◇ Interaction ◇

Up to now we have described simple processes in isolation. Although we have often assumed that our processes might be placed in some environment and expected to interact with it — for example, there should be a customer who will use the ticket machine — this environment has not been made explicit.

We will now see how to take two (or more) processes and force them to interact with each other. Interaction between two processes means that they simultaneously perform events; an event thus becomes a joint activity in which two (or more) processes may participate.

When placing processes in parallel so that they can interact, it is important to specify which events they are supposed to be interacting on, or sharing. This is where alphabets (interfaces) come into play.

If the interfaces of processes P and Q are A and B respectively, then the process

$$P \ {}_A\|_B \ Q$$

is a parallel combination of P and Q.

In this combination, P can only perform events in A, Q can only perform events in B, and any events in the intersection of A and B require synchronisation between P and Q.

The interface of P should contain at least all the events used in the definition of P, and similarly for the interface of Q.

Example: Consider processes representing a vending machine, and a customer:

$$\begin{array}{rcl} VM & = & coin \rightarrow (choc \rightarrow STOP \mid toffee \rightarrow STOP) \\ CUST & = & coin \rightarrow choc \rightarrow STOP \end{array}$$

$$\alpha(VM) = \alpha(CUST) = \{coin, choc, toffee\} = A.$$

The process $VM \ {}_A\|_A \ CUST$ models the interaction of the customer with the machine. How does it behave? Any event done by $VM \ {}_A\|_A \ CUST$ must be an event which is done simultaneously by both VM and $CUST$.

At the first step, both VM and $CUST$ can do the event $coin$. We therefore expect $VM \ {}_A\|_A \ CUST$ to do $coin$. Subsequently, VM and $CUST$ enter new states which continue to interact.

After the event $coin$, VM becomes

$$choc \rightarrow STOP \mid toffee \rightarrow STOP$$

and $CUST$ becomes

$$choc \rightarrow STOP.$$

Synchronisation is still required for all events, and therefore only $choc$ can happen. The choice between $choc$ and $toffee$ in VM is resolved in favour of $choc$.

After the event $choc$, both processes become $STOP$, so the system becomes $STOP \ {}_A\|_A \ STOP$, which cannot do anything else.

We can draw a transition diagram for $VM \ {}_A\|_A \ CUST$.

$$\begin{tikzcd} VM \ {}_A\|_A \ CUST \arrow[d, "coin"] \\ (choc \rightarrow STOP \mid toffee \rightarrow STOP) \ {}_A\|_A \ choc \rightarrow STOP \arrow[d, "choc"] \\ STOP \ {}_A\|_A \ STOP \end{tikzcd}$$

In this example, both VM and $CUST$ continued to the end of their potential behaviour. This may not happen in general: if we change the definition to

$$CUST = coin \rightarrow STOP$$

then after the event $coin$ we get

$$(choc \rightarrow STOP \mid toffee \rightarrow STOP) \ {}_A\|_A \ STOP$$

and nothing further can happen. Although one of the processes could do either $choc$ or $toffee$, both of these events require synchronisation with the other process; but because $STOP$ cannot do anything, synchronisation is not possible.

Example: Recall the definition of $STUDENT$:

$$\begin{array}{rcl} STUDENT & = & year1 \rightarrow (pass \rightarrow YEAR2 \\ & & \qquad\qquad \mid fail \rightarrow STUDENT) \\ YEAR2 & = & year2 \rightarrow (pass \rightarrow YEAR3 \\ & & \qquad\qquad \mid fail \rightarrow YEAR2) \\ YEAR3 & = & year3 \rightarrow (pass \rightarrow graduate \\ & & \qquad\qquad\qquad \rightarrow STOP \\ & & \qquad\qquad \mid fail \rightarrow YEAR3) \end{array}$$

We will now explicitly state that the alphabet is

$$\begin{array}{rcl} \alpha(STUDENT) & = & \{year1, year2, year3, \\ & & pass, fail, graduate\} \end{array}$$

which we will abbreviate to S.

Suppose that the student has a generous parent, who buys a present every time the student passes the exams.

$$PARENT \;=\; pass \rightarrow present \rightarrow PARENT$$

Again we explicitly define the alphabet:

$$\alpha(PARENT) = \{pass, present\} = P.$$

Notice that the event $pass$ now has two different interpretations. For the student it means passing the exams, but for the parent it means seeing the student pass the exams.

We can now consider the parallel combination of the student and the parent:

$$STUDENT\ {}_S\|_P\ PARENT.$$

Synchronisation is required for the event $pass$, which is the only event in both alphabets. The other events can happen independently.

The behaviour of this system will be explored in Practical Sheet 2.

◇ More Processes ◇

Any number of processes can be put in parallel, by using the $\|$ operator repeatedly.

Example: Suppose the student has a tutor who is annoyed by failure.

$$TUTOR \;=\; fail \rightarrow shout \rightarrow TUTOR$$

$$\alpha(TUTOR) = \{fail, shout\} = T$$

We can add the tutor to the system consisting of the student and the parent.

$$(STUDENT\ {}_S\|_P\ PARENT)\ {}_{S\cup P}\|_T\ TUTOR$$

As before, $pass$ must be synchronised between $STUDENT$ and $PARENT$. Also, $fail$ (which is the only event in both $S \cup P$ and T) must be synchronised between $STUDENT\ {}_S\|_P\ PARENT$ and $TUTOR$.

We know that $fail$ events come from $STUDENT$ not $PARENT$, so in effect this means that $pass$ must be synchronised between $STUDENT$ and $PARENT$, and $fail$ must be synchronised between $STUDENT$ and $TUTOR$.

◇ More Synchronisation ◇

Some parallel combinations require some events to be synchronised between more than two processes.

Example: If a student completes the degree programme without failing at all, then the college awards a prize.

$$\begin{aligned}
COLLEGE &= fail \rightarrow STOP \mid pass \rightarrow C1\\
C1 &= fail \rightarrow STOP \mid pass \rightarrow C2\\
C2 &= fail \rightarrow STOP \mid\ pass \rightarrow\\
&\qquad\qquad\qquad prize \rightarrow STOP
\end{aligned}$$

$$\alpha(COLLEGE) = \{pass, fail, prize\} = C$$

Now we can consider combinations of $STUDENT$ with any or all of $PARENT$, $TUTOR$ and $COLLEGE$. If we combine everything:

$$((STUDENT\ {}_S\|_P\ PARENT)\ {}_{S\cup P}\|_T\ TUTOR)$$
$$_{S\cup P\cup T}\|_C\ COLLEGE$$

then $pass$ must be synchronised between $STUDENT$, $PARENT$ and $COLLEGE$, and so on.

Consider the processes $PASS$ ("passenger") and $TICKETS$, both with alphabet

$$A = \{ashford, staines, feltham, ticket, pound\}$$

defined by

$$\begin{aligned}
PASS &= ashford \rightarrow pound \rightarrow\\
&\qquad (ticket \rightarrow PASS\\
&\qquad \mid pound \rightarrow ticket \rightarrow PASS)\\
&\quad \mid\ feltham \rightarrow pound \rightarrow ticket \rightarrow STOP\\
TICKETS &= staines \rightarrow pound \rightarrow\\
&\qquad ticket \rightarrow TICKETS\\
&\quad \Box\ ashford \rightarrow pound \rightarrow pound \rightarrow\\
&\qquad ticket \rightarrow TICKETS
\end{aligned}$$

△ What is the behaviour of $TICKETS\ {}_A\|_A\ PASS$? Draw a transition diagram.

Given a transition diagram, it is possible to define a process, without using the parallel operator, which has the same transition diagram.

△ Do this for $TICKETS\ {}_A\|_A\ PASS$.

◇ Student and Parent ◇

The student and the parent, in parallel, behave more or less as we expected. The only slight surprise is that after the student has passed an exam, $present$ and the next $year$ can happen in either order. The transition diagram contains two squares, which are characteristic of a pair of events which must both happen but in either order.

If processes P and Q are completely independent (there are no events which are in both alphabets) then the number of states of $P\ {}_A\|_B\ Q$ is the product of the number of states of P and the number of states of Q. However, if the processes must synchronise on some events, this is no longer true. For example, $STUDENT$ has 8 states and $PARENT$ has 2 states, but their parallel combination has only 14 states. Because $pass$ cannot happen until after $year1$, $PARENT$ cannot get into its second state while $STUDENT$ is still in its first state.

Any process can be rewritten in a form which does not involve $\|$. Try it for $STUDENT\ {}_S\|_P\ PARENT$ — it becomes fairly complex. Roughly speaking, if P has m states and Q has n states, then $P\ {}_A\|_B\ Q$ has $m \times n$ states (although synchronisation might reduce the number).

If we define a process R which has the same transition diagram as $P\ {}_A\|_B\ Q$ but does not use $\|$, then the syntactic "size" of R will be $m \times n$. However, the syntactic size of $P\ {}_A\|_B\ Q$ is only $m + n$. Defining a system as a parallel combination of several processes is very compact, and is closer to the way we think about it.

◇ Prizes ◇

Recall the parallel combination of $STUDENT$, $PARENT$ and $COLLEGE$. If the student passes every year, then the system works as we intended and eventually $COLLEGE$ does $prize$. However, if $fail$ happens, then $COLLEGE$ becomes $STOP$ and cannot do anything else afterwards. This causes a problem because $pass$ and $fail$ must still be synchronised, and therefore $STUDENT$ can no longer either pass or fail — the whole system stops.

We need to change the definition of $COLLEGE$ so that after $fail$ it can still do $pass$ or $fail$ — but never do $prize$.

△ Write down the new definition of $COLLEGE$.

◇ Operational Semantics ◇

The *semantics* of a programming language is a definition of what expressions in the language (either complete programs or program fragments) mean. One style of semantics is *operational* — the meaning of program expressions is defined by describing how they should be executed. An operational semantics can be thought of as an idealised implementation, or as instructions to an implementor.

In CSP, we are interested in the events which a process may perform, and we have informally introduced the operators by describing when processes can do certain events. We will now introduce the idea of *labelled transitions* as the basis of the operational semantics of CSP. Labelled transitions allow us to define CSP operators more formally; they contain the same information as transition diagrams, but in a more manageable form.

A labelled transition has the form

$$P \xrightarrow{e} Q$$

where P and Q are processes and e is an event. It captures the idea that P can change state to Q by doing the event e.

Example: The execution of the process

$$coin \to choc \to STOP$$

can be described by the labelled transitions:

$$(coin \to choc \to STOP) \xrightarrow{coin} (choc \to STOP)$$
$$(choc \to STOP) \xrightarrow{choc} STOP$$

When defining CSP operators, we will use labelled transitions to precisely describe the possible behaviour of the processes being defined. We use *inference rules* of the form

$$\frac{\text{hypothesis } 1 \ldots \text{hypothesis } n}{\text{conclusion}}\ [\text{side condition}]$$

In such a rule, the hypotheses are usually labelled transitions of certain processes; the conclusion is a labelled transition of a process being defined by means of a new operator. Some rules have a *side condition*, which is an extra condition necessary for the rule to be applicable. We will often refer to these rules as *transition rules*.

The rule for prefixing is

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

There are no hypotheses, which means that we always know that $(a \to P) \xrightarrow{a} P$. This is true for *all* processes P, and *all* events a.

There is no transition rule for $STOP$. This means that it is never possible to deduce a transition for $STOP$, which is exactly what we want.

To define choice (from a finite number of alternatives) we use one rule for each possible initial event. For example, the process $a \to P \mid b \to Q$ is defined by the following pair of rules.

$$\frac{}{a \to P \mid b \to Q \xrightarrow{a} P}$$

$$\frac{}{a \to P \mid b \to Q \xrightarrow{b} Q}$$

For menu choice we use this rule:

$$\frac{}{x : A \to P(x) \xrightarrow{a} P(a)}\,[a \in A]$$

The side condition $a \in A$ indicates that the rule only applies to events in the specified set A of initial possibilities.

Notation: the use of x in the process $x : A \to P(x)$ suggests a general, as yet undetermined event. The use of a for the event labelling the transition represents a particular event. This usage follows the common mathematical convention of using letters close to the end of the alphabet as variables, and letters close to the beginning of the alphabet as constants.

When a named process is defined, we should be able to replace the name by its definition wherever it is used. The transition rule for named processes states that any transition of the right hand side of a definition is also a transition of the defined process.

$$\frac{P \xrightarrow{e} P'}{N \xrightarrow{e} P'}\,[N = P]$$

Example: If we define

$$DOOR = open \to close \to DOOR$$

then because we have

$$(open \to close \to DOOR) \xrightarrow{open} (close \to DOOR)$$

we also have

$$DOOR \xrightarrow{open} (close \to DOOR).$$

Then

$$(close \to DOOR) \xrightarrow{close} DOOR$$

This is all the information we need about the behaviour of $DOOR$.

Note: the operational semantics of CSP appears in "Concurrent and Real Time Systems: the CSP Approach" and Roscoe's "Theory and Practice of Concurrency" but not in Hoare's "Communicating Sequential Processes".

◇ Transitions for Concurrency ◇

Here are the transition rules for the concurrency operator.

$$\frac{P \xrightarrow{a} P'}{P \;{}_A\|_B\; Q \xrightarrow{a} P' \;{}_A\|_B\; Q}\,[a \in A, a \notin B]$$

$$\frac{Q \xrightarrow{a} Q'}{P \;{}_A\|_B\; Q \xrightarrow{a} P \;{}_A\|_B\; Q'}\,[a \in B, a \notin A]$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \;{}_A\|_B\; Q \xrightarrow{a} P' \;{}_A\|_B\; Q'}\,[a \in A \cap B]$$

◇ Examples ◇

Example: Processes VM and $CUST$ with

$$\begin{aligned} \alpha(VM) &= \{coin, choc, beep\} = A \\ \alpha(CUST) &= \{coin, choc, eat\} = B \\ VM &= coin \to beep \to choc \to VM \\ CUST &= coin \to choc \to eat \to CUST. \end{aligned}$$

In

$$VM \;{}_{\{coin,choc,beep\}}\|_{\{coin,choc,eat\}}\; CUST$$

the events $beep$ and eat happen independently, but $coin$ and $choc$ require synchronisation.

$$\begin{tikzcd}
VM \;{}_A\|_B\; CUST \arrow[d, "coin"] \\
beep \to choc \to VM \;{}_A\|_B\; choc \to eat \to CUST \arrow[d, "beep"] \\
choc \to VM \;{}_A\|_B\; choc \to eat \to CUST \arrow[d, "choc"] \\
VM \;{}_A\|_B\; eat \to CUST \arrow[d, "eat"] \\
VM \;{}_A\|_B\; CUST
\end{tikzcd}$$

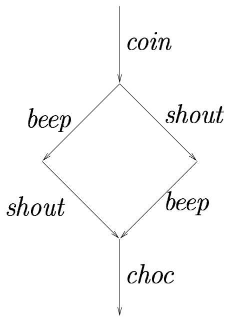
If we change $CUST$ so that

$$\begin{aligned}\alpha(CUST) &= \{coin, choc, shout\} = A\\ CUST &= coin \rightarrow shout \rightarrow choc \rightarrow CUST\end{aligned}$$

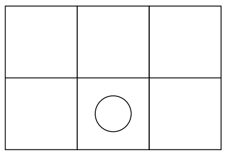
then

$VM\ {}_A\|_B\ CUST \xrightarrow{coin}$
$beep \rightarrow choc \rightarrow VM\ {}_A\|_B\ shout \rightarrow choc \rightarrow CUST$

and now $beep$ and $shout$, neither of which requires synchronisation, could happen in either order. Here is the complete transition diagram.



Example: To describe the movement of a counter on the board



we can define two processes:

$$\begin{aligned}\alpha(LR) &= \{left, right\}\\ \alpha(UD) &= \{up, down\}\\ LR &= left \rightarrow right \rightarrow LR \mid right \rightarrow left \rightarrow LR\\ UD &= up \rightarrow down \rightarrow UD\end{aligned}$$

and then

$$LR\ {}_{\{left,right\}}\|_{\{up,down\}}\ UD$$

describes the whole system.

An alternative way of describing this system is to define a collection of processes $R_{x,y}$ representing the behaviour when the counter starts from coordinate position (x, y):

$$\begin{aligned}R_{0,0} &= right \rightarrow R_{1,0} \mid up \rightarrow R_{0,1}\\ R_{0,1} &= right \rightarrow R_{1,1} \mid down \rightarrow R_{0,0}\\ &\dots\end{aligned}$$

and then

$$R_{1,0} = LR\ {}_{\{left,right\}}\|_{\{up,down\}}\ UD.$$

Because of the way synchronisation is needed for events in both alphabets, it is possible to control or restrict the behaviour of a process by adding another process in parallel.

Example: Recall that with the most recent definitions of VM and $CUST$, $VM \parallel CUST$ can do $beep$ and $shout$ in either order. If we define another process $CONTROL$ with

$$\begin{aligned}\alpha(CONTROL) &= \{beep, shout\} = C\\ CONTROL &= beep \rightarrow shout \rightarrow CONTROL\end{aligned}$$

then

$$(VM\ {}_A\|_B\ CUST)\ {}_{A\cup B}\|_C\ CONTROL$$

behaves like the process P defined by

$$P = coin \rightarrow beep \rightarrow shout \rightarrow choc \rightarrow P.$$

This also illustrates the need to be careful about alphabets: if

$$\alpha(CONTROL) = \{beep, shout, coin, choc\} = D$$

and $CONTROL$ has the same definition, then

$$(VM\ {}_A\|_B\ CUST)\ {}_{A\cup B}\|_D\ CONTROL = STOP$$

because $CONTROL$ cannot do a $coin$ event.