◇ What's it all about? ◇

Concurrent systems — made up of independent but communicating components — are all around us. Familiar examples include:

- ◇ The network of bank cash machines
- ◇ The internet
- ◇ The network of "Switch" machines
- ◇ The components of a PC
- ◇ The telephone system

Understanding, designing and building concurrent systems is a major challenge for computer science. The problems involved are in a different league from the problems of sequential programming, and a systematic approach is essential.

This course aims to equip you with some of the theory, tools and techniques needed to understand and analyse concurrent systems, and to enable you to take a systematic approach to designing your own.

◇ CSP ◇

We will learn CSP (*C*ommunicating Sequential Processes), which is a theoretical notation or language for modelling concurrent systems. CSP is supported by various software tools which enable systems to be analysed and debugged, and we will use two in particular — ProBE and FDR — to assist in learning CSP and also to perform analyses of the systems which we consider.

CSP is a language which allows concurrent systems to be described in a more fundamental and abstract way. It was devised by C. A. R. Hoare. and developed at the University of Oxford during the 1980s.

CSP describes *processes* — objects or entities which exist independently, but may communicate. During its lifetime, a process may perform (engage in, do) various *events* or *actions*. These events are the visible parts of the behaviour of the process. In different systems, events correspond to different physical activities, but CSP treats them in a uniform way. As we will see, various styles of inter-process communication can be built up from the idea of events.

◇ Processes and Events ◇

Example: When describing a simple vending machine, which sells chocolates, we may be interested in the events $coin$, representing insertion of a coin into the machine, and $choc$, representing the appearance of a chocolate.

Example: To describe a more complex vending machine, which sells two sizes of chocolate and gives change, we might need the events in the set

$$\{in1p, in2p, small, large, out1p\}.$$

Notice that we make no distinction between events caused by the machine and events caused by the user of the machine. We will see later how to represent the machine and the user as separate processes.

The set of events which a process may use is called its *alphabet* or *interface*. The alphabet of a process P is written $\alpha(P)$.

Example: To describe a lecture as a process $LECT$, we might decide that

$$\alpha(LECT) = \{start, end, exercise\}.$$

◇ Events and Interfaces ◇

During the lifetime of a process, each event in the interface may occur once, many times, or not at all.

Which events we decide to include in the interface of a process depends on which aspects of its behaviour we are interested in. If we only care about the beginnings and ends of lectures, we might decide that

$$\alpha(LECT) = \{start, end\}.$$

For the moment, we will not normally define the interface of a process separately; it will be defined implicitly by the events which appear in the process definition. Later it will become important to specify interfaces in advance.

◇ Process Behaviour ◇

The simplest possible behaviour is to do nothing. The process which does nothing is written $STOP$.

The simplest way of constructing non-trivial processes is by means of *prefixing*, which allows events to occur in sequence. If P is a process and a is an event, then

$$a \rightarrow P$$

is a process which can perform the event a and then behave like the process P.

Example: Defining

$$VM = coin \rightarrow STOP$$

gives a vending machine which accepts a coin but then does nothing else.

$$VM = coin \rightarrow (choc \rightarrow STOP)$$

gives a machine which works, but only once.

$$VM = STOP$$

is a broken machine which cannot even accept a coin.

The expressions $P \rightarrow Q$ and $a \rightarrow b$, where P, Q are processes and a, b are events, are not allowed. Prefixing is *only* used with an event and a process. In expressions such as $a \rightarrow (b \rightarrow P)$, the brackets are usually omitted.

When we define a CSP process, we are only describing the relative order of events; nothing is said about timing. It is not possible for two or more events to occur simultaneously.

Example: If $LECT = start \rightarrow end \rightarrow STOP$ then we have captured the fact that a lecture begins and ends, but not the fact that a set time elapses in between.

◇ Recursion ◇

Using $STOP$ and prefixing we can only construct processes which must stop after a finite number of events. Very often we are interested in processes which run forever. To describe them we need recursive definitions.

Example: To describe a clock, we are only interested in the fact that it ticks, so we just need one event $tick$. We can define

$$CLOCK = tick \rightarrow CLOCK.$$

The process $CLOCK$ can perform the $tick$ event repeatedly. Substituting for $CLOCK$ on the right hand side of the definition gives

$$\begin{aligned} CLOCK &= tick \rightarrow tick \rightarrow CLOCK \\ &= tick \rightarrow tick \rightarrow tick \ldots \end{aligned}$$

Example: We can define a vending machine which does not stop after one transaction:

$$VM = coin \rightarrow choc \rightarrow VM$$

△ What is the difference between the recursive definitions we have seen so far, and a typical recursively defined function in C++ or ML?

In CSP we can define a collection of processes by mutual recursion, such as

$$\begin{aligned} VM &= coin \rightarrow VM\_PAID \\ VM\_PAID &= choc \rightarrow VM. \end{aligned}$$

Example: If we define

$$\begin{aligned} LECT &= start \rightarrow INLECT \\ INLECT &= exercise \rightarrow INLECT \end{aligned}$$

then we have a never-ending lecture in which you can't even go to sleep.

◇ Choice ◇

So far we have only defined processes which perform a single sequence of events, either just once or repeatedly. We also want to describe systems which may have alternative behaviours, perhaps determined by their environment.

If P, Q are processes and x, y are distinct events, then

$$x \rightarrow P \mid y \rightarrow Q$$

is a process which can *either* do the event x and then behave like P, *or* do the event y and then behave like Q.

This is pronounced "x then P choice y then Q", or sometimes "x then P or y then Q"

Example: A ticket machine sells tickets to Staines, for one pound, or Ashford, for two pounds. We can describe it as a process $TICKET$, with interface $\{staines, ashford, pound, ticket\}$.

$$\begin{aligned} &TICKET = \\ &\quad staines \rightarrow pound \rightarrow ticket \rightarrow STOP \\ &\mid\ ashford \rightarrow pound \rightarrow pound \rightarrow ticket \rightarrow STOP \end{aligned}$$

We can combine choice with recursion, for example to define a more useful ticket machine:

$$\begin{aligned}&\mathit{TICKETS} =\\&\quad \mathit{staines} \to \mathit{pound} \to \mathit{ticket} \to \mathit{TICKETS}\\&\mid \mathit{ashford} \to \mathit{pound} \to \mathit{pound} \to \mathit{ticket} \to \mathit{TICKETS}\end{aligned}$$

Some choices in a recursive process may lead to termination:

$$\begin{aligned}&\mathit{TICKETS} =\\&\quad \mathit{staines} \to \mathit{pound} \to \mathit{ticket} \to \mathit{TICKETS}\\&\mid \mathit{ashford} \to \mathit{pound} \to \mathit{pound} \to \mathit{ticket} \to \mathit{STOP}\end{aligned}$$

We can also define choices with more than two alternatives:

$$x \to P \mid y \to Q \mid \ldots \mid z \to R.$$

Note that we cannot write $P \mid Q$ for processes P and Q. We can only use $\mid$ in conjunction with a collection of distinct prefixes. This is to ensure that situations such as $x \to P \mid x \to Q$ cannot arise.

Example: Suppose the ticket machine needs to be turned on before use, and can be turned off after any transaction.

$$\begin{aligned}&\mathit{MACHINE} = \mathit{on} \to \mathit{TICKETS}\\&\mathit{TICKETS} =\\&\quad \mathit{staines} \to \mathit{pound} \to \mathit{ticket} \to \mathit{TICKETS}\\&\mid \mathit{ashford} \to \mathit{pound} \to \mathit{pound} \to \mathit{ticket} \to \mathit{TICKETS}\\&\mid \mathit{off} \to \mathit{MACHINE}\end{aligned}$$

Suppose we want to model a lecture as a process LECT with alphabet $\{\mathit{start}, \mathit{end}, \mathit{exercise}\}$, as before.

△ Define LECT so that a lecture starts, may contain any number of exercises, and may eventually end.

We can model the career of an undergraduate as a process $\mathit{STUDENT}$ with alphabet

$$\{\mathit{year}1, \mathit{year}2, \mathit{year}3, \mathit{pass}, \mathit{graduate}\}.$$

A simple definition of an ideal degree programme is

$$\begin{aligned}\mathit{STUDENT} \;=\; &\mathit{year}1 \to \mathit{pass} \to \mathit{year}2 \to \mathit{pass} \to\\&\mathit{year}3 \to \mathit{pass} \to \mathit{graduate} \to \mathit{STOP}.\end{aligned}$$

△ Add an event fail to the alphabet of $\mathit{STUDENT}$, and modify the definition so that a student can fail at any point and repeat a year.

When discussing choice, we have ignored the question of how a choice is made — we have simply listed alternative possibilities. Later we will be able to distinguish between choices made by a process and choices made by the environment in which it is placed.

◇ Menu Choice ◇

There is another notation for choice, known as *menu choice*. If A is a set of events, and for each event x in A there is a process $P(x)$, then

$$x : A \to P(x)$$

(pronounced "x from A then P of x") is a process which can do any of the events in A and then become the appropriate $P(x)$.

Example: Suppose we define a collection of processes with alphabet $\mathbb{N}$:

$$\begin{aligned}\mathit{COUNTDOWN}_0 &= 0 \to \mathit{STOP}\\\mathit{COUNTDOWN}_1 &= 1 \to \mathit{COUNTDOWN}_0\\&\vdots\\\mathit{COUNTDOWN}_n &= n \to \mathit{COUNTDOWN}_{n-1}\\&\vdots\end{aligned}$$

we can then define

$$\mathit{COUNTDOWN} = x : \mathbb{N} \to \mathit{COUNTDOWN}_x$$

which allows the starting point of the countdown to be chosen.

Think of this definition as

$$x : \mathbb{N} \to P(x)$$

where, for each $x \in \mathbb{N}$, $P(x) = \mathit{COUNTDOWN}_x$.

Menu choice subsumes all the operations we have seen so far. The choice

$$a_1 \to P_1 \mid a_2 \to P_2 \mid \ldots \mid a_n \to P_n$$

can be written

$$x : A \to P(x)$$

where $A = \{a_1, \ldots, a_n\}$ and for each i, $P(a_i) = P_i$.

The prefixing construction

$$a \to P$$

can be written

$$x : A \to P(x)$$

where $A = \{a\}$ and $P(a) = P$. STOP can be written

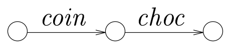
$$x : \{\} \to P(x)$$

where no definition for $P(x)$ needs to be supplied.

It will sometimes be useful to think of STOP, prefixing and choice in this way, as special cases of menu choice.

◇ Transition Diagrams ◇

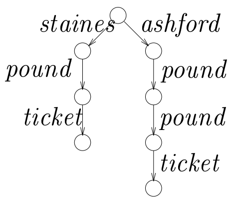
It is sometimes useful to view processes pictorially. For example, the process $coin \rightarrow choc \rightarrow STOP$ can be represented by this diagram:



Such diagrams are called *state transition diagrams* or just *transition diagrams*. Each circle represents a state of the process; in this example, the states are $coin \rightarrow choc \rightarrow STOP$, $choc \rightarrow STOP$, and $STOP$. Each arrow represents an event which the process may do when in a certain state.

Choices are represented by multiple arrows (with different labels) from a single state.

Example: The transition diagram for the process $TICKET$ is

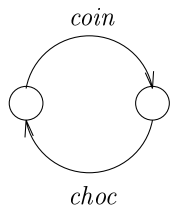


A state with no arrows leaving it corresponds to $STOP$.

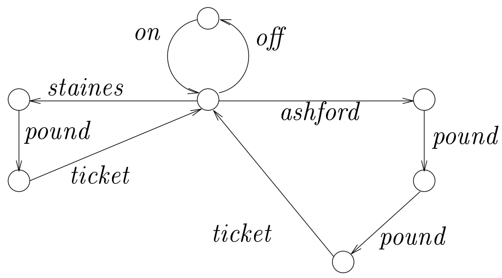
The transition diagram for a recursive process is cyclic. For example,

$$VM = coin \rightarrow choc \rightarrow VM$$

has this diagram:

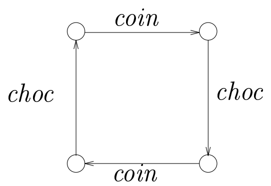


A larger example: the process $MACHINE$.



Problems with transition diagrams include:

- ◇ Very large diagrams are hard to draw (and some processes have an infinite number of states, which is even worse).
- ◇ Different diagrams can be drawn for the same process. For example, VM also corresponds to the following diagram:



Later we will introduce a mathematical theory of process equivalence, with a collection of algebraic laws.

However, it is still useful to talk about process states and transitions, as a way of defining process operators.