Consider what we would see if we were to watch these two workers. The behaviour of these two processes P and Q running together would be represented by the following process, where we represent the two relevant process states side-by-side separated by parallel lines $\|$:



Note that having passed a job on, the first worker can take in another job; however, this job cannot be passed on until the second worker has sent out the previous job.

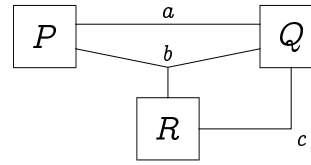
In the above example, two processes P and Q are made to run in parallel. This parallel composition is written as $P \parallel Q$, and each process is allowed

to perform certain of its actions independent of the other, but is forced to synchronise with the other process on certain other actions. With this understanding, we are now prepared to explain the formal definition of the parallel composition operator $\|$.

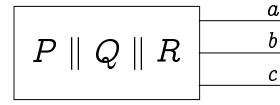
We first require each process state E to have a well-defined ***synchronisation sort*** $\mathsf{Sort}(E)$, denoting the subset of actions of the process on which it synchronises with other processes; every state of a given process will possess the same sort. The synchronisation sort of a process identifies those actions which are, in essence, external to the process, and represent those actions through which the process communicates with other processes via synchronisation. They are the actions by which processes are interconnected.

Example 14.2

Suppose $\mathsf{Sort}(P) = \{a, b\}$, $\mathsf{Sort}(Q) = \{a, b, c\}$ and $\mathsf{Sort}(R) = \{b, c\}$. Composing these processes in parallel gives a system $P \parallel Q \parallel R$ in which the individual components P, Q and R are directly connected through the actions of their respective synchronisation sorts. The composed system can thus be viewed schematically as follows:



This depicts the whole system as consisting of the three physical processes P, Q and R all operating independently. The behaviour of these three processes is not depicted in the diagram, but they are inter-connected through the three actions a, b and c, which may be thought of as physical ports. The result is a process $P \parallel Q \parallel R$, which can itself be composed in parallel with further processes, with the synchronisation sort $\mathsf{Sort}(P \parallel Q \parallel R) = \{a, b, c\}$:



The intention of the synchronisation sort of a process is to define which actions are of importance when it comes to interaction; the individual actions of E may take place in the composition $E \parallel F$ so long as they are not in the sort of the process F. However, E must synchronise with F on any action from the sort of F which E is prepared to do. That is, E cannot do an action $a \in \mathsf{Sort}(F)$ unless F itself is prepared to do this action, in which case E and F can perform this action in synchrony. Note that when we

compose two processes E and F, the sort of the resulting processes is the union of the sorts of the components:

$$\mathsf{Sort}(E \parallel F) \quad = \quad \mathsf{Sort}(E) \cup \mathsf{Sort}(F).$$

With this understanding in place, we may give the formal semantic definition of the ***synchronisation merge*** $E \parallel F$ of processes E and F. There are three rules governing the behaviour of $E \parallel F$:

1. one which stipulates that $E \parallel F$ may perform a transition of E as long as it does not involve an action from the synchronisation sort of F;
2. one which stipulates that $E \parallel F$ may perform a transition of F as long as it does not involve an action from the synchronisation sort of E; and
3. one which stipulates that $E \parallel F$ may synchronise on the performance, by E and F together, of an action in either (or both) of their synchronisation sorts.

Formally, these rules are as follows.

1. If $E \xrightarrow{a} E'$ and $a \notin \mathsf{Sort}(F)$ then $E \parallel F \xrightarrow{a} E' \parallel F$.
2. If $F \xrightarrow{a} F'$ and $a \notin \mathsf{Sort}(E)$ then $E \parallel F \xrightarrow{a} E \parallel F'$.
3. If $E \xrightarrow{a} E'$ and $F \xrightarrow{a} F'$ and $a \in \mathsf{Sort}(E) \cup \mathsf{Sort}(F)$ then $E \parallel F \xrightarrow{a} E' \parallel F'$.

One further point to make is that two equivalent processes must have the same synchronisation sort.

Exercise 14.2 (Solution on page 483)

Why is it important that equivalent processes have the same sort?

Hint: We wish to make sure that if $A \sim B$ then $A \parallel X \sim B \parallel X$, that is, there should be no effect in the functioning of a system if we replace one component A with an equivalent component B. What might happen, though, if A and B have the same behaviour but different synchronisation sorts?

14.2 Counters

For any integer $k>0$, a ***k-counter*** is a system which stores an integer value between 0 and k (inclusively). The k-counter can be:

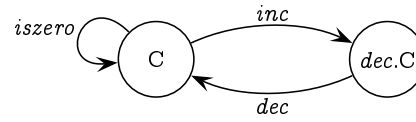
- *incremented*, as long as its value is less than k;

- *decremented*, as long as its value is greater than 0; and
- *tested* if its value is zero.

For example, we can define a 1-counter C by:

$$\mathrm{C} \stackrel{\text{def}}{=} \mathit{iszero}.\mathrm{C} + \mathit{inc}.\mathit{dec}.\mathrm{C}$$

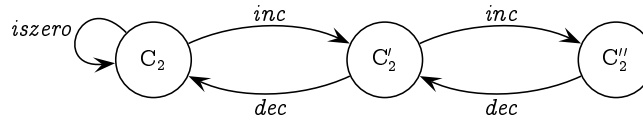
which defines the following transition system:



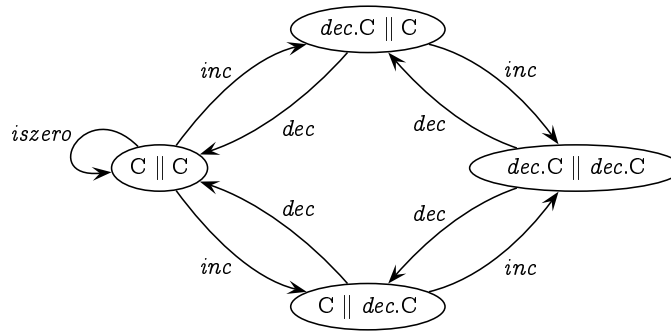
Almost as simply, we can define a 2-counter by:

$$\mathrm{C}_2 \stackrel{\text{def}}{=} \mathit{iszero}.\mathrm{C}_2 + \mathit{inc}.\mathrm{C}'_2$$
$$\mathrm{C}'_2 \stackrel{\text{def}}{=} \mathit{inc}.\mathrm{C}''_2 + \mathit{dec}.\mathrm{C}_2$$
$$\mathrm{C}''_2 \stackrel{\text{def}}{=} \mathit{dec}.\mathrm{C}'_2$$

which defines the following transition system:



We can use two copies of the simple 1-counter to "implement" a 2-counter. Assuming that $\mathsf{Sort}(\mathrm{C}) = \{\mathit{iszero}\}$, the transition system $\mathrm{C} \parallel \mathrm{C}$ is depicted as follows:



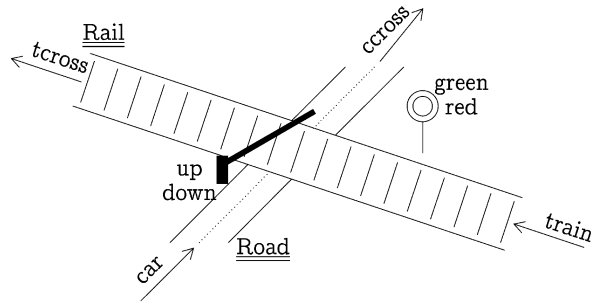


Figure 14.1: The railway level crossing.

Here, the initial state $C \parallel C$ can do an increment action *inc* in two ways, either by allowing the left-hand 1-counter C perform this action, or by allowing the right-hand 1-counter C perform it; as this action is not in the synchronisation sort of C, neither process will block the other from performing this action.

On the other hand, the *iszero* action is only possible in the initial state $C \parallel C$; as the action *iszero* is in the synchronisation sort of C, both components of the parallel composition must participate in this action.

Generalising this result, we can show that a k-counter, for any k, can be implemented by combining k copies of the simple 1-counter in parallel; that is,

$$C_k \sim \underbrace{C \parallel C \parallel \cdots \parallel C}_{k \text{ copies}}.$$

Exercise 14.3 (Solution on page 484)

Prove that $C_2 \sim C \parallel C$.

14.3 Railway Level Crossing

Consider the railway level crossing depicted in Figure 14.1. This system consists of three components working in parallel.

- A `Rail` process, which represents the arrival of trains, assuring that they only cross if the signal is green.

- A Road process, which represents the arrival of cars, assuring that they only cross if the barrier is up.
- A Controller process, which regulates the signal and barrier, assuring that the barrier is never up at the same time that the signal is green.

This is a typical example of a control system, in which a controller process is regulating the behaviour of other processes in order to prevent undesirable behaviours. The desirable properties which the controller would like to attain are of two kinds.

1. ***Safety Properties***: (*No crashes*)
   - A car may not cross at the same time as a train.
2. ***Liveness Properties***: (*Eventual service*)
   - If a car arrives, eventually the barrier goes up.
   - If a train arrives, eventually the signal turns green.

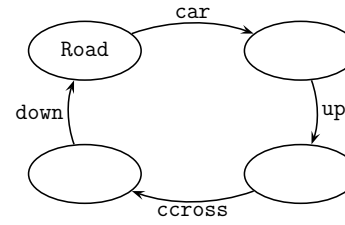
We shall now describe the behaviour of the three component processes.

1. $\texttt{Road} \stackrel{\text{def}}{=} \texttt{car.up.ccross.down.Road}$,
   with $\mathsf{Sort}(\texttt{Road}) = \{\texttt{up}, \texttt{down}\}$.

   The Road process repeatedly carries out the following events.

   (a) signals the arrival of a car at the crossing (the car action);
   (b) witnesses the raising of the barrier (the up action);
   (c) signals the crossing of the car (the ccross action); and finally
   (d) witnesses the lowering of the barrier (the down action).

   The Road process is thus depicted by the following transition system.



2. $\texttt{Rail} \stackrel{\text{def}}{=} \texttt{train.green.tcross.red.Rail}$,
   with $\mathsf{Sort}(\texttt{Rail}) = \{\texttt{green}, \texttt{red}\}$.

   Analogous to the Road process, the Rail process repeatedly carries out the following events.

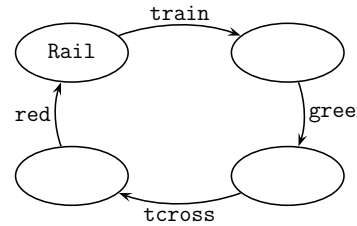
(a) signals the arrival of a train at the crossing (the `train` action);

(b) witnesses the signal turning green (the `green` action);

(c) signals the crossing of the train (the `tcross` action); and finally

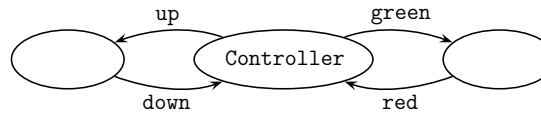
(d) witnesses the signal turning red (the `red` action).

The `Rail` process is thus depicted by the following transition system.



3. $\texttt{Controller} \stackrel{\text{def}}{=} \texttt{green.red.Controller} + \texttt{up.down.Controller}$, with $\mathsf{Sort}(\texttt{Controller}) = \{\texttt{up}, \texttt{down}, \texttt{green}, \texttt{red}\}$.

   The `Controller` process is thus depicted by the following transition system.

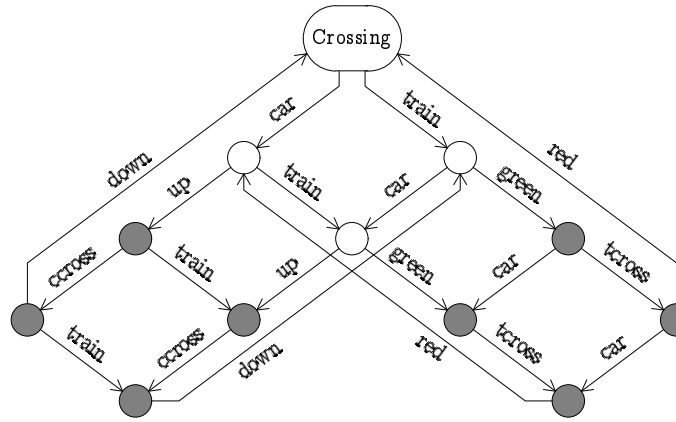


The complete railway level crossing system then consists of the above three components executing in parallel:

$$\texttt{Crossing} \stackrel{\text{def}}{=} \texttt{Road} \parallel \texttt{Controller} \parallel \texttt{Rail}$$

Its structure can be depicted as follows.

Its behaviour is thus given by the following transition system.



Exercise 14.4 (Solution on page 484)

Do the desired safety and liveness properties mentioned above hold? Explain why or why not. If any of these properties fail,

- can you propose a weaker yet acceptable property which does hold?
- can you propose a way to alter the definitions of the components of the system so that the property does hold?

14.4 Mutual Exclusion

When two tasks are being carried out together, problems can occur if they want to access some shared resource at the same time. A striking illustration of this is the Clayton Tunnel Accident (page 2), wherein one train was allowed to enter the tunnel which was currently occupied by another train. ***Mutual exclusion*** refers to the problem of ensuring that two processes can never be in their ***critical section*** – ie, using a shared resource such as a shared memory or printer – at the same time. If a process is granted use of such a shared resource, it must be allowed to maintain exclusive use of this resource until it has completed its use and released it. This is a ubiquitous problem in the design of concurrency systems.

14.4.1 Dining Philosophers

The problem of mutual exclusion was first identified and solved in 1965 by Edsger W. Dijkstra who also proposed the following illustration of how a system can deadlock if due concern is not taken to the use of shared

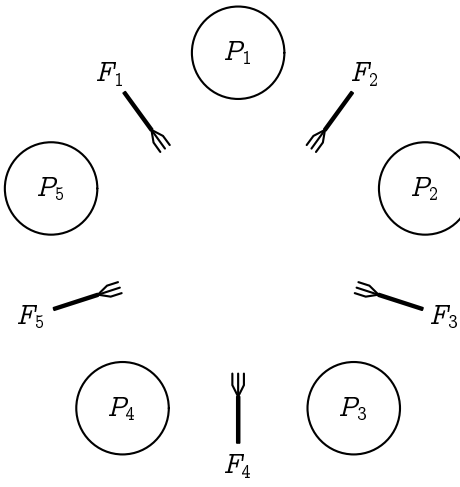


Figure 14.2: Dining Philosophers.

resources. It is a very simple problem to consider, yet offers a wealth of insight into the challenges posed by synchronisation.

Imagine there are five philosophers sitting at a round dining table thinking. Each philosopher has a plate of spaghetti in front of them, and there is a fork between every pair of plates. Figure 14.2 depicts the situation. The spaghetti is hopelessly tangled, meaning that a philosopher must use two forks together to eat it[†]. A philosopher may only use the two forks that are on either side of their own plate, which they pick up one at a time in either order. After taking a mouthful of spaghetti, the philosopher then replaces the two forks, in either order, to where they were lifted from the table. As the philosophers are deep in their own thoughts, at no point do they communicate with one another.

Our task is to design a protocol – that is, model the interactions between the philosophers and the forks – which satisfies the following correctness properties:

1. A philosopher eats only when holding two forks.

[†]The story is often described with rice rather than spaghetti, and chopsticks in place of forks, making it more immediate that two utensils are needed to eat.

2. No two philosophers may hold the same fork simultaneously (mutual exclusion).
3. The philosophers never get stuck, with every one of them forever waiting for some fork to become available (deadlock freedom).

To model this problem, we introduce the following actions:

- eat_i: philosopher i takes a bite to eat.
- $lift_{ij}$: philosopher i picks up fork j.
- $drop_{ij}$: philosopher i puts down fork j.

The behaviour of the forks is easy to describe:

$$
\begin{aligned}
F_1 &\stackrel{\text{def}}{=} lift_{51} \,.\, drop_{51} \,.\, F_1 + lift_{11} \,.\, drop_{11} \,.\, F_1 \\
F_2 &\stackrel{\text{def}}{=} lift_{12} \,.\, drop_{12} \,.\, F_2 + lift_{22} \,.\, drop_{22} \,.\, F_2 \\
F_3 &\stackrel{\text{def}}{=} lift_{23} \,.\, drop_{23} \,.\, F_3 + lift_{33} \,.\, drop_{33} \,.\, F_3 \\
F_4 &\stackrel{\text{def}}{=} lift_{34} \,.\, drop_{34} \,.\, F_4 + lift_{44} \,.\, drop_{44} \,.\, F_4 \\
F_5 &\stackrel{\text{def}}{=} lift_{45} \,.\, drop_{45} \,.\, F_5 + lift_{55} \,.\, drop_{55} \,.\, F_5
\end{aligned}
$$

That is to say, a fork is picked up by one of the two philosophers nearest to it at the table, and is subsequently placed back on the table by that same philosopher.

We have some freedom in how to define the behaviour of a philosopher, in that it is unspecified which order they pick up and set down their forks. As a first attempt, we assume that they each pick up the fork to their right first (as well as set this one down first):

$$
\begin{aligned}
P_1 &\stackrel{\text{def}}{=} lift_{11} \,.\, lift_{12} \,.\, eat_1 \,.\, drop_{11} \,.\, drop_{12} \,.\, P_1 \\
P_2 &\stackrel{\text{def}}{=} lift_{22} \,.\, lift_{23} \,.\, eat_2 \,.\, drop_{22} \,.\, drop_{23} \,.\, P_2 \\
P_3 &\stackrel{\text{def}}{=} lift_{33} \,.\, lift_{34} \,.\, eat_3 \,.\, drop_{33} \,.\, drop_{34} \,.\, P_3 \\
P_4 &\stackrel{\text{def}}{=} lift_{44} \,.\, lift_{45} \,.\, eat_4 \,.\, drop_{44} \,.\, drop_{45} \,.\, P_4 \\
P_5 &\stackrel{\text{def}}{=} lift_{55} \,.\, lift_{51} \,.\, eat_5 \,.\, drop_{55} \,.\, drop_{51} \,.\, P_5
\end{aligned}
$$

The synchronisation sorts for forks and philosopher processes are defined naturally as follows:

$$
\begin{aligned}
\mathsf{Sort}(P_i) &= \{\, lift_{ij} \,,\, drop_{ij} \;:\; 1 \leq j \leq 5 \,\}. \\
\mathsf{Sort}(F_j) &= \{\, lift_{ij} \,,\, drop_{ij} \;:\; 1 \leq i \leq 5 \,\}.
\end{aligned}
$$

Unfortunately this protocol has the possibility of deadlocking: every philosopher may pick up a fork with their right hand before any one of them picks up the fork to their left, at which time they will all be waiting forever for their left-hand neighbour to return the fork to the table.

We can resolve this problem by changing the definition of the first (and only the first) philosopher, who is required to pick up the fork to their *left* first:

$$P_1 \stackrel{\text{def}}{=} lift_{12} \,.\, lift_{11} \,.\, eat_1 \,.\, drop_{12} \,.\, drop_{11} \,.\, P_1$$

With some thought, it becomes apparent that this refined protocol cannot deadlock.

Exercise 14.5 (Solution on page 484)

Argue that the refined protocol, in which the first philosopher picks up the left fork first, cannot deadlock.

14.4.2 Peterson's Algorithm

There have been various solutions proposed for dealing with the mutual exclusion problem. Here we examine an elegant solution proposed by Gary L. Peterson in 1981.

We consider two processes, P_1 and P_2, both of which wanting at times to enter some critical section. There are two Boolean variables: `b1` which is true if P_1 wants to enter (or is in) the critical section, and `b2` which is true if P_2 wants to enter (or is in) the critical section; and a variable `k` which has value 1 or 2 indicating which process has "ownership" of (ie, the authority to enter) the critical section. The Boolean variables `b1` and `b2` are initially set to false, while the initial value of `k` is arbitrary.

The two processes are then defined as follows (where the actual details of the critical and noncritical sections are left unspecified).

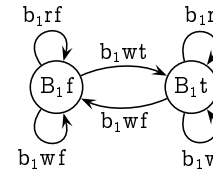
P_1: `while true do`	P_2: `while true do`
$\cdots$ *noncritical section* $\cdots$	$\cdots$ *noncritical section* $\cdots$
`b1 := true;`	`b2 := true;`
`k := 2;`	`k := 1;`
`while (b2 and k=2) do`	`while (b1 and k=1) do`
`skip;`	`skip;`
$\cdots$ *critical section* $\cdots$	$\cdots$ *critical section* $\cdots$
`b1 := false`	`b2 := false`

When process P_1 wishes to enter the critical section, it indicates this by setting `b1` to true, but also sets `k` to 2 granting authority to the other process P_2 to enter the critical section. It then waits until either the other process P_2 does not wish to enter the critical region (ie, `b2` is false) or the other process grants it authority to enter the critical region (ie, `k` has value 1), at which time it enters the critical region; when it exits the critical region it indicates this by setting `b1` to false. Process P_2 is defined in an identical fashion.

To model these processes as labelled transition systems, we first need to represent the variables `b1`, `b2` and `k` themselves as processes which interact

with the processes P_1 and P_2. The variable b1 is represented by the following two-state system:

$$B_1f \stackrel{\text{def}}{=} b_1rf.B_1f + b_1wf.B_1f + b_1wt.B_1t$$
$$B_1t \stackrel{\text{def}}{=} b_1rt.B_1t + b_1wt.B_1t + b_1wf.B_1f$$
$$\mathsf{Sort}(B_1f) = \{b_1rf, b_1rt, b_1wf, b_1wt\}$$



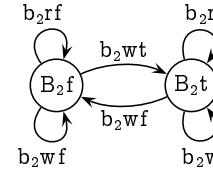
The state B_1f ("f" for "false") signifies that the variable b1 has the value false, while the state B_1t ("t" for "true") signifies that the variable b1 has the value true.

- Processes read the value of the variable by synchronising with the process on the actions b_1rf and b_1rt ("r" for "read"): the action b_1rf represents the process telling the environment that the value of b1 is false, while the action b_1rt represents the process telling the environment that the value of b1 is true. The state of the variable process does not change on these actions.
- Processes write a value to the variable by synchronising with the process on the actions b_1wf and b_1wt ("w" for "write"): the action b_1wf ("w" for "write") represents the value of the variable b1 being set to false, while the action b_1wt represents the value of the variable b1 being set to true. The state of the variable process changes as appropriate on these actions.

All of the reading and writing actions are included in the sort of the process, as clearly these actions must be done in synchrony with this process.

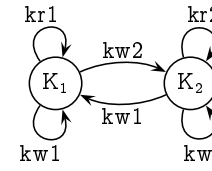
The variable b2 has an analogous definition:

$$B_2f \stackrel{\text{def}}{=} b_2rf.B_2f + b_2wf.B_2f + b_2wt.B_2t$$
$$B_2t \stackrel{\text{def}}{=} b_2rt.B_2t + b_2wt.B_2t + b_2wf.B_2f$$
$$\mathsf{Sort}(B_2f) = \{b_2rf, b_2rt, b_2wf, b_2wt\}$$



Finally, the variable k is similarly defined:

$$K_1 \stackrel{\text{def}}{=} kr1.K_1 + kw1.K_1 + kw2.K_2$$
$$K_2 \stackrel{\text{def}}{=} kr2.K_2 + kw2.K_2 + kw1.K_1$$
$$\mathsf{Sort}(K_1) = \{kr1, kr2, kw1, kw2\}$$



Again, the variable k is represented by a two-state process, representing the two values that the variable k can hold (either 1 or 2); and there are actions representing the reading and writing of these two values.

We now turn our attention to defining the processes P_1 and P_2. As the behaviour of the processes within the noncritical and critical sections are irrelevant for our study – we are only interested in ensuring that mutual exclusion is attained – we ignore these completely. The behaviour of the process P_1 thus starts with setting the value of b1 to true and the value of k to 2:

$$P_1 \stackrel{\text{def}}{=} \mathrm{b}_1\mathrm{wt} \,.\, \mathrm{kw2}.W_1$$

The process W_1 represents the process at the point of executing the while loop waiting to enter the critical section:

```
while (b2 and k=2) do skip
```

The process will stay in the state W_1 for as long as the value of b2 is true (ie, the action $\mathrm{b}_2\mathrm{rt}$ can occur) and the value of k is 2 (ie, the action kr2 can occur). However, if either of these is false, that is if the value of b2 is false (ie, the action $\mathrm{b}_2\mathrm{rf}$ can occur) or the value of k is 1 (ie, the action kr1 can occur), then the process will move into a new state R_1 signifying that the process is ready to enter the critical section:

$$W_1 \stackrel{\text{def}}{=} \mathrm{b}_2\mathrm{rt} \,.\, W_1 \;+\; \mathrm{kr2} \,.\, W_1 \;+\; \mathrm{b}_2\mathrm{rf} \,.\, R_1 \;+\; \mathrm{kr1} \,.\, R_1$$

Finally, in the state R_1 the process will enter the critical section, then (ultimately) exit it, and set the value of the variable b1 to be false, before returning then to the initial state:

$$R_1 \stackrel{\text{def}}{=} \mathit{enter} \,.\, \mathit{exit} \,.\, \mathrm{b}_1\mathrm{wf} \,.\, P_1$$

The synchronisation sort of the process P_1 contains the three relevant writing events:

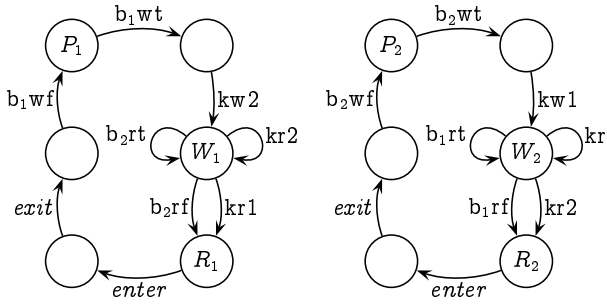
$$\mathsf{Sort}(P_1) = \{\, \mathrm{b}_1\mathrm{wt},\ \mathrm{b}_1\mathrm{wf},\ \mathrm{kw2} \,\},$$

as the variables can only change value if they are written to.

The process P_2 is defined analogously to process P_1:

$$\begin{aligned} P_2 &\stackrel{\text{def}}{=} \mathrm{b}_2\mathrm{wt} \,.\, \mathrm{kw1} \,.\, W_2 \\ W_2 &\stackrel{\text{def}}{=} \mathrm{b}_1\mathrm{rt} \,.\, W_2 \;+\; \mathrm{kr1} \,.\, W_2 \;+\; \mathrm{b}_1\mathrm{rf} \,.\, R_2 \;+\; \mathrm{kr2} \,.\, R_2 \\ R_2 &\stackrel{\text{def}}{=} \mathit{enter} \,.\, \mathit{exit} \,.\, \mathrm{b}_2\mathrm{wf} \,.\, P_2 \\ \mathsf{Sort}(P_2) &= \{\, \mathrm{b}_2\mathrm{wt},\ \mathrm{b}_2\mathrm{wf},\ \mathrm{kw1} \,\} \end{aligned}$$

The two processes P_1 and P_2 are then depicted by the following labelled transition systems:



The whole system is then the concurrent composition of the processes P_1 and P_2 with the variable processes:

$$\textsc{Peterson} \stackrel{\text{def}}{=} P_1 \parallel P_2 \parallel \mathrm{B_1f} \parallel \mathrm{B_2f} \parallel \mathrm{K_1}.$$

Exercise 14.6 (Solution on page 485)

Argue that the two processes P_1 and P_2 can never both be in the critical section at the same time.

14.5 A Message Delivery System

We now wish to specify a simple message delivery system, which models the sending of a message by a Sender process to a Receiver process. The Sender and Receiver are not directly connected; rather, the message is routed through some Medium. For example, the Sender and Receiver may be two devices on a local area network connected by an Ethernet; or they may be computers on opposite sides of the globe connected by a complex mesh of links between routers. In our simple system, we ignore the actual content of the message being sent, as well as the address of the Receiver, as there will only be a single Receiver process.

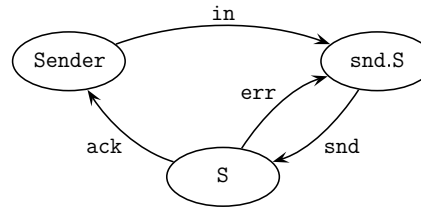
When the Sender accepts a message to send to the Receiver (modelled by an "in" action), it sends the message to the Medium (modelled by a "snd" action), and awaits an acknowledgement that the message has been successfully delivered to the Receiver (modelled by an "ack" action); when an acknowledgement is received, the Sender will be ready to accept the next message to send. It may be the case that the message is lost or corrupted by the Medium, in which case the Medium signals to the Sender that a fault has occurred (modelled by an "err" action); this typically occurs in practice through a time-out mechanism. The Sender responds to this fault by re-transmitting the message to the Medium. The behaviour of the Sender is thus modelled by the process Sender defined as follows:

$$\mathtt{Sender} \stackrel{\mathrm{def}}{=} \mathtt{in.snd.S}$$

$$\mathtt{S} \stackrel{\mathrm{def}}{=} \mathtt{ack.Sender} + \mathtt{err.snd.S}$$

$$\mathsf{Sort}(\mathtt{Sender}) = \{\mathtt{snd}, \mathtt{ack}, \mathtt{err}\}$$

Its transition system is depicted thus:

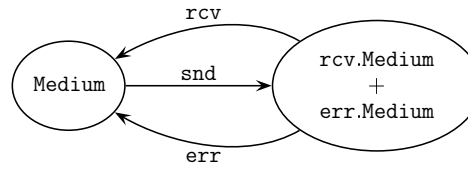


The MEDIUM accepts a message from the SENDER(via the snd action), and either delivers it to the RECEIVER (modelled by a rcv action) and returns to its original state to await the next message to be sent, or it signals to the SENDER that a fault has occurred (via the err action) and again returns to its original state to await the retransmission of the previous message. The behaviour of the MEDIUM is thus modelled by the process Medium defined as follows:

$$\mathtt{Medium} \stackrel{\mathrm{def}}{=} \mathtt{snd.(rcv.Medium} + \mathtt{err.Medium)}$$

$$\mathsf{Sort}(\mathtt{Medium}) = \{\mathtt{snd}, \mathtt{rcv}, \mathtt{err}\}$$

Its transition system is depicted thus:

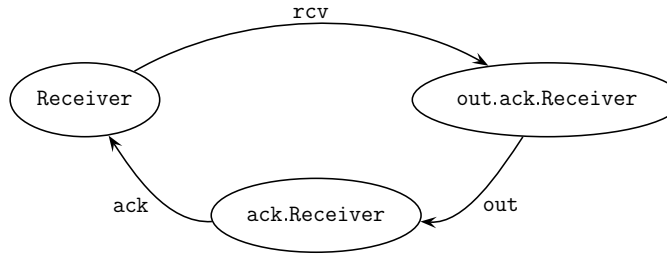


Finally, the RECEIVER awaits the delivery of a message (via the rcv action), and outputs the message (modelled by an out action) before issuing an acknowledgement (via the ack action) that the message has been successfully received and delivered. Its behaviour is modelled by the process Receiver defined as follows:

$$\mathtt{Receiver} \stackrel{\mathrm{def}}{=} \mathtt{rcv.out.ack.Receiver}$$

$$\mathsf{Sort}(\mathtt{Receiver}) = \{\mathtt{rcv}, \mathtt{ack}\}$$

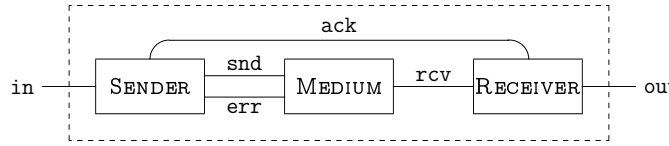
The transition system is depicted thus:



The complete system is defined to be the composition of these three components:

$$\texttt{System} \stackrel{\text{def}}{=} \texttt{Sender} \parallel \texttt{Medium} \parallel \texttt{Receiver}$$

and has the following configuration:



Note that in this simple model, the acknowledgement is communicated directly from the RECEIVER to the SENDER, which is unrealistic given the purpose of the System to relay messages between them. In reality, the acknowledgement would be routed through the MEDIUM resulting in a second phase which is identical to the first but with the roles of the SENDER and RECEIVER reversed.

The behaviour of the complete system is thus depicted by the transition system depicted in Figure 14.3.

Exercise 14.7 (Solution on page 485)

Enhance the simple message passing system so that acknowledgements are routed through the MEDIUM from the RECEIVER to the SENDER. Don't neglect the possibility of acknowledgements being lost.

14.6 Alternating Bit Protocol

The message passing system in the previous section is an example of a very important concept in communication networks: that of ***communication protocols***.

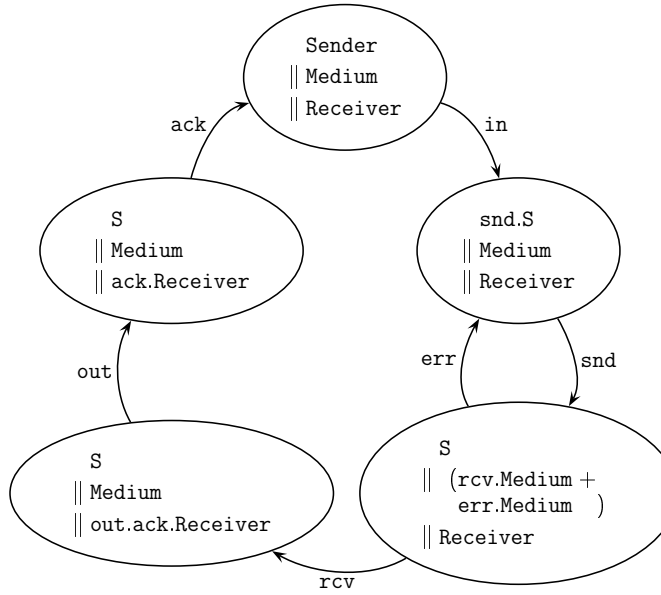


Figure 14.3: The message-passing system.

When you click on a link in your favourite browser to a Web site on the opposite side of the globe, or send an email to your friend who is perhaps thousands of miles away, a complicated procedure is carried out between dozens of computers in relaying your message to its destination (either the computer hosting the Web site you are wanting to access, or the computer on which your friend reads email). Your message gets relayed, bit-by-bit, through a long chain of intermediate computers as it gets routed towards its destination. At any point in this chain, a bit of your message can get lost in transmission, and the particular computer which sent the bit that got lost needs to know that the bit was lost so that it can retransmit it.

Of course, one computer cannot tell another computer that it didn't get a message, as it wouldn't know that it was supposed to get one; and in fact the most common cause of a message being lost in transmission is due to a receiving computer being broken, and thus unable to receive the message or send an acknowledgement. Thus, when messages are passed from one

computer to another, the sending computer will wait for an acknowledgement from the receiving computer; if this doesn't come within a reasonable amount of time, the sending computer will assume that the message got lost and retransmit it. Of course, it may be the acknowledgement that got lost: the receiving computer may receive a message and send an acknowledgement and subsequently receive the same message again. In this case, the receiver will assume that its acknowledgement was lost, leading the sender to retransmit the message, in which case the receiver will retransmit the acknowledgement.

There are very many different communication protocols implemented on computers carrying out the above task. In this section we consider a common simple protocol: the alternating bit protocol. This protocol again involves a SENDER and a RECEIVER communicating through a MEDIUM, and works as follows.

- The SENDER accepts a message to be sent to the RECEIVER (modelled by an "in" action), and sends it into the MEDIUM tagged with a protocol bit 0 or 1 (modelled by the actions "$\mathtt{s}_0$" and "$\mathtt{s}_1$", respectively). It then awaits an acknowledgement from the MEDIUM tagged by the same protocol bit (modelled by the actions "$\mathtt{ack}_0$" and "$\mathtt{ack}_1$", respectively).

  When the SENDER receives an acknowledgement tagged by the correct protocol bit, it flips the protocol bit and repeats the protocol for the next message.

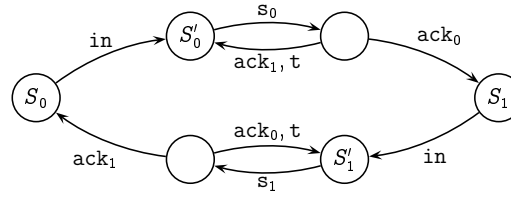
  If the SENDER receives an acknowledgement tagged by the wrong protocol bit, or if it times out waiting for the acknowledgement to arrive (modelled by a "t" action), it retransmits the message (again with the corresponding bit attached).

  The behaviour of the SENDER is thus defined by the following process:

$$\textsc{Sender} \stackrel{\text{def}}{=} S_0 \qquad \mathsf{Sort}(\textsc{Sender}) = \{\mathtt{s}_0, \mathtt{s}_1\}$$

$$S_0 \stackrel{\text{def}}{=} \mathtt{in}.S_0' \qquad S_0' \stackrel{\text{def}}{=} \mathtt{s}_0.(\mathtt{ack}_0.S_1 + \mathtt{ack}_1.S_0' + \mathtt{t}.S_0')$$

$$S_1 \stackrel{\text{def}}{=} \mathtt{in}.S_1' \qquad S_1' \stackrel{\text{def}}{=} \mathtt{s}_1.(\mathtt{ack}_1.S_0 + \mathtt{ack}_0.S_1' + \mathtt{t}.S_1')$$



The synchronisation sort of the SENDER process contains the actions s_0 and s_1, as the only way the MEDIUM could receive a message is through a communication with the SENDER; it can only do these actions if and when the SENDER does them.

- When the RECEIVER receives a message from the MEDIUM tagged by the expected protocol bit (modelled by the actions "r_0" and "r_1", respectively), it outputs the message (modelled by an "out" action) and sends an acknowledgement into the MEDIUM tagged by that protocol bit (modelled by the actions "$rack_0$" and "$rack_1$", respectively).

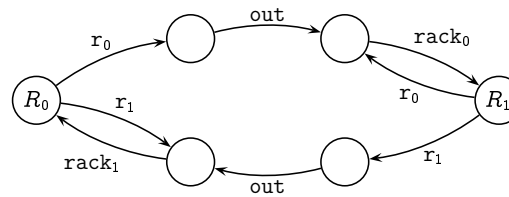
  The RECEIVER then awaits a new message tagged by the opposite protocol bit, with which it will repeat this protocol. In the meantime, it will acknowledge any further messages tagged by the old bit.

  The behaviour of the RECEIVER is thus defined by the following process:

$$\textsc{Receiver} \stackrel{\text{def}}{=} R_0 \qquad \mathsf{Sort}(\textsc{Receiver}) = \{rack_0, rack_1\}.$$

$$R_0 \stackrel{\text{def}}{=} r_0.out.rack_0.R_1 + r_1.rack_1.R_0$$

$$R_1 \stackrel{\text{def}}{=} r_1.out.rack_1.R_0 + r_0.rack_0.R_1$$

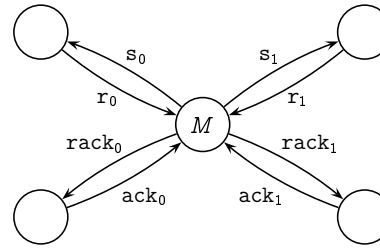


  The synchronisation sort of the RECEIVER process contains the actions $rack_0$ and $rack_1$, as the only way the MEDIUM could receive an acknowledgement is through a communication with the RECEIVER; it can only do these actions if and when the RECEIVER does them.

- The MEDIUM merely passes messages from the SENDER to the RECEIVER and acknowledgements from the RECEIVER to the SENDER. Its behaviour is defined by the following process:

$$\textsc{Medium} \stackrel{\text{def}}{=} M \qquad \mathsf{Sort}(\textsc{Medium}) = \{r_0, r_1, ack_0, ack_1\}.$$

$$M \stackrel{\text{def}}{=} s_0.r_0.M + s_1.r_1.M + rack_0.ack_0.M + rack_1.ack_1.M$$

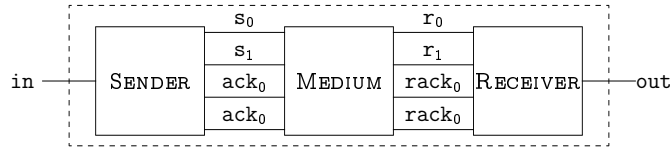


The synchronisation sort of the MEDIUM process does not contain the actions s_0 and s_1; the SENDER may send a message without it being received by the MEDIUM. Nor does it contain the actions $rack_0$ and $rack_1$; the RECEIVER may send an acknowledgement without it being received by the MEDIUM. However, it does contain the actions r_0 and r_1, as the RECEIVER can only receive a message from the MEDIUM; as well as the actions ack_0 and ack_1, as the SENDER can only receiver a message from the MEDIUM.

The complete system is defined to be the composition of these three components

$$\mathtt{System} \stackrel{\text{def}}{=} \mathtt{Sender} \parallel \mathtt{Medium} \parallel \mathtt{Receiver}$$

and has the following configuration:



Its complete transition system is large, but by considering it carefully, it can be verified that the in and out actions occur in an alternating fashion and, equally important, that the protocol can never deadlock.

Exercise 14.8 (Solution on page 488)

Argue that the in and out actions occur in alternating fashion in the alternating bit protocol, and that the protocol can never deadlock.

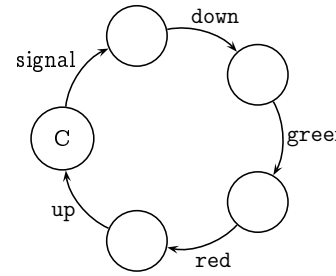
14.7 Additional Exercises

1. (a) Give a definition for a 3-counter C_3, and draw its labelled transition system.

(b) Draw the labelled transition system for $C \parallel C \parallel C$, where C is the 1-counter given in Section 14.2.

(c) Prove that $C_3 \sim C \parallel C \parallel C$.

2. Normally, a barrier at a railway level crossing remains up until a train arrives; this signals the controller, which then lowers the barrier, then turns the signal green, then turns the signal red again, and finally raises the barrier once again. Such a controller C is thus represented by the following LTS:



(a) Give a definition for C. This includes defining its sort $\mathcal{L}$(C).

(b) Give the definitions and associated LTS for the new Road and Rail systems Ro and Ra, respectively, which correspond to the new Controller C. (Keep in mind that the new Road system starts in a state where the barrier is up; and the new Rail system must signal the controller when a train arrives, using the new event "signal" common to their sorts.)

(c) Now consider the liveness properties again.

i. Is it now the case that, if the barrier is down when a car arrives, then the barrier will eventually go up?

ii. Is it now the case that, if the signal is red when a train arrives, then the signal will eventually turn green?

3. Argue that the process for Peterson's Algorithm can never deadlock.

4. Model Dekker's Algorithm for mutual exclusion, as outlined as follows.

P_1:	P_2:
`while true do`	`while true do`
$\cdots$ *noncritical section* $\cdots$	$\cdots$ *noncritical section* $\cdots$
`b1 := true;`	`b2 := true;`
`while b2 do`	`while b1 do`
`  if k=2 then`	`  if k=1 then`
`    b1 := false`	`    b2 := false`
`    while k=2 do skip`	`    while k=1 do skip`
`    b1 := true`	`    b2 := true`
$\cdots$ *critical section* $\cdots$	$\cdots$ *critical section* $\cdots$
`k := 2;`	`k := 1;`
`b1 := false`	`b2 := false`

5. Argue that the complete alternating bit protocol system can never deadlock, and that the `in` and `out` actions alternate as desired.

6. Show that the operator $\|$ is commutative, by showing that the transition systems defined by $E \parallel F$ and $F \parallel E$ are isomorphic (ie, identical, disregarding the – irrelevant – labels of the states).

7. Show that the operator $\|$ is *not* associative, by showing that $(E \parallel F) \parallel F \nsim E \parallel (F \parallel F)$, where $E \stackrel{\text{def}}{=} a.\mathbf{0}$ with $\text{Sort}(E) = \{a\}$ and $F \stackrel{\text{def}}{=} a.b.\mathbf{0}$ with $\text{Sort}(F) = \{b\}$.

   What restriction on synchronisation sorts would make this operator associative?

8. Consider a new parallel operator $E \mid F$ defined by the following transition rules:

   (a) If $E \stackrel{a}{\rightarrow} E'$ and $a \notin \text{Sort}(F)$ then $E \mid F \stackrel{a}{\rightarrow} E' \mid F$.
   (b) If $F \stackrel{a}{\rightarrow} F'$ and $a \notin \text{Sort}(E)$ then $E \mid F \stackrel{a}{\rightarrow} E \mid F'$.
   (c) If $E \stackrel{a}{\rightarrow} E'$ and $F \stackrel{a}{\rightarrow} F'$ and $a \in \text{Sort}(E) \cap \text{Sort}(F)$ then $E \mid F \stackrel{a}{\rightarrow} E' \mid F'$.

   This is identical to the synchronisation merge except that the transition rule for synchronising processes requires the action on which the processes synchronise to be in the sorts of *both* processes rather than just one of them.

   Show that the operator $\mid$ is both commutative and associative.