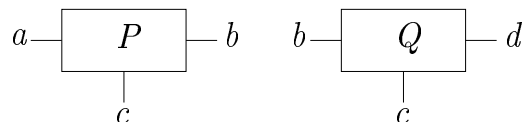
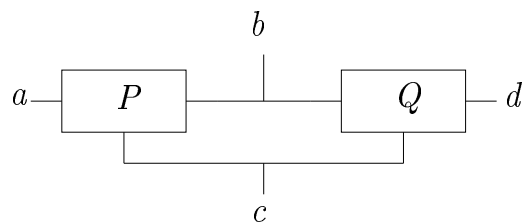


◇ Connection Diagrams ◇

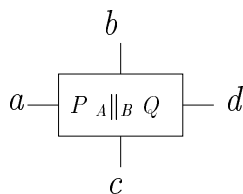
We can think of a process P with alphabet $A = \{a, b, c\}$ as a box with three possible points of connection to the outside world. Similarly, Q with alphabet $B = \{b, c, d\}$.



If P and Q are put in parallel, b and c are events which they may (indeed must) jointly participate in. This can be represented by joining the appropriate lines; of course, the events a and d are still available for connection to other processes.



Of course we can also consider the process $P_A \parallel_B Q$ as a single box:



◇ Generalized Operators ◇

We have seen binary (two-argument) forms of internal and external choice, and parallel composition. There are more general forms of all these operators, which provide a compact notation for a combination of an arbitrary number of processes.

Suppose we want to define a process $RELAY$ with n input channels of type T (called $in.1 \dots in.n$) and n output channels of type T (called $out.1 \dots out.n$). This process should receive a message on any input channel and send it out on the corresponding output channel, repeatedly.

We need to define

$$\begin{aligned} RELAY &= in.1?x : T \rightarrow out.1!xT \rightarrow RELAY \\ &\square in.2?x : T \rightarrow out.2!xT \rightarrow RELAY \\ &\vdots \\ &\square in.n?x : T \rightarrow out.n!x \rightarrow RELAY. \end{aligned}$$

It is possible to shorten this definition as follows:

$$RELAY = \square_{i \in \{1, \dots, n\}} in.i?x : T \rightarrow out.i!x \rightarrow RELAY$$

In general, if I is a finite indexing set and for each $i \in I$ there is a process P_i , then the process

$$\square_{i \in I} P_i$$

is defined.

It behaves as we would expect, given the example above. Formally the transition rules are

$$\frac{P_j \xrightarrow{a} P'}{ \Box_{i \in I} P_i \xrightarrow{a} P' } \quad j \in I$$

and, to deal with internal events:

$$\frac{P_j \xrightarrow{\tau} P'_j}{ \Box_{i \in I} P_i \xrightarrow{\tau} \Box_{i \in I} P'_i } \quad j \in I$$

In the second rule, $P'_i = P_i$ for $i \neq j$.

◇ General Internal Choice ◇

The same applies to internal choice. If I is an indexing set (finite or infinite) and for each $i \in I$ there is a process P_i , then the process

$$\Box_{i \in I} P_i$$

is a process which can behave like any of the P_i . Here is the transition rule.

$$\frac{}{ \Box_{i \in I} P_i \xrightarrow{\tau} P_i } \quad i \in I$$

Example: A random number generator could be described by the process

$$\Box_{i \in \mathbb{N}} \text{out}!i \rightarrow \text{Stop}$$

Remember that \Box is a specification construct.

◇ General Parallel ◇

If I is a finite indexing set such that for each $i \in I$ there is a process P_i and an interface set A_i , then the process

$$\parallel_{A_i}^{i \in I} P_i$$

is defined.

Any event a requires synchronisation from all processes P_i for which $a \in A_i$.

Example: A group of people must all be present for a meeting to take place. If N is the set of all the people's names, then we can define the interface and behaviour of each person as follows.

$$A_n = \{\text{enter}.n, \text{leave}.n, \text{meeting}\}$$

$$\begin{aligned} \text{PERSON}_n &= \text{enter}.n \rightarrow \text{PRESENT}_n \\ \text{PRESENT}_n &= \text{leave}.n \rightarrow \text{PERSON}_n \\ &\quad \Box \text{meeting} \rightarrow \text{PRESENT}_n \end{aligned}$$

The process

$$\text{GROUP} = \parallel_{A_n}^{n \in N} \text{PERSON}_n$$

describes the situation.

◇ Shared Resources ◇

It is common in concurrent systems for a resource to be shared between a number of processes. Examples might be a printer or a file server, or an individual file. It is straightforward to describe this kind of situation by placing several processes in parallel.

Example: Two users sharing a printer:

$PRINTER = request1 \rightarrow print \rightarrow PRINTER$

□ $request2 \rightarrow print \rightarrow PRINTER$

$USER1 = request1 \rightarrow work1 \rightarrow USER1$

$USER2 = request2 \rightarrow work2 \rightarrow USER2$

The parallel combination

$USER1 \parallel USER2 \parallel PRINTER$

allows each user to work independently, but requires synchronisation on *request1* and *request2* events. If both users want to print at the same time, one of them gets in first and the other has to wait.

This is fine, although there is nothing to prevent *USER1* from getting access to the printer every time, and excluding *USER2*.

◇ Deadlock ◇

Now consider a situation in which there are two shared resources, and both of them must be acquired before some task can be carried out. One example would be two shared files, and two programs, both of which need access to both files simultaneously.

Here is an example borrowed from Schneider. Two children share a paintbox and an easel. If one child wants to paint, she has to find the box and the easel; after painting, she drops both the box and the easel.

$JANE = jane.get.box \rightarrow jane.get.easel \rightarrow jane.paint \rightarrow jane.put.box \rightarrow jane.put.easel \rightarrow JANE$

□ $jane.get.easel \rightarrow jane.get.box \rightarrow jane.paint \rightarrow jane.put.easel \rightarrow jane.put.box \rightarrow JANE$

$KATE = kate.get.box \rightarrow kate.get.easel \rightarrow kate.paint \rightarrow kate.put.box \rightarrow kate.put.easel \rightarrow KATE$

□ $kate.get.easel \rightarrow kate.get.box \rightarrow kate.paint \rightarrow kate.put.easel \rightarrow kate.put.box \rightarrow KATE$

The easel and the box can each be used by just one child at a time.

$EASEL = jane.get.easel \rightarrow jane.put.easel \rightarrow EASEL$
 $\square kate.get.easel \rightarrow kate.put.easel \rightarrow EASEL$

$BOX = jane.get.box \rightarrow jane.put.box \rightarrow BOX$
 $\square kate.get.box \rightarrow kate.put.box \rightarrow BOX$

The combination of the two girls, the box and the easel is

$PAINTING = JANE \parallel KATE \parallel EASEL \parallel BOX$

There is a problem with these definitions. If both children decide to paint at about the same time, it is possible that one of them finds the box (for example, $jane.get.box$ happens) and then the other finds the easel (for example, $kate.get.easel$). Then none of the processes can do another event: $JANE$ is waiting to do $jane.get.easel$ and $KATE$ is waiting to do $kate.get.box$. In effect, each child is waiting for the other, and nothing happens. The system as a whole, after doing two events, has reached a state of *Stop*. This is an example of a *deadlock*.

\triangle Draw a transition diagram for this system.

Another example (also from Schneider): two furniture movers who need to move a table and a piano. Each object requires two people to lift it.

$PETE = lift.piano \rightarrow PETE$
 $\square lift.table \rightarrow PETE$

$DAVE = lift.piano \rightarrow DAVE$
 $\square lift.table \rightarrow DAVE$

$TEAM = PETE \parallel DAVE$

If both people make the same choice, they are able to cooperate in lifting an object. If their choices are different, then the result is deadlock:

$PETE \xrightarrow{\tau} lift.piano \rightarrow PETE$

$DAVE \xrightarrow{\tau} lift.table \rightarrow DAVE$

and

$lift.piano \rightarrow PETE \parallel lift.table \rightarrow DAVE$

cannot do anything; it is equivalent to *Stop*, or deadlock.

\triangle Draw a transition diagram for this system, including the τ transitions.

If the definition of $PETE$ is changed, then the problem can be avoided:

$$PETE' = lift.piano \rightarrow PETE$$

$$\square lift.table \rightarrow PETE$$

In these examples, our intention was to produce a system whose behaviour continues indefinitely, and we view termination (reaching *Stop*) as deadlock. If we want to distinguish between intended and unintended termination, then we can introduce a new event to indicate successful termination. (Conventional CSP notation for such an event is \checkmark , and the process *Skip* is defined by $\checkmark \rightarrow Stop$. Roscoe's presentation of CSP deals with *Skip* in detail; we will not use it.)

In general, if we want to check whether a given process can deadlock, we have to examine all its possible behaviours (effectively constructing a state transition diagram) and look to see whether any *Stop* states appear. An alternative is to exploit regularity in the structure of the process to construct a mathematical argument proving that deadlock is impossible.

FDR can check for possible deadlocks in a system, and is able to handle reasonably large systems (containing a few million states) efficiently.

◇ Channels and Connections ◇

$$COPYBIT = in?x : \{0, 1\} \rightarrow out!x \rightarrow COPYBIT$$

where we suppose that

$$\alpha in(COPYBIT) = \alpha out(COPYBIT) = \{0, 1\}.$$

COPYBIT has two channels, *in* and *out*. It repeatedly receives a single bit on the *in* channel and outputs it on the *out* channel.

$$\alpha COPYBIT = \{in.0, in.1, out.0, out.1\}.$$

By convention, a channel is used for communication between two processes, and in one direction only. Each channel of a process is either an output channel or an input channel, according to its use.

In connection diagrams, channels are drawn as arrows, labelled with the channel name.

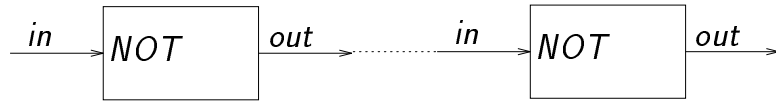


A variation on *COPYBIT* is an inverter:

$$NOT = in?x : \{0, 1\} \rightarrow out!(1-x) \rightarrow NOT$$

This illustrates the way that in general an output value may be an expression involving values which have previously been input.

Suppose we want to connect two copies of *NOT* together, so that the output of one becomes the input of the other.



We would like to do this by placing them in parallel, but there is a problem: an input *in.0* or *in.1* is ambiguously an input for both processes, and there is no link between the *out* channel on the left and the *in* channel to which it should be connected.

To solve this problem we introduce some new notation: *renaming*. Defining two functions *f* and *g* on events by

$$\begin{aligned} f(out.x) &= mid.x & g(out.x) &= out.x \\ f(in.x) &= in.x & g(in.x) &= mid.x \end{aligned}$$

(so we have also introduced a new channel called *mid*) then *f*(*NOT*) is *NOT* with all events renamed according to *f*.

f(*NOT*) behaves as if defined by

$$f(NOT) = in?x : \{0, 1\} \rightarrow mid!(1-x) \rightarrow f(NOT)$$

and similarly *g*(*NOT*) behaves as if defined by

$$g(NOT) = mid?x : \{0, 1\} \rightarrow out!(1-x) \rightarrow g(NOT).$$

In general, if *P* is any process and *f* : $\alpha P \rightarrow A$ is a function, the *f*(*P*) has alphabet *A* and has transitions defined by

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

Now we can form *f*(*NOT*) || *g*(*NOT*), and events on the *mid* channel represent messages sent from *f*(*NOT*) to *g*(*NOT*). Synchronisation is required for the events *mid.0* and *mid.1*.

A possible execution of *f*(*NOT*) || *g*(*NOT*) is:

In general if c is an output channel of P and an input channel of Q , then in $P \parallel Q$ communication occurs on channel c each time P does the event $c.v$ (outputs message v) and Q simultaneously does the event $c.v$ (inputs message v). Q is prepared to accept any $c.x$, so it is P which determines the actual message.

We require $\alpha c(P) = \alpha c(Q)$. We can then write αc for $\alpha c(P)$.

In $f(NOT) \parallel g(NOT)$ the $mid.0$ and $mid.1$ events are visible outside the system. Potentially they could be interfered with by other processes, although we would not normally want this to happen; for example,

$$f(NOT) \parallel g(NOT) \parallel Stop_{\alpha mid}$$

cannot do the mid events.

The hiding operator can be used to convert mid into a local channel:

$$(f(NOT) \parallel g(NOT)) \setminus \alpha mid.$$