◇ Choice between processes ◇

We have used | and menu choice to describe processes which have alternative behaviours. We have emphasised that | is *not* an operation on processes, but can only be used in conjunction with distinct prefixing events.

However, CSP does have operators which can be used to provide a choice between two (or more) existing processes. They are:

external choice - the environment can choose between the various processes

internal choice - the choice is made within the process, and cannot be observed by the environment.

By *the environment*, we mean whatever processes are in parallel with the process containing the choice.

The distinction between choice made by a process and choice made by its environment is important, because problems could arise if two processes have both been given control over a particular choice.

◇ External Choice ◇

The process $P \Box Q$ (pronounced "P external choice Q") is initially prepared to do any event which either P or Q could do. After the first event, the behaviour is either that of P or that of Q, depending on which process did the event. The choice is called "external" because the environment (another process in parallel) can choose the first event.

Example: The journey from A (the bus station) to B is covered by two bus routes: the 37 and the 111. If both buses are present at the bus station, then the service offered to the passenger is described by the process

$$SERVICE = BUS37 \Box BUS111.$$

The passenger can choose which bus to use.

Here are possible definitions:

$$\begin{aligned} &BUS37 = \\ &\quad board.37.A \rightarrow (pay.90 \rightarrow alight.37.B \rightarrow STOP \\ &\qquad\qquad\qquad\quad \mid alight.37.A \rightarrow STOP) \end{aligned}$$

$$\begin{aligned} &BUS111 = \\ &\quad board.111.A \rightarrow (pay.70 \rightarrow alight.111.B \rightarrow STOP \\ &\qquad\qquad\qquad\quad \mid alight.111.A \rightarrow STOP) \end{aligned}$$

Note that in this case, we do not think of events such as $alight.111.B$ as related to input or output.

If the passenger is defined by

$$\begin{aligned} PASS &= board.37.A \rightarrow pay.90 \\ &\qquad \rightarrow alight.37.B \rightarrow STOP \end{aligned}$$

then we can consider what happens when the passenger and the bus service interact, i.e. when we construct

$$SERVICE \;_{\alpha(SERVICE)}\|_{\alpha(PASS)}\; PASS.$$

$SERVICE$ can behave either as $BUS37$ or as $BUS111$, and the choice is made by the environment. The fact that $PASS$ can only do $board.37$ as its first event, means that $BUS37$ is chosen.

The system behaves exactly as if we had written

$$\begin{aligned} &SERVICE \\ &= board.37.A \rightarrow \; (pay.90 \rightarrow alight.37.B \rightarrow STOP \\ &\qquad\qquad\qquad\quad \mid alight.37.A \rightarrow STOP) \\ &\mid \; board.111.A \rightarrow (pay.70 \rightarrow alight.111.B \rightarrow STOP \\ &\qquad\qquad\qquad\quad \mid alight.111.A \rightarrow STOP) \end{aligned}$$

In general, $(a \rightarrow P) \Box (b \rightarrow Q)$ is equivalent to $a \rightarrow P \mid b \rightarrow Q$, and it is possible to use $\Box$ instead of | (this is what FDR does).

However, we can also write $(a \rightarrow P) \Box (a \rightarrow Q)$ (remember that $a \rightarrow P \mid a \rightarrow Q$ is illegal) — we will see what this means soon.

◇ Defining External Choice ◇

Here are the transition rules for external choice.

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'}$$

$$\frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{\tau} P \Box Q'}$$

The first two capture the intention that the choice is resolved by the first event. The second two allow either process to change state internally without resolving the choice.

Example: Going back to

$$SERVICE = BUS37 \Box BUS111$$

we have the transitions

$$\begin{aligned} &SERVICE \xrightarrow{board.37.A} \\ &\quad pay.90 \rightarrow \ldots \mid alight.37.A \rightarrow STOP \end{aligned}$$

$$\begin{aligned} &SERVICE \xrightarrow{board.111.A} \\ &\quad pay.70 \rightarrow \ldots \mid alight.111.A \rightarrow STOP \end{aligned}$$

◇ Internal Choice ◇

The process $P \sqcap Q$ describes a choice between P and Q, but the environment has no control over the choice. Internal choice is often also known as *nondeterministic choice*. The choice is resolved internally by the process.

Suppose the bus company agrees to provide a bus from A to B, but does not say whether it will be the 37 or the 111. The situation at the bus station is now described by the process

$$SERVICE = BUS37 \sqcap BUS111.$$

We should interpret this as a specification of a bus service. The company could implement the service by always providing bus 37, or by deciding each morning which bus to provide, etc. The passenger has no control over the decision, and cannot tell which bus will be available until she arrives at the bus station.

If a system is specified by the description $P \sqcap Q$, then all of the following are acceptable implementations.

- ◇ provide both P and Q, and use some internal means to choose between them
- ◇ just provide P
- ◇ just provide Q

◇ Internal Choice ◇

To define internal choice by means of transition rules, we use the *internal event* τ. A transition $P \xrightarrow{\tau} Q$ represents a change of state which is not accompanied by any observable event; it is a change of state whose occurrence cannot be observed directly by the environment. We use τ transitions to model the resolution of an internal choice.

Here are the transition rules:

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

Note that these rules capture one approach to implementing $P \sqcap Q$, namely to implement both P and Q and then choose between them at random. In order to give transition rules we are forced to choose an implementation, and this is the most general.

◇ Example ◇

Consider

$$SERVICE = BUS37 \sqcap BUS111$$

again, and put it in parallel with $PASS$. According to the transition rules, the first event which $SERVICE$ does will be a τ event, resulting in either $BUS37$ or $BUS111$. All the events of $PASS$ require synchronisation, so nothing can happen until τ has been done.

There are two ways for $SERVICE$ to do τ. The first results in

$$BUS37 \;{}_{\alpha(SERVICE)}\|_{\alpha(PASS)}\; PASS$$

and then $PASS$ can interact with $BUS37$.

The other possibility results in

$$BUS111 \;{}_{\alpha(SERVICE)}\|_{\alpha(PASS)}\; PASS$$

and now the whole system stops because $BUS111$ and $PASS$ cannot synchronise on any events. This is another example of *deadlock*.

◇ Another example ◇

Keep the definition

$$SERVICE = BUS37 \sqcap BUS111$$

and suppose that there is also a train service from A to B, described by the process $TRAIN$. Now the options available to the passenger are described by the process

$$TRAIN \;\Box\; SERVICE$$

which expands to

$$TRAIN \;\Box\; (BUS37 \sqcap BUS111).$$

We have the transition

$$BUS37 \sqcap BUS111 \xrightarrow{\tau} BUS37$$

and so the transition rules for external choice give

$$\begin{tikzcd} TRAIN \;\Box\; (BUS37 \sqcap BUS111) \arrow[d, "\tau"] \\ TRAIN \;\Box\; BUS37 \end{tikzcd}$$

We can interpret this transition as the fact that one bus service may disappear while the passenger is still thinking about whether to take the bus or the train.

If the definition of $TRAIN$ is

$$TRAIN = board.train.A \rightarrow alight.train.B \rightarrow STOP$$

then there is also the transition

$$\begin{tikzcd} TRAIN \,\square\, (BUS37 \sqcap BUS111) \arrow[d, "board.train.A"] \\ alight.train.B \rightarrow STOP \end{tikzcd}$$

which we can interpret as the passenger choosing the train without ever discovering which bus is available.

◇ Nondeterminism ◇

The first form of choice, $|$, is a special case of external choice. The process

$$a \rightarrow P \mid b \rightarrow Q$$

is equivalent to

$$a \rightarrow P \,\square\, b \rightarrow Q.$$

However, general external choice has some extra power. Because it is possible to construct an external choice between any two processes, we can write, for example

$$a \rightarrow P \,\square\, a \rightarrow Q$$

(recall that $a \rightarrow P \mid a \rightarrow Q$ is forbidden).

We consider $\rightarrow$ to have higher precedence than $\square$, so that this process is the same as

$$(a \rightarrow P) \,\square\, (a \rightarrow Q).$$

What does this mean? The process

$$a \rightarrow P \,\square\, a \rightarrow Q$$

can either do a and then behave like P, or do a and behave like Q. The environment cannot influence which of these possibilities will occur: all it can do is choose to do a in order to interact.

More generally, the external choice

$$a \rightarrow P \,\square\, a \rightarrow Q \,\square\, b \rightarrow R$$

allows the environment to choose between a and b, but if a is chosen then the subsequent behaviour could be that of either P or Q.

Using external choice with several occurrences of the same prefixing event leads to nondeterminism, in the sense that the event which is observed does not determine the subsequent behaviour.

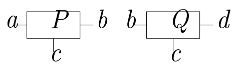
We will eventually see that

$$a \rightarrow P \,\square\, a \rightarrow Q = a \rightarrow P \sqcap a \rightarrow Q$$

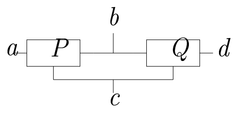
which emphasises the fact that the environment cannot choose between P and Q.

◇ Connection Diagrams ◇

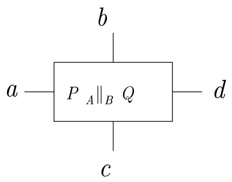
We can think of a process P with alphabet $A = \{a, b, c\}$ as a box with three possible points of connection to the outside world. Similarly, Q with alphabet $B = \{b, c, d\}$.



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 b and c are still available for connection to other processes.



Of course we can also consider the process $P \,{}_A\|_B\, Q$ 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 $in.1 \ldots in.n$) and n output channels of type T (called $out.1 \ldots out.n$. This process should receive a message on any input channel and send it out on the corresponding output channel, repeatedly.

We need to define

$$\begin{array}{rcl} RELAY & = & in.1?x :\to out.1!xT \to RELAY \\ & \Box & in.2?x :\to out.2!xT \to RELAY \\ & \vdots & \\ & \Box & in.n?x : T \to out.n!x \to RELAY. \end{array}$$

It is possible to shorten this definition as follows:

$$RELAY = \Box_{i \in \{1,\ldots,n\}}\ in.i?x :\to out.i!x \to RELAY$$

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

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

is defined.

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

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

and, to deal with internal events:

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

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

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

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

$$\frac{}{\sqcap_{i \in I}\ P_i \xrightarrow{\tau} P_i}\ i \in I$$

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

$$\sqcap_{i \in \mathbb{N}}\ out!i \to STOP$$

Remember that $\sqcap$ 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

$$\|_{A_i}^{i \in I}\ P_i$$

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.

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

$$\begin{array}{rcl} PERSON_n & = & enter.n \to PRESENT_n \\ PRESENT_n & = & leave.n \to PERSON_n \\ & \Box & meeting \to PRESENT_n \end{array}$$

The process

$$GROUP = \|_{A_n}^{n \in N}\ PERSON_n$$

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:

$$\begin{array}{rcl} PRINTER & = & request1 \to print \to PRINTER \\ & \Box & request2 \to print \to PRINTER \\ USER1 & = & request1 \to work1 \to USER1 \\ USER2 & = & request2 \to work2 \to USER2 \end{array}$$

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.

$$
\begin{aligned}
&ELLA = \\
&\quad (ella.get.box \rightarrow ella.get.easel \rightarrow ella.paint \rightarrow \\
&\qquad ella.put.box \rightarrow ella.put.easel \rightarrow ELLA) \\
&\square\ (ella.get.easel \rightarrow ella.get.box \rightarrow ella.paint \rightarrow \\
&\qquad ella.put.easel \rightarrow ella.put.box \rightarrow ELLA)
\end{aligned}
$$

$$
\begin{aligned}
&KATE = \\
&\quad (kate.get.box \rightarrow kate.get.easel \rightarrow kate.paint \rightarrow \\
&\qquad kate.put.box \rightarrow kate.put.easel \rightarrow KATE) \\
&\square\ (kate.get.easel \rightarrow kate.get.box \rightarrow kate.paint \rightarrow \\
&\qquad kate.put.easel \rightarrow kate.put.box \rightarrow KATE)
\end{aligned}
$$

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

$$
\begin{aligned}
&EASEL = \\
&\quad ella.get.easel \rightarrow ella.put.easel \rightarrow EASEL \\
&\square\ kate.get.easel \rightarrow kate.put.easel \rightarrow EASEL
\end{aligned}
$$

$$
\begin{aligned}
&BOX = \\
&\quad ella.get.box \rightarrow ella.put.box \rightarrow BOX \\
&\square\ kate.get.box \rightarrow kate.put.box \rightarrow BOX
\end{aligned}
$$

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

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

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, $ella.get.box$ happens) and then the other finds the easel (for example, $kate.get.easel$). Then none of the processes can do another event: $ELLA$ is waiting to do $ella.get.easel$ 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*.

△ 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.

$$
\begin{aligned}
PETE &= lift.piano \rightarrow PETE \\
&\sqcap\ lift.table \rightarrow PETE \\
\\
DAVE &= lift.piano \rightarrow DAVE \\
&\sqcap\ lift.table \rightarrow DAVE \\
\\
TEAM &= PETE \parallel DAVE
\end{aligned}
$$

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.

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

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

$$
\begin{aligned}
PETE' &= lift.piano \rightarrow PETE \\
&\square\ lift.table \rightarrow PETE
\end{aligned}
$$

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 ✓, and the process $SKIP$ is defined by ✓ $\rightarrow STOP$. 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.

◇ Livelock ◇

An attempt to prevent $ELLA$ and $KATE$ from deadlocking might adapt $ELLA$'s description so that she can return items before they have been used, rather than wait indefinitely for them to become available. Thus extra choices are introduced for $ELLA$ when she holds only one item.

$$
\begin{array}{rcl}
ELLA & = & ella.get.box \rightarrow \\
 & & \quad (ella.put.box \rightarrow ELLA \\
 & & \quad\quad \Box\ ella.get.easel \rightarrow ella.paint \rightarrow \\
 & & \quad\quad\quad ella.put.box \rightarrow ella.put.easel \rightarrow ELLA) \\
 & & \Box\ ella.get.easel \rightarrow \\
 & & \quad (ella.put.easel \rightarrow ELLA \\
 & & \quad\quad \Box\ ella.get.box \rightarrow ella.paint \rightarrow \\
 & & \quad\quad\quad ella.put.easel \rightarrow ella.put.box \rightarrow ELLA)
\end{array}
$$

If we are interested only in the $paint$ events, then we might hide the put and get events. The system we wish to consider is

$$SYSTEM = PAINTING \setminus INT$$

where

$$INT = \{ella, kate\}.\{put, get\}.\{easel, box\}$$

However, it is possible for $ELLA$ to loop forever, repeatedly getting an item and then immediately putting it back, without achieving any painting. Because these events are all hidden, this becomes an infinite loop of τ events. This is what CSP calls *livelock*, or *divergence*—the possibility of an infinite sequence of τ events.

FDR can be used to detect divergence, and indeed detects it for this example. (Select "Livelock" from the tabs below the menu bar, then select $SYSTEM$ in the "Implementation" field. Clicking on "Check" does a check for divergence.) A process that can diverge can never be guaranteed to make any real progress.

Because the parallel operator in CSP does not make any guarantees about how often each process will be executed, (it is not necessarily fair to $KATE$ or $ELLA$) and the choice operator makes no guarantees about how often each of its options will be executed, it is possible for this painting system to execute for ever without performing a $paint$ event. However, a real implementation might well be fair to $KATE$, and thus not be divergent in practice. Care is needed to ensure that the situation detected by FDR would really arise in the situation being modelled.

◇ Channels and Connections ◇

$$COPYBIT = in?x :\rightarrow out!x \rightarrow COPYBIT$$

where we suppose that

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

$COPYBIT$ 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.

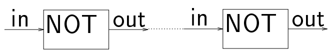
In connection diagrams, channels are drawn as arrows, labelled with the channel name.

A variation on $COPYBIT$ is an inverter:

$$NOT = in?x :\rightarrow out!(1-x) \rightarrow NOT$$

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

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

(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

$$f(NOT) = in?x :\to mid!(1-x) \to f(NOT)$$

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

$$g(NOT) = mid?x :\to out!(1-x) \to g(NOT).$$

In general, if P is any process and $f : \alpha(P) \to A$ 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 $f(NOT) \parallel g(NOT)$, 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 sequence of transitions of $f(NOT) \parallel g(NOT)$ is:

$$\langle in.1, mid.0, out.1, in.0, mid.1, in.0\rangle$$

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 $c(P) = c(Q)$. We can then write c for $c(P)$.

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,

$$f(NOT) \parallel g(NOT) \parallel STOP_{mid}$$

cannot do the mid events.

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

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