◇ 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 for *SYSTEM* 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 *SYSTEM* 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 *SYSTEM* — 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*.

◇ Peterson's Algorithm ◇

We can define a model of Peterson's Algorithm in CSP. We define separate processes to represent the variables `flag1`, `flag2` and `turn`, and the two turnstile processes P1 and P2.

Events such as *p1setflag1*, *p2resetflag2* and so on are used to represent the interaction between the processes and the variable; for example, if *P1* and *FLAG1* synchronise on the event *p1setflag1*, that corresponds to the instruction `flag1 := true` being executed by P1.

The large number of events, and the large number of choices within some of the processes, make the definitions look quite complex. We will see later that it is possible to simplify them considerably.

ProBE can be used to explore the behaviour of the system and investigate mutual exclusion. Later, we will see how to write a specification of mutual exclusion which can be automatically checked by the FDR tool.

The definitions are in the file `peterson1.csp`. They could be modified to correspond to the programs for Coursework 1, and then ProBE provides an alternative way of tackling the question.

◇ 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

$$\mathit{coin} \to \mathit{choc} \to \mathit{Stop}$$

can be described by the labelled transitions:

$$(\mathit{coin} \to \mathit{choc} \to \mathit{Stop}) \xrightarrow{\mathit{coin}} (\mathit{choc} \to \mathit{Stop})$$
$$(\mathit{choc} \to \mathit{Stop}) \xrightarrow{\mathit{choc}} \mathit{Stop}$$

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

$$\mathit{DOOR} = \mathit{open} \rightarrow \mathit{close} \rightarrow \mathit{DOOR}$$

then because we have

$$(\mathit{open} \rightarrow \mathit{close} \rightarrow \mathit{DOOR}) \xrightarrow{\mathit{open}} (\mathit{close} \rightarrow \mathit{DOOR})$$

we also have

$$\mathit{DOOR} \xrightarrow{\mathit{open}} (\mathit{close} \rightarrow \mathit{DOOR}).$$

Then

$$(\mathit{close} \rightarrow \mathit{DOOR}) \xrightarrow{\mathit{close}} \mathit{DOOR}$$

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

Note: the operational semantics of CSP appears in 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

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

In

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

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

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

If we change *CUST* so that

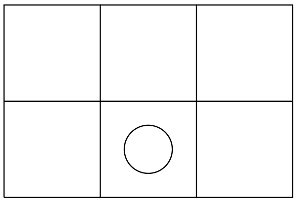
$$\begin{aligned} \alpha \mathit{CUST} &= \{\mathit{coin}, \mathit{choc}, \mathit{shout}\} = A \\ \mathit{CUST} &= \mathit{coin} \rightarrow \mathit{shout} \rightarrow \mathit{choc} \rightarrow \mathit{CUST} \end{aligned}$$

then

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

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:

$$
\begin{aligned}
\alpha LR &= \{\mathit{left}, \mathit{right}\}\\
\alpha UD &= \{\mathit{up}, \mathit{down}\}\\
LR &= \mathit{left} \to \mathit{right} \to LR \mid \mathit{right} \to \mathit{left} \to LR\\
UD &= \mathit{up} \to \mathit{down} \to UD
\end{aligned}
$$

and then

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

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):

$$
\begin{aligned}
R_{0,0} &= \mathit{right} \to R_{1,0} \mid \mathit{up} \to R_{0,1}\\
R_{0,1} &= \mathit{right} \to R_{1,1} \mid \mathit{down} \to R_{0,0}\\
&\ldots
\end{aligned}
$$

and then

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

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*, $\mathit{VM} \parallel \mathit{CUST}$ can do *beep* and *shout* in either order. If we define another process *CONTROL* with

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

then

$$(\mathit{VM}\ {}_{A}\|_{B}\ \mathit{CUST})\ {}_{A\cup B}\|_{C}\ \mathit{CONTROL}$$

behaves like the process P defined by

$$P = \mathit{coin} \rightarrow \mathit{beep} \rightarrow \mathit{shout} \rightarrow \mathit{choc} \rightarrow P.$$

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

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

and *CONTROL* has the same definition, then

$$(\mathit{VM}\ {}_{A}\|_{B}\ \mathit{CUST})\ {}_{A\cup B}\|_{D}\ \mathit{CONTROL} = Stop$$

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