◇ Credits ◇

These slides, together with accompanying practical coursework, were originally produced by Simon Gay for an introductory course on CSP taught in the Department of Computer Science, at Royal Holloway, University of London. Some adaptations have been introduced by Steve Schneider.

This material has been made publically available to accompany Chapters 1–8 of the book

Concurrent and Real Time Systems: the CSP approach
by Steve Schneider
published by John Wiley.

Copies of these slides and the practical coursework, may be found at the book's web site:

`http://www.cs.rhbnc.ac.uk/concurrency`

If you have any comments or questions about any of this material, please contact

`S.Gay@rhbnc.ac.uk`
`S.Schneider@rhbnc.ac.uk`

◇ 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

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

$\diamond$ Events and Interfaces $\diamond$

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.

$\diamond$ Process Behaviour $\diamond$

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

$$VM = coin \to STOP$$

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

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

gives a machine which works, but only once.

$$VM = STOP$$

is a broken machine which cannot even accept a coin.

The expressions $P \to Q$ and $a \to b$, 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 $a \to (b \to P)$, 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 $LECT = start \to end \to STOP$ 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.

$\diamond$ Choice $\diamond$

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 $\{staines, ashford, pound, ticket\}$.

$$\begin{array}{l} TICKET = \\ \quad staines \rightarrow pound \rightarrow ticket \rightarrow STOP \\ \mid \; ashford \rightarrow pound \rightarrow pound \rightarrow ticket \rightarrow STOP \end{array}$$

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

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

Some choices in a recursive process may lead to termination:

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

We can also define choices with more than two alternatives:

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

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 $x \rightarrow P \mid x \rightarrow Q$ cannot arise.

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

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

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

$$\{year1, year2, year3, pass, graduate\}.$$

A simple definition of an ideal degree programme is

$$\begin{aligned} STUDENT \;=\; & year1 \rightarrow pass \rightarrow year2 \rightarrow pass \rightarrow \\ & year3 \rightarrow pass \rightarrow graduate \rightarrow STOP. \end{aligned}$$

△ Add an event $fail$ to the alphabet of $STUDENT$, 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.

$\diamond$ Menu Choice $\diamond$

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 $\mathbb{N}$:

$$\begin{aligned} COUNTDOWN_0 &= 0 \to STOP \\ COUNTDOWN_1 &= 1 \to COUNTDOWN_0 \\ &\vdots \\ COUNTDOWN_n &= n \to COUNTDOWN_{n-1} \\ &\vdots \end{aligned}$$

we can then define

$$COUNTDOWN = x : \mathbb{N} \to COUNTDOWN_x$$

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 $x \in \mathbb{N}$, $P(x) = COUNTDOWN_x$.

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

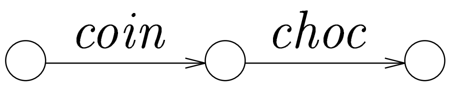
$$x : \{\} \to P(x)$$

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 ◇

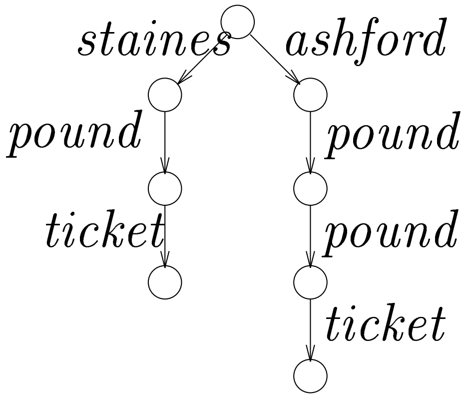
It is sometimes useful to view processes pictorially. For example, the process $coin \rightarrow choc \rightarrow STOP$ can be represented by this diagram:



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

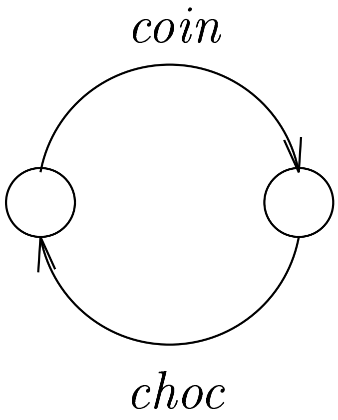


A state with no arrows leaving it corresponds to $STOP$.

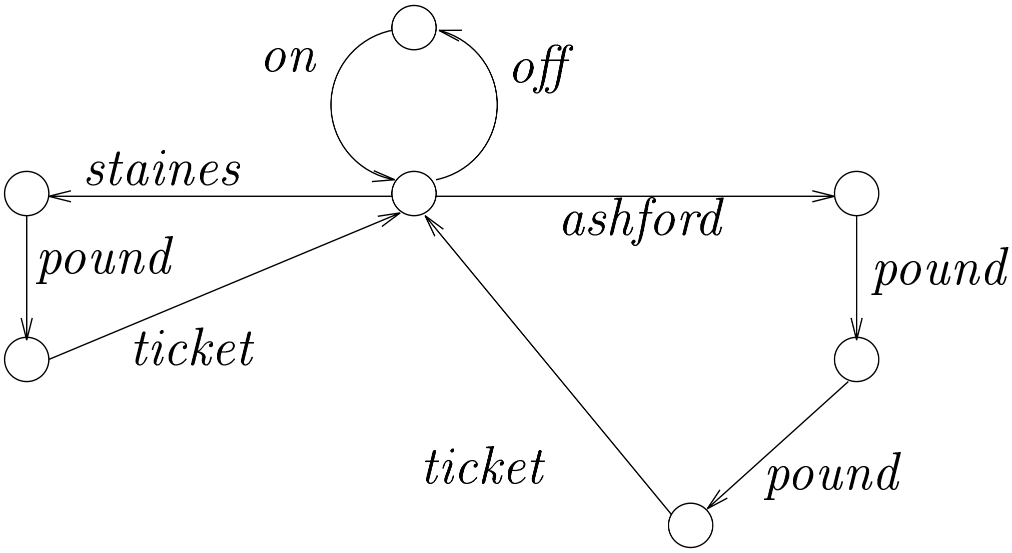
The transition diagram for a recursive process is cyclic. For example,

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

has this diagram:

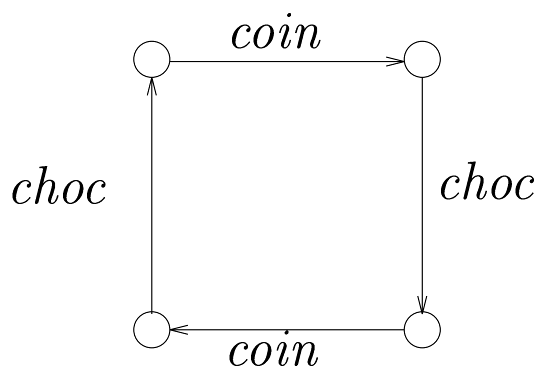


A larger example: the process $MACHINE$.



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.

◇ 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:

$$\begin{aligned} VM &= coin \to (choc \to STOP \mid toffee \to STOP) \\ CUST &= coin \to choc \to STOP \end{aligned}$$

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

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 $VM\ {}_A\|_A\ CUST$.

$$\begin{array}{c}
VM\ {}_A\|_A\ CUST \\
\Big\downarrow{\scriptstyle coin} \\
(choc \rightarrow STOP \mid toffee \rightarrow STOP)\ {}_A\|_A\ choc \rightarrow STOP \\
\Big\downarrow{\scriptstyle choc} \\
STOP\ {}_A\|_A\ STOP
\end{array}$$

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.

$\diamond$ More Processes $\diamond$

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.

$$TUTOR = fail \to shout \to TUTOR$$

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

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.

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

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

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

$$\begin{array}{rcl} PASS & = & ashford \to pound \to \\ & & \quad (ticket \to PASS \\ & & \quad \mid pound \to ticket \to PASS) \\ & \mid & feltham \to pound \to ticket \to STOP \\ TICKETS & = & staines \to pound \to \\ & & \quad ticket \to TICKETS \\ & & \Box\ ashford \to pound \to pound \to \\ & & \quad ticket \to TICKETS \end{array}$$

$\triangle$ 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.

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

$\diamond$ Student and Parent $\diamond$

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 $prize$.

$\triangle$ 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

$$coin \rightarrow choc \rightarrow STOP$$

can be described by the labelled transitions:

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

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

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

There are no hypotheses, which means that we always know that $(a \rightarrow P) \xrightarrow{a} P$. 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.

$$
\begin{array}{c}
VM\ {}_A\|_B\ CUST \\
\Big\downarrow coin \\
beep \to choc \to VM\ {}_A\|_B\ choc \to eat \to CUST \\
\Big\downarrow beep \\
choc \to VM\ {}_A\|_B\ choc \to eat \to CUST \\
\Big\downarrow choc \\
VM\ {}_A\|_B\ eat \to CUST \\
\Big\downarrow eat \\
VM\ {}_A\|_B\ CUST
\end{array}
$$

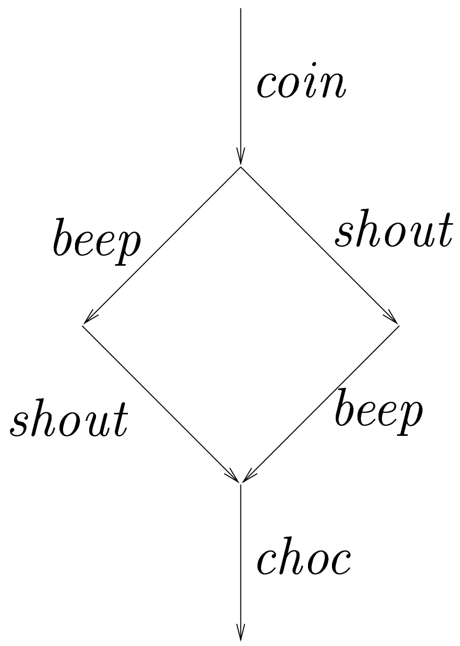
If we change $CUST$ so that

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

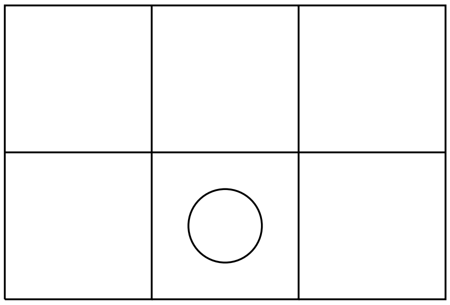
then

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

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:

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

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 $CUST$, $VM \parallel CUST$ 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.

◇ Traces ◇

A *trace* of a process is a finite sequence of events, representing the behaviour of the process up to a certain point in time. Traces are written as comma-separated sequences of events, enclosed in angle brackets: for example, $\langle coin, choc, coin\rangle$. This is a trace of the recursive version of VM.

Example: $\langle open, close\rangle$ and $\langle open, close, open\rangle$ are traces of $DOOR$.

$(DOOR = open \rightarrow close \rightarrow DOOR)$

Example: $\langle staines, pound\rangle$ and $\langle ashford, pound\rangle$ are traces of $TICKET$, and also of $TICKETS$.

We will only consider *finite* traces.

The empty trace, containing no events, is written $\langle\rangle$ and pronounced “empty” or “nil”. It is a trace of every process, corresponding to an observation when no event has yet happened.

If a process is defined without recursion, then it has a bound on the length of its traces. For example, if

$$PHONE = ring \rightarrow answer \rightarrow STOP$$

then the only traces of $PHONE$ are $\langle\rangle$, $\langle ring\rangle$ and $\langle ring, answer\rangle$.

A recursive process, which can keep performing events forever, can have an infinite set of traces. For example, if

$$CLOCK = tick \rightarrow CLOCK$$

then the traces of $CLOCK$ are

$$\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \langle tick, tick, tick\rangle, \ldots$$

It is important to be clear about the fact that we are interested in potentially *infinite* sets of *finite* traces.

◇ Operations on Traces ◇

We will use various operations on traces, and a number of facts or laws about them. Most of the laws are rather obvious.

◇ Concatenation ◇

The first operation is *concatenation*, also called *catenation*. It joins traces together into longer traces:

$$\langle a_1, \ldots, a_m\rangle ^\frown \langle b_1, \ldots, b_n\rangle = \langle a_1, \ldots, a_m, b_1, \ldots, b_n\rangle.$$

Example: $\langle coin, choc\rangle ^\frown \langle choc\rangle = \langle coin, choc, choc\rangle$.
Concatenation is associative, and the empty trace is a unit, i.e.

$$tr_0 ^\frown (tr_1 ^\frown tr_2) = (tr_0 ^\frown tr_1) ^\frown tr_2$$
$$\langle\rangle ^\frown tr = tr = tr ^\frown \langle\rangle$$

The following laws are useful:

$$tr_0 \frown tr_1 = tr_0 \frown tr_2 \text{ if and only if } tr_1 = tr_2$$
$$tr_0 \frown tr_2 = tr_1 \frown tr_2 \text{ if and only if } tr_0 = tr_1$$
$$tr_0 \frown tr_1 = \langle\rangle \text{ if and only if } tr_0 = \langle\rangle \text{ and } tr_1 = \langle\rangle$$

If n is a positive integer, then tr^n is defined to be n copies of the trace tr concatenated together. tr^n can be defined recursively by

$$\begin{aligned} tr^0 &= \langle\rangle \\ tr^{n+1} &= tr \frown tr^n. \end{aligned}$$

◇ Functions on Traces ◇

Suppose f is a function which maps traces to traces. f is said to be *strict* if $f(\langle\rangle) = \langle\rangle$, and *distributive* if $f(tr_0 \frown tr_1) = f(tr_0) \frown f(tr_1)$.

In fact, any distributive function is strict: if f is distributive then

$$\begin{aligned} f(tr) \frown \langle\rangle = f(tr) &= f(tr \frown \langle\rangle) \\ &= f(tr) \frown f(\langle\rangle) \end{aligned}$$

and so $f(\langle\rangle) = \langle\rangle$.

If f is distributive then its action on traces can be put together from its action on single-event traces:

$$\begin{aligned} f(\langle a_1, \ldots, a_n\rangle) &= f(\langle a_1\rangle \frown \ldots \frown \langle a_n\rangle) \\ &= f(\langle a_1\rangle) \frown \ldots \frown f(\langle a_n\rangle). \end{aligned}$$

$\diamond$ Restriction $\diamond$

The expression $tr \upharpoonright A$ denotes the trace tr when *restricted* to events in the set A. $tr \upharpoonright A$ consists of tr with all events outside A omitted.

Example:

$\langle start, exercise, exercise, end\rangle \upharpoonright \{start, end\}$
$= \langle start, end\rangle$.

$\langle start, exercise, exercise, end\rangle \upharpoonright \{start, exercise\}$
$= \langle start, exercise, exercise\rangle$.

Restriction is distributive and therefore strict:

$$\begin{aligned} \langle\rangle \upharpoonright A &= \langle\rangle \\ (tr_0 \frown tr_1) \upharpoonright A &= (tr_0 \upharpoonright A) \frown (tr_1 \upharpoonright A). \end{aligned}$$

The effect of restriction on single-event traces is clear:

$$\begin{aligned} \langle x\rangle \upharpoonright A &= \langle x\rangle \text{ if } x \in A \\ \langle x\rangle \upharpoonright A &= \langle\rangle \text{ if } x \notin A \end{aligned}$$

Two other facts:

$$\begin{aligned} tr \upharpoonright \{\} &= \langle\rangle \\ (tr \upharpoonright A) \upharpoonright B &= tr \upharpoonright (A \cap B) \end{aligned}$$

$\diamond$ Head and Tail $\diamond$

If tr is a non-empty trace, its first event is denoted tr_0 and the trace consisting of all events after the first is denoted tr'.

Neither $\langle\rangle_0$ nor $\langle\rangle'$ is defined.

Example: $\langle coin, choc\rangle_0 = coin$.

$\langle coin, choc\rangle' = \langle choc\rangle$.

A few facts:

$$\begin{aligned}(\langle x\rangle \frown tr)_0 &= x \\ (\langle x\rangle \frown tr)' &= tr \\ tr &= \langle tr_0\rangle \frown tr'\end{aligned}$$

$\diamond$ Star $\diamond$

If A is a set of events, the set A^* is the set of all finite traces, including $\langle\rangle$, containing events from A.

Example:

$\{a, b\}^* = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, a\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle b, b\rangle, \ldots\}$

$\diamond$ Ordering $\diamond$

A trace tr_0 is a *prefix* of a trace tr_1 if there is some extension tr_2 of tr_0 such that $tr_0 \frown tr_2 = tr_1$. We then write $tr_0 \leqslant tr_1$.

Example:

$$\begin{aligned} \langle a, b, c \rangle &\leqslant \langle a, b, c, d \rangle \\ \langle \rangle &\leqslant \langle a, b \rangle \end{aligned}$$

$\diamond$ Length $\diamond$

The length of the trace tr is denoted $\#tr$.

Example: $\#\langle a, b \rangle = 2$, $\#\langle \rangle = 0$.

$\diamond$ Traces of a Process $\diamond$

In general a process has many different possible behaviours, and we do not know in advance which traces will be generated by a particular execution. However, we can determine in advance the set of all possible traces of a process P. This set is written $traces(P)$.

Examples: $traces(STOP) = \{\langle \rangle\}$.

$traces(coin \rightarrow STOP) = \{\langle \rangle, \langle coin \rangle\}$.

$$\begin{aligned} traces(CLOCK) &= \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \ldots\} \\ &= \{tick\}^* \end{aligned}$$

We can now systematically write down definitions of $traces(P)$ for processes P constructed from the operators we have seen so far. We already know the definition for $STOP$:

$$traces(STOP) = \{\langle\rangle\}.$$

$traces(a \to P)$ is constructed from $traces(P)$ by the addition of a as an initial event:

$$traces(a \to P) = \{\langle\rangle\} \cup \{\langle a\rangle ^\frown tr \mid tr \in traces(P)\}.$$

Notice the addition of the trace $\langle\rangle$, which must always be a trace of any process.

The definition of $traces(a \to P \mid b \to Q)$ is similar, taking account of the two possible first events:

$$\begin{aligned} traces(a \to P \mid b \to Q) &= \{\langle\rangle\} \\ &\cup \{\langle a\rangle ^\frown tr \mid tr \in traces(P)\} \\ &\cup \{\langle b\rangle ^\frown tr \mid tr \in traces(Q)\}. \end{aligned}$$

Also similarly, we can give a general definition of $traces(x : A \to P(x))$.

$$\begin{aligned} & traces(x : A \to P(x)) \\ &\quad = \{\langle\rangle\} \\ &\quad \cup \{\langle a\rangle ^\frown tr \mid a \in A, tr \in traces(P(a))\}. \end{aligned}$$

A few facts about $traces$:

$\langle\rangle \in traces(P)$, for any P.

If $tr_0 \frown tr_1 \in traces(P)$ then $tr_0 \in traces(P)$.

$traces(P) \subseteq (\alpha(P))^*$.

Describing the set of traces of a recursive process is more complicated. Suppose we have the definition

$$X = F(X)$$

where $F(X)$ is a guarded expression. Guardedness means that we know at least the possible first events of $F(X)$. In fact, they are the same as the possible first events of $F(STOP)$, whatever X is.

Example: If $X = a \rightarrow X$ then we know that X can do a first, and this is the same first event as in $a \rightarrow STOP$.

Depending on the form of $F(X)$, we may know more than just the first event.

Example: If $X = a \rightarrow b \rightarrow X \mid c \rightarrow X$ we know that X can either do a then b, or c, so we know that $\langle a, b\rangle$ and $\langle c\rangle$ are traces of X. They are also traces of $a \rightarrow b \rightarrow STOP \mid c \rightarrow STOP$.

We can discover some traces of X by looking at $F(STOP)$. For the traces corresponding to running through F twice, we need to look at $F(F(STOP))$.

Example: If $X = a \to X$ we also have

$$X = a \to a \to X$$

so $\langle a, a \rangle$ is a trace of X.

If $X = a \to b \to X \mid c \to X$ we also have

$$\begin{aligned} X \;=\; & a \to b \to (a \to b \to X \mid c \to X) \\ & \mid c \to (a \to b \to X \mid c \to X) \end{aligned}$$

So $\langle a, b, a \rangle$, $\langle a, b, c \rangle$, $\langle c, a, b \rangle$ etc. are traces of X.

In general we can define iteration of F:

$$\begin{aligned} F^0(X) &= X \\ F^{n+1}(X) &= F(F^n(X)) \end{aligned}$$

and then, for $X = F(X)$, we have

$$\begin{aligned} traces(X) &= \bigcup_{n \geqslant 0} traces(F^n(STOP)) \\ &= traces(STOP) \cup traces(F(STOP)) \\ &\cup \; traces(F(F(STOP))) \cup \ldots \end{aligned}$$

Writing down the set of traces of a recursive process in a compact form is a little challenging. For example, if $X = a \to b \to X$, then $traces(X)$ contains not only $\langle a, b \rangle$, $\langle a, b, a, b \rangle$, $\langle a, b \rangle^3$ and so on, but also the intermediate traces ending in a. One way to describe $traces(X)$ is:

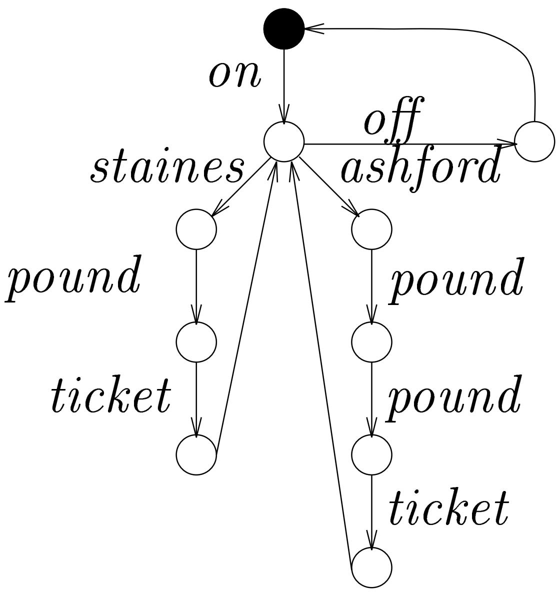
$$traces(X) = \{ tr \mid \text{for some } n,\ tr \leqslant \langle a, b \rangle^n \}$$

◇ Traces and Diagrams ◇

There is a connection between the transition diagram of a process, and its traces. For example, recall the process $TICKETS$ defined by

$$\begin{aligned}
MACHINE &= on \rightarrow TICKETS \\
TICKETS &= staines \rightarrow pound \rightarrow ticket \\
&\qquad\qquad \rightarrow TICKETS \\
&\mid ashford \rightarrow pound \rightarrow pound \rightarrow ticket \\
&\qquad\qquad \rightarrow TICKETS \\
&\mid off \rightarrow MACHINE
\end{aligned}$$

and its transition diagram:



For any path through the diagram, starting from the black state, there is a trace consisting of the sequence of labels on the path. $traces(TICKETS)$ is the set of traces corresponding to all these paths.

◇ Traces and Transitions ◇

The operational semantics of CSP allows us to unwind the behaviour of a process, one event at a time. Looking at the traces of a process gives us an overall view. Since the traces can be extracted from a transition diagram, and labelled transitions are supposed to capture the same information as the diagrams, we should also be able to write down a relationship between a process' traces and its labelled transitions. Here it is:

$$
\begin{aligned}
&traces(P) = \{\langle\rangle\} \\
&\qquad \cup\ \{\langle a\rangle \frown tr \mid P \xrightarrow{a} Q, tr \in traces(Q)\}.
\end{aligned}
$$

Later we will be defining new CSP operators, by means of labelled transition rules. We will use this relationship between transitions and traces to calculate the traces of processes defined in terms of the new operators.

◇ Exercises ◇

△ Write down $traces(TICKET)$, where $TICKET$ is defined as before by

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

◇ Exercises ◇

△ Define a process P such that

$$traces(P) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle b, c\rangle\}.$$

△ Define a process P such that $\langle a, b, c\rangle$ and $\langle a, b, a\rangle$ are both traces of P.

△ Is there a process P such that

$$traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c, d\rangle\}?$$

◇ Traces for Concurrency ◇

$$\begin{aligned} traces(P \;{}_A\|_B\; Q) \;=\; \{tr \mid\ & tr \in (A \cup B)^* \\ & \text{and } tr\restriction A \in traces(P) \\ & \text{and } tr\restriction B \in traces(Q)\} \end{aligned}$$

If $A = B$, this definition reduces to

$$\begin{aligned} traces(P \;{}_A\|_A\; Q) \;=\; \{tr \mid\ & tr \in A^* \\ & \text{and } tr\restriction A \in traces(P) \\ & \text{and } tr\restriction A \in traces(Q)\} \end{aligned}$$

i.e. $traces(P \;{}_A\|_A\; Q) = traces(P) \cap traces(Q)$, because if $tr \in A^*$ then $tr\restriction A = tr$. This fits in with the earlier discussion of concurrency with the same alphabet.

If $A \cap B = \{\}$ then every event in a possible trace of $P \;{}_A\|_B\; Q$ is either an event from A or an event from B. In a trace tr of $P \;{}_A\|_B\; Q$, the events from A (i.e. $tr\restriction A$) must form a trace of P, and similarly the events from B must form a trace of Q. Any pair of traces, one from P and one from Q, can be *interleaved* to form a trace of $P \;{}_A\|_B\; Q$.

Example: $\langle left, right, right\rangle$ is a trace of LR and $\langle up, down\rangle$ is a trace of UD. So

$$\langle left, up, down, right, right\rangle$$

is a trace of $LR \parallel UD$.

In general, a trace of P and a trace of Q can be used to form a trace of $P \ {}_A\|_B \ Q$ as long as the events in $A \cap B$ appear in the same order in both traces.

Example: $\langle coin, beep, choc \rangle$ is a trace of VM and $\langle coin, shout, choc \rangle$ is a trace of $CUST$. The events common to both alphabets (i.e. $coin$ and $choc$) appear in the same order in both traces.

$\langle coin, beep, shout, choc \rangle$ and $\langle coin, shout, beep, choc \rangle$ are both traces of $VM \parallel CUST$.

$\diamond$ Trace Equivalence $\diamond$

We have spoken vaguely of processes being equivalent to each other — for example, a process which can do no events is equivalent to $STOP$. In CSP there are in fact several notions of process equivalence, each of which is useful in different situations. The first is *trace equivalence*, denoted by $=_T$, and defined by

$$P =_T Q$$
$$\text{if and only if}$$
$$traces(P) = traces(Q)$$

Two processes are trace equivalent if they have the same observable behaviour, as measured by $traces$.

Example: Consider the process

$$a \rightarrow STOP \;{}_{\{a,b\}}\|_{\{a,b\}}\; b \rightarrow STOP.$$

The definition of $traces$ for a parallel combination of processes gives
$traces(a \rightarrow STOP \;{}_{\{a,b\}}\|_{\{a,b\}}\; b \rightarrow STOP)$
$= \{tr \in \{a,b\}^* \mid tr \upharpoonright \{a,b\} \in traces(a \rightarrow STOP)$
and $tr \upharpoonright \{a,b\} \in traces(b \rightarrow STOP)\}$.

i.e. $traces(a \rightarrow STOP \;{}_{\{a,b\}}\|_{\{a,b\}}\; b \rightarrow STOP)$
$= traces(a \rightarrow STOP) \cap traces(b \rightarrow STOP)$.

Because

$$traces(a \rightarrow STOP) = \{\langle\rangle, \langle a \rangle\}$$

and

$$traces(b \rightarrow STOP) = \{\langle\rangle, \langle b \rangle\}$$

we get

$$traces(a \rightarrow STOP \ {}_{\{a,b\}}\|_{\{a,b\}} \ b \rightarrow STOP) = \{\langle\rangle\}.$$

Therefore

$$a \rightarrow STOP \ {}_{\{a,b\}}\|_{\{a,b\}} \ b \rightarrow STOP =_T STOP.$$

◇ Refinement and Specification ◇

The *refinement* relation $\sqsubseteq_T$ on processes is defined by

$$P \sqsubseteq_T Q$$
if and only if
$$traces(Q) \subseteq traces(P)$$

$P \sqsubseteq_T Q$ is pronounced "P is refined by Q". The subscript T indicates that we are working with traces — later we will see other forms of refinement.

P is refined by Q if Q exhibits at most the behaviour exhibited by P — possibly less.

Example:

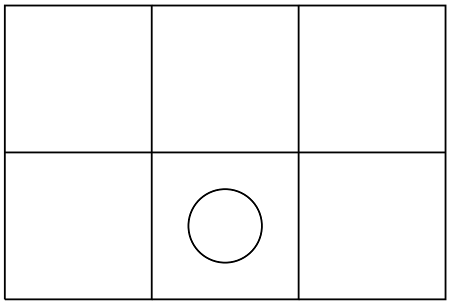
$$a \rightarrow b \rightarrow STOP \sqsubseteq_T a \rightarrow STOP$$

Example: For any process P, $P \sqsubseteq_T STOP$.

The main use of refinement is in specification. If we think of P as defining a range of permissible behaviour, then the statement $P \sqsubseteq_T Q$ can be read as the specification that Q's behaviour must stay within this range.

◇ Example ◇

Recall the example of a counter moving on a board.



$$
\begin{aligned}
LR &= \mathit{left} \to \mathit{right} \to LR \;\Box\; \mathit{right} \to \mathit{left} \to LR \\
UD &= \mathit{up} \to \mathit{down} \to UD
\end{aligned}
$$

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

We can now interpret $SPEC$ as a specification for processes which might describe movements of the counter. Because $SPEC$ describes exactly the behaviours which correspond to staying on the board, the specification

$$SPEC \sqsubseteq_T P$$

specifies that P must describe movement within the board — possibly restricted movement.

For example,

$$SPEC \sqsubseteq_T \mathit{left} \to \mathit{up} \to STOP$$

which we can check by writing down all the traces of the process on the right and showing that they are all traces of $SPEC$.

The specification

$$SPEC \sqsubseteq_T P$$

limits what P can do, but does not require it to do anything. For example,

$$SPEC \sqsubseteq_T STOP.$$

Specifications which simply restrict behaviour without requiring any particular behaviour are known as *safety specifications*. They specify that nothing bad can happen, without specifying that anything good must happen. $STOP$ satisfies any safety specification — doing nothing is always safe.

All specifications which can be expressed using trace refinement are safety specifications.

Specifications which require something positive to happen are called *liveness specifications*. We will see later how they can be expressed in CSP.

Example: If we define P by

$$P = \mathit{left} \rightarrow \mathit{left} \rightarrow STOP$$

then we do not have $SPEC \sqsubseteq_T P$ because

$$\begin{array}{l} \langle \mathit{left}, \mathit{left} \rangle \ \in \ \mathit{traces}(P) \\ \langle \mathit{left}, \mathit{left} \rangle \ \notin \ \mathit{traces}(SPEC). \end{array}$$

◇ The Level Crossing ◇

As an example of writing a specification in CSP, we will look at a railway level crossing. One road and one railway line cross each other, and as usual there is a gate which can be lowered to prevent cars crossing the railway. If the gate is raised, then cars can freely cross the track. Trains can cross the road regardless of whether the gate is up or down.

We will consider the obvious safety property for the level crossing, which is:

There should never be a train and a car on the crossing at the same time.

Of course there are many other properties which we might like to specify, for example a liveness property:

Whenever a car approaches the crossing, it should eventually be able to cross.

but for the moment we will stick to safety.

We will use the following events to represent the interesting aspects of the behaviour of the system.

$car.approach$, $car.enter$, $car.leave$, $train.approach$, $train.enter$, $train.leave$, $gate.lower$, $gate.raise$ $crash$

The processes $CARS$ and $TRAINS$ supply streams of cars and trains.

$$
\begin{aligned}
CARS &= car.approach \to car.enter \to \\
&\quad car.leave \to CARS \\
TRAINS &= train.approach \to train.enter \to \\
&\quad train.leave \to TRAINS
\end{aligned}
$$

The following processes model the behaviour of the crossing. This is a complete description of all possibilities, including a car and a train simultaneously using the crossing. Later we will add a control process which uses the gate to restrict access by cars.

CR models the crossing with cars and trains. The processes C, T, CT model the crossing when there is a car, train or both present, respectively:

$$
\begin{aligned}
CR &= car.approach \to car.enter \to C \\
&\quad \Box\ train.approach \to train.enter \to T \\[1em]
C &= car.leave \to CR \\
&\quad \Box\ train.approach \to train.enter \to CT \\[1em]
T &= train.leave \to CR \\
&\quad \Box\ car.approach \to car.enter \to CT \\[1em]
CT &= crash \to STOP
\end{aligned}
$$

Cars can only enter the crossing when the gate is up:

$$\begin{aligned} GATE \;=\; & gate.lower \rightarrow gate.raise \rightarrow GATE \\ & \Box\; car.enter \rightarrow GATE \end{aligned}$$

Defining some sets of events:

$$\begin{aligned} E_T &= \{train.approach, train.enter, train.leave\} \\ E_C &= \{car.approach, car.enter, car.leave\} \\ E_{GC} &= \{gate.raise, gate.lower, car.enter\} \\ E_X &= \{crash\} \\ E_S &= E_T \cup E_C \cup E_{GC} \cup E_X \end{aligned}$$

allows us to define the whole system as

$$\begin{aligned} SYSTEM \;=\; & ((CR \;{}_{E_T \cup E_C \cup E_X}\|_{E_{GC}}\; GATE) \\ & \quad {}_{E_S}\|_{E_C}\; CARS) \;{}_{E_S}\|_{E_T}\; TRAINS. \end{aligned}$$

To specify that no crashes occur, we need a process $SPEC$ which can do any event except for $crash$.

$$\begin{aligned} SPEC \;=\; & train?x : \{approach, enter, leave\} \rightarrow SPEC \\ & \Box\; car?x : \{approach, enter, leave\} \rightarrow SPEC \\ & \Box\; gate?x : \{raise, lower\} \rightarrow SPEC \end{aligned}$$

In general, RUN_A is the process which can repeatedly do events from the set A:

$$RUN_A \;=\; x : A \rightarrow Run_A$$

so $SPEC = RUN_{E_C \cup E_T \cup E_{GC}}$.

The requirement that the crossing satisfies this specification is expressed by

$$SPEC \sqsubseteq_T SYSTEM.$$

It is possible to use the FDR tool to check trace refinement, and this is the easiest way to show that the specification is not satisfied (not surprisingly, as we haven't imposed any restrictions on when the gate can be raised or lowered).

Now we will define a process $CONTROL$ which, when placed in parallel with $SYSTEM$, will constrain the behaviour so that whenever a train approaches the gate must be lowered. This will be achieved by making $CONTROL$ and $SYSTEM$ synchronise on certain events. We hope that the result will be a system which satisfies the safety specification.

$$\begin{array}{rcl} CONTROL & = & (train.approach \to gate.lower \to \\ & & train.enter \to train.leave \to \\ & & gate.raise \to CONTROL) \\ & \Box & (car.approach \to car.enter \to \\ & & car.leave \to CONTROL) \end{array}$$

$$\begin{array}{l} SAFE\_SYSTEM = \\ \quad SYSTEM \; {}_{E_S}\|_{E_T \cup E_C \cup E_{GC}} \; CONTROL \end{array}$$

Again, FDR can be used to test whether

$$SPEC \sqsubseteq_T SAFE\_SYSTEM$$

and this time we will find that it does.

Here is an alternative way of checking $SYSTEM$. Notice that when a car and a train use the crossing at the same time, the event $crash$ occurs, and the system stops. This is the only point at which we have deliberately introduced $STOP$ into the system, and we hope that there are no other deadlocks.

If we use FDR to check $SYSTEM$ for deadlock-freedom, then every time a deadlock is found we will see a trace leading to $STOP$. If the trace ends in $crash$, then we have identified a violation of safety. If the trace ends with some other event, then there is another deadlock in the system, which presumably represents a mistake in our model.

In general there are many different ways of modelling a system, and many different ways of writing a specification. The challenge is to model the system in such a way that the bad property appears as a kind of behaviour (in this example, occurrence of $crash$) which can be ruled out by a suitable specification.

◇ Another Level Crossing ◇

Here is another way of modelling the level crossing. Remove the *crash* event, and change the definition of CT to

$$\begin{aligned} CT \;=\; & car.leave \rightarrow T \\ & \Box\; train.leave \rightarrow C. \end{aligned}$$

Also introduce

$$E_G \;=\; \{gate.raise, gate.lower\}$$

and change the definition of E_S to

$$E_S = E_C \cup E_T \cup E_G.$$

Similarly to before,

$$\begin{aligned} SYSTEM \;=\; & ((CR \;{}_{E_T \cup E_C}\|_{E_{GC}}\; GATE) \\ & \quad {}_{E_S}\|_{E_C}\; CARS) \;{}_{E_S}\|_{E_T}\; TRAINS. \end{aligned}$$

The specification now consists of two parts.

$$\begin{aligned} SPEC1 \;&=\; RUN_{E_G} \\ SPEC2 \;&=\; train.approach \rightarrow train.enter \rightarrow \\ & \quad\; train.leave \rightarrow SPEC2 \\ & \;\Box\; car.approach \rightarrow car.enter \rightarrow \\ & \quad\; car.leave \rightarrow SPEC2 \\ SPEC \;&=\; SPEC1 \;{}_{E_G}\|_{E_T \cup E_C}\; SPEC2 \end{aligned}$$

$SPEC1$ allows the gate to be raised and lowered freely. $SPEC2$ only allows trains and cars to enter the crossing separately.

Again we can check

$$SPEC \sqsubseteq_T SYSTEM$$

which is not true, and define

$$SAFE\_SYSTEM = SYSTEM \;{}_{E_S}\|_{E_S}\; CONTROL$$

and check

$$SPEC \sqsubseteq_T SAFE\_SYSTEM$$

which is true.

◇ Input and Output ◇

So far we have treated all events in the same way, regardless of whether they are thought of as inputs or outputs. It is useful, however, to introduce separate notation for inputs and outputs.

We will use events of the form $c.v$ where c is the name of a *channel* and v is the *value* of a message passing along the channel. Each channel has a *type*, which is simply the set of possible values which can be transmitted along it. If the type of c is T, then the set of events associated with c is $\{c.t \mid t \in T\}$.

We can define two new forms of prefixing. The process $c!v \rightarrow P$ outputs the message v on the channel c and then behaves like P. We require $v \in T$, where T is the type of c. In fact, $c!v \rightarrow P = c.v \rightarrow P$ (using the ordinary prefix notation), but the $c!v$ notation emphasises the fact that c and v are viewed as a channel and a message.

The process $c?x : T \rightarrow P(x)$ is prepared to input any value x of type T, and then behave like $P(x)$. In the ordinary menu choice notation,

$$c?x : T \rightarrow P(x) =$$
$$y : \{c.z \mid z \in T\} \rightarrow P(message(y)),$$

where, if $y = c.z$, $message(y) = z$.

We can define input and output prefixes, using labelled transition rules, as follows.

$$\frac{}{(c!v \to P) \xrightarrow{c.v} P}$$

$$\frac{}{(c?x : T \to P(x)) \xrightarrow{c.v} P(v)}\ [v \in T]$$

Example:

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

$$COPY = in?x : \mathbb{N} \to out!x \to COPY$$

$$SQUARE = in?x : \mathbb{Z} \to out!(x * x) \to SQUARE$$

$\diamond$ Specifications $\diamond$

Recall the definitions for the specification of the system consisting of the student and the college.

$$
\begin{aligned}
STUDENT &= year1 \to (pass \to YEAR2 \\
&\qquad\qquad\quad \mid fail \to STUDENT) \\
YEAR2 &= year2 \to (pass \to YEAR3 \\
&\qquad\qquad\quad \mid fail \to YEAR2) \\
YEAR3 &= year3 \to (pass \to graduate \to STOP \\
&\qquad\qquad\quad \mid fail \to YEAR3)
\end{aligned}
$$

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

$$CF = fail \to CF \;\Box\; pass \to CF$$

$$SYSTEM = STUDENT \;{}_S\|_C\; COLLEGE$$

Initially we defined

$$
\begin{aligned}
SPECF &= pass \to SPECF \mid fail \to SPECF \\
SPEC &= pass \to SPEC1 \mid fail \to SPECF \\
SPEC1 &= pass \to SPEC2 \mid fail \to SPECF \\
SPEC2 &= pass \to prize \to STOP \mid fail \to SPECF
\end{aligned}
$$

but the specification

$$SPECP \sqsubseteq_T SYSTEM$$

is not quite what we want, because it does not allow $SYSTEM$ to do $year1$, $year2$, $year3$ or $graduate$.

◇ The Correct Specification ◇

To allow for $year1, year2, year3$ and $graduate$ we defined

$$\begin{aligned} EXTRA &= year1 \rightarrow EXTRA \\ &\mid year2 \rightarrow EXTRA \\ &\mid year3 \rightarrow EXTRA \\ &\mid graduate \rightarrow EXTRA \end{aligned}$$

and then

$$SPEC = SPECP \;{}_{SP}\|_{E}\; EXTRA$$

where

$$\begin{aligned} SP &= \{pass, fail, prize\} \\ E &= \{year1, year2, year3, graduate\}. \end{aligned}$$

In general, to simplify the definition of processes such as $EXTRA$, we can define, for any set A of events, the process RUN_A.

$$RUN_A = x : A \rightarrow RUN_A$$

Then $EXTRA = RUN_E$, and $SPECF = RUN_{\{pass,fail\}}$.

$\diamond$ Hiding $\diamond$

There is an alternative approach to this kind of specification. Instead of putting a process in parallel with the specification to generate the events which we don't care about, we can *hide* those events from the process being specified.

If we define

$NEWSYSTEM =$

$SYSTEM \setminus \{year1, year2, year3, graduate\}$

then the behaviour of $NEWSYSTEM$ is derived from that of $SYSTEM$ by making the listed events invisible. The traces of $NEWSYSTEM$ are the traces of $SYSTEM$ with these events removed.

Now we can simply write

$$SPEC \sqsubseteq_T NEWSYSTEM.$$

as the specification. $SPEC$ only involves the events which we are interested in, and the hiding in the definition of $NEWSYSTEM$ shows which events we are leaving out of the specification.

◇ Using Hiding ◇

Returning to the level crossing example, there is an alternative approach to specifying the desired behaviour. We can use hiding to avoid specifying the events which we don't care about. In this case, all we want to do is specify that $crash$ never occurs.

If we hide all the events except $crash$ from $SYSTEM$ (or $SAFE\_SYSTEM$) then all we need for the specification is a process which never does $crash$:

$$STOP \sqsubseteq_T SYSTEM \setminus (E_T \cup E_C \cup E_G)$$

◇ Defining Hiding ◇

The transition rules defining hiding are

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\,[a \in A]$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A}\,[a \notin A]$$

As we saw when using FDR, the hidden events are replaced by τ, representing “silent” or “internal” events. τ events are not normally included in traces, although as we have seen, FDR can show where in a trace the τ events occur. When we discuss traces, we will not include τ.

◇ 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.

$\diamond$ External Choice $\diamond$

The process $P \sqcap Q$ (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:

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

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

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

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

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 $\mid$ (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.

$\diamond$ Defining External Choice $\diamond$

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

$$SERVICE \xrightarrow{board.37.A} pay.90 \rightarrow \ldots \mid alight.37.A \rightarrow STOP$$

$$SERVICE \xrightarrow{board.111.A} pay.70 \rightarrow \ldots \mid alight.111.A \rightarrow STOP$$

$\diamond$ Internal Choice $\diamond$

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.

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

$$TRAIN \,\square\, SERVICE$$

which expands to

$$TRAIN \,\square\, (BUS37 \sqcap BUS111).$$

We have the transition

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

and so the transition rules for external choice give

$$\begin{array}{c} TRAIN \,\square\, (BUS37 \sqcap BUS111) \\ \Big\downarrow \tau \\ TRAIN \,\square\, BUS37 \end{array}$$

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

$$TRAIN \,\Box\, (BUS37 \sqcap BUS111) \xrightarrow{board.train.A} alight.train.B \rightarrow STOP$$

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

$\diamond$ Nondeterminism $\diamond$

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

$$a \to P \mid b \to Q$$

is equivalent to

$$a \to P \,\Box\, b \to Q.$$

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

$$a \to P \,\Box\, a \to Q$$

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

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

$$(a \to P) \,\Box\, (a \to Q).$$

What does this mean? The process

$$a \to P \,\Box\, a \to Q$$

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

$$a \to P \mathbin{\Box} a \to Q \mathbin{\Box} b \to R$$

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.

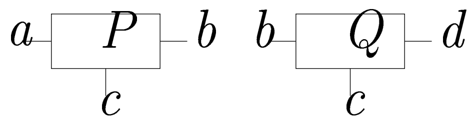
We will eventually see that

$$a \to P \mathbin{\Box} a \to Q = a \to P \sqcap a \to Q$$

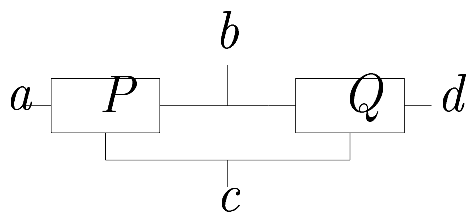
which emphasises the fact that the environment cannot choose between P and Q.

◇ Connection Diagrams ◇

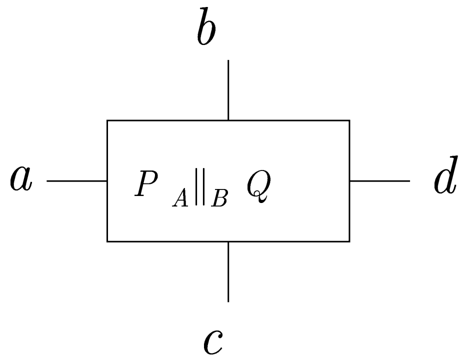
We can think of a process P with alphabet $A = \{a, b, c\}$ as a box with three possible points of connection to the outside world. Similarly, Q with alphabet $B = \{b, c, d\}$.



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 $i \neq j$.

$\diamond$ General Internal Choice $\diamond$

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

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

Remember that $\sqcap$ is a specification construct.

$\diamond$ General Parallel $\diamond$

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.

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

$$\begin{aligned} PERSON_n &= enter.n \rightarrow PRESENT_n \\ PRESENT_n &= leave.n \rightarrow PERSON_n \\ &\ \ \Box\ meeting \rightarrow PRESENT_n \end{aligned}$$

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:

$$
\begin{aligned}
PRINTER &= request1 \rightarrow print \rightarrow PRINTER \\
&\;\Box\; request2 \rightarrow print \rightarrow PRINTER \\
USER1 &= request1 \rightarrow work1 \rightarrow USER1 \\
USER2 &= request2 \rightarrow work2 \rightarrow USER2
\end{aligned}
$$

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.

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

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

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*.

$\triangle$ 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.

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

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

$$TEAM = PETE \parallel DAVE$$

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.

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

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

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

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 $\checkmark$, 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.

◇ 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.

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

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.

$\diamond$ Channels and Connections $\diamond$

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

where we suppose that

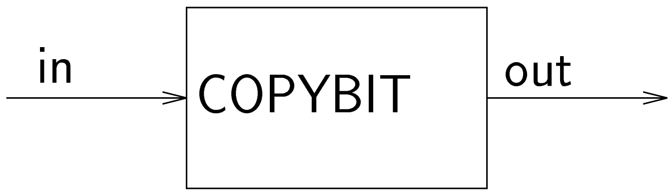
$$in(COPYBIT) = out(COPYBIT) = \{0, 1\}.$$

$COPYBIT$ 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.

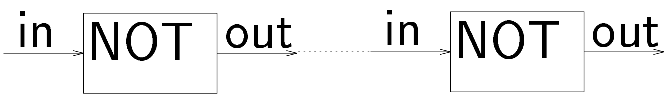


A variation on $COPYBIT$ is an inverter:

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

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

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

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

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

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 $f(NOT) \parallel g(NOT)$, 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,

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

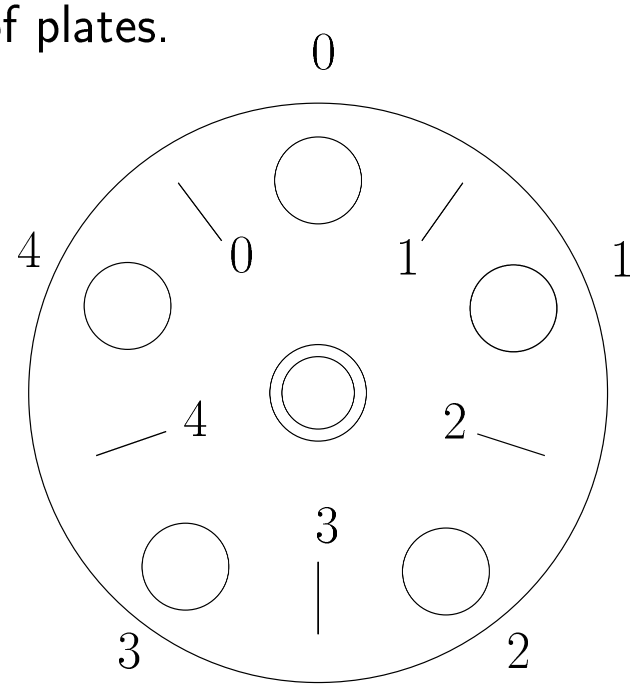
cannot do the mid events.

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

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

◇ The Dining Philosophers ◇

Five philosophers live in a college; they spend most of their time thinking, but occasionally become hungry. The college has a communal dining room, with a circular table and five chairs. In the middle of the table is a large bowl of rice, and the table is set with five plates. There are also five chopsticks, one between each pair of plates.



When a philosopher is hungry, he enters the dining room, sits down in his chair, picks up the chopsticks on either side of his plate (first the one on the left, then the one on the right), and eats. Two chopsticks are needed to eat rice, so if one of the chopsticks is already in use, he has to wait. When the philosopher has finished eating he puts down the chopsticks, gets up from the chair, and leaves the dining room.

We will model this system in CSP, and analyse its behaviour. The relevant components are the five philosophers, which we will model as processes $PHIL_0 \ldots PHIL_4$, and the five chopsticks, which we will model as processes $CHOP_0 \ldots CHOP_4$.

Using the symbols $\oplus$ and $\ominus$ to denote addition and subtraction modulo 5 (so that $4 \oplus 1 = 0$ and $0 \ominus 1 = 4$), philosopher $PHIL_i$ will sit in seat i and use chopsticks i and $i \oplus 1$.

The alphabet of $PHIL_i$ is

$$\begin{aligned}\alpha(PHIL_i) \;=\; & \{sitdown_i, getup_i,\\ & pickup.i.i, pickup.i.(i \oplus 1),\\ & putdown.i.i, putdown.i.(i \oplus 1)\}\end{aligned}$$

In the events $pickup.i.i$ etc. the "." is being used purely as a symbol.

The event $pickup.i.i$ represents $PHIL_i$ picking up chopstick i, and so on.

We will ignore the actions of eating, thinking, and entering and leaving the dining room.

Because the alphabets of the processes $PHIL_i$ are mutually disjoint, there can be no direct interaction between the philosophers. The only way in which they affect each other will be as a consequence of the fact that they are competing for access to the chopsticks.

The relevant events for the chopsticks are the $pickup$ and $putdown$ events. $CHOP_i$ can potentially be picked up or put down by either $PHIL_i$ or $PHIL_{i \ominus 1}$.

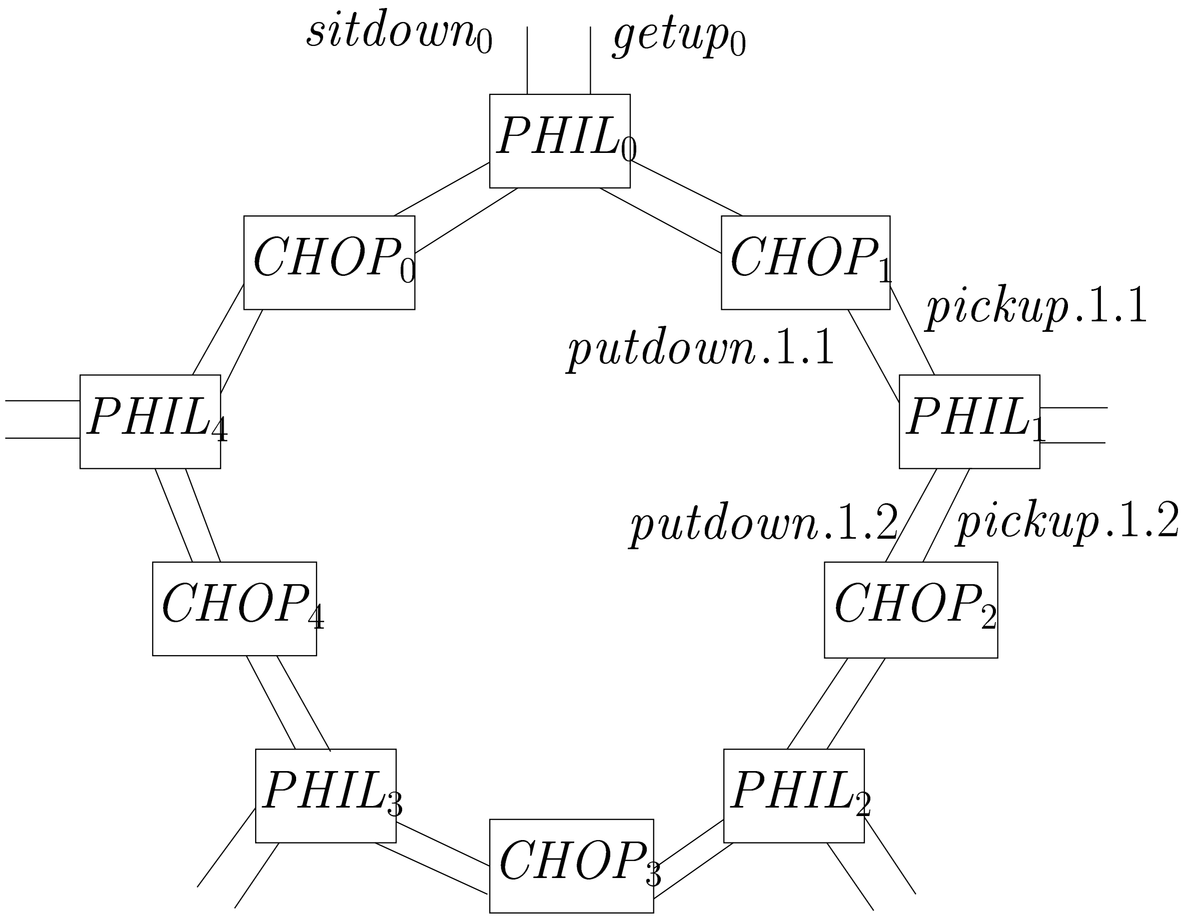
$$\begin{aligned}\alpha(CHOP_i) = \{&pickup.i.i, pickup.(i \ominus 1).i,\\ &putdown.i.i, putdown.(i \ominus 1).i\}\end{aligned}$$

We will define the system as a concurrent combination of the philosophers and the chopsticks.

$$PHILS = \|^{i \in 0..4} PHIL_i$$

$$CHOPS = \|^{i \in 0..4} CHOP_i$$

$$COLLEGE = PHILS \parallel CHOPS$$



Each process $PHIL_i$ simply cycles through a sequence of six events:

$$\begin{aligned} PHIL_i \; = \; & sitdown_i \rightarrow pickup.i.(i \oplus 1) \rightarrow pickup.i.i \rightarrow \\ & putdown.i.(i \oplus 1) \rightarrow putdown.i.i \rightarrow \\ & getup_i \rightarrow PHIL_i \end{aligned}$$

Each process $CHOP_i$ can be repeatedly picked up and put down, but there is a choice of who picks it up:

$$\begin{aligned} & CHOP_i = \\ & \quad pickup.i.i \rightarrow putdown.i.i \rightarrow CHOP_i \\ & \quad \Box \; pickup.(i \ominus 1).i \rightarrow putdown.(i \ominus 1).i \rightarrow CHOP_i \end{aligned}$$

Now we can look at the possible behaviour of $COLLEGE$. Suppose all the philosophers sit down in order, and then each one picks up the chopstick to his left.

What can happen next? Each $PHIL_i$ can only do $pickup.i.i$, which requires synchronisation with $CHOP_i$. However, $CHOP_i$ has just done $pickup.(i \ominus 1).i$ and therefore can only do $putdown.(i \ominus 1).i$ next. This means that there is no possible next event for $COLLEGE$. We have a deadlock.

How can we modify $COLLEGE$ to remove the possibility of deadlock? There are a number of obvious but unsatisfactory ideas.

- ◇ Provide two chopsticks for each philosopher. But if the chopsticks represent scarce resources, this may not be feasible.
- ◇ Provide a single extra chopstick, in the middle of the table, which can be used by any of the philosophers. Similarly, this may not be feasible.
- ◇ Modify the definition of just one of the philosophers, so that the chopsticks are picked up in the opposite order. This will work (although it takes some thought to be sure) but it breaks the symmetry of the system.

Instead we will try to control the way in which the philosophers sit down, the idea being that if only 4 philosophers are seated at any one time, then even if everyone picks up the left chopstick, one philospher will be sitting on the left of an empty place, and can pick up the chopstick to his right.

As we have seen, the behaviour of a system can be controlled by adding another process in parallel, and taking advantage of the fact that certain events require synchronisation.

We can define a process $BUTLER$ with alphabet

$$\alpha(BUTLER) = D \cup U,$$

where

$$\begin{aligned} D &= \{sitdown_0, \ldots, sitdown_4\} \\ U &= \{getup_0, \ldots, getup_4\}, \end{aligned}$$

to control the sitting down and getting up of the philosophers. $BUTLER$ is defined in terms of auxiliary processes $BUTLER_0, \ldots, BUTLER_4$, all with alphabet $\alpha(BUTLER)$.

$$\begin{aligned} BUTLER_0 &= x : D \rightarrow BUTLER_1 \\ BUTLER_i &= x : D \rightarrow BUTLER_{i+1} \\ &\quad \Box \; y : U \rightarrow BUTLER_{i-1} \;\; 1 \leqslant i \leqslant 3 \\ BUTLER_4 &= y : U \rightarrow BUTLER_3 \\ BUTLER &= BUTLER_0 \end{aligned}$$

The notation in the second line is shorthand for

$$BUTLER_i = z : (D \cup U) \rightarrow P(z)$$

where

$$\begin{aligned} P(z) &= BUTLER_{i+1} \text{ if } z \in D \\ &= BUTLER_{i-1} \text{ if } z \in U. \end{aligned}$$

Now we can define

$$NEWCOLLEGE = COLLEGE \parallel BUTLER$$

$\triangle$ Convince yourself that $NEWCOLLEGE$ does not deadlock. How formal can you be?

We could consider checking the entire state space of the system, to discover whether or not it can deadlock. Since each philosopher has 6 states and each chopstick has 3 states, the total number of possible states of $COLLEGE$ is $6^5 \times 3^5$, or about 1.8 million, though not all of these will be reachable (since the states of the chopsticks must be consistent with the states of the philosophers). Since the effect of $BUTLER$ is to restrict the number of states, this is also a limit on the number of states of $NEWCOLLEGE$. Systems of this complexity are within the scope of current software tools such as FDR.

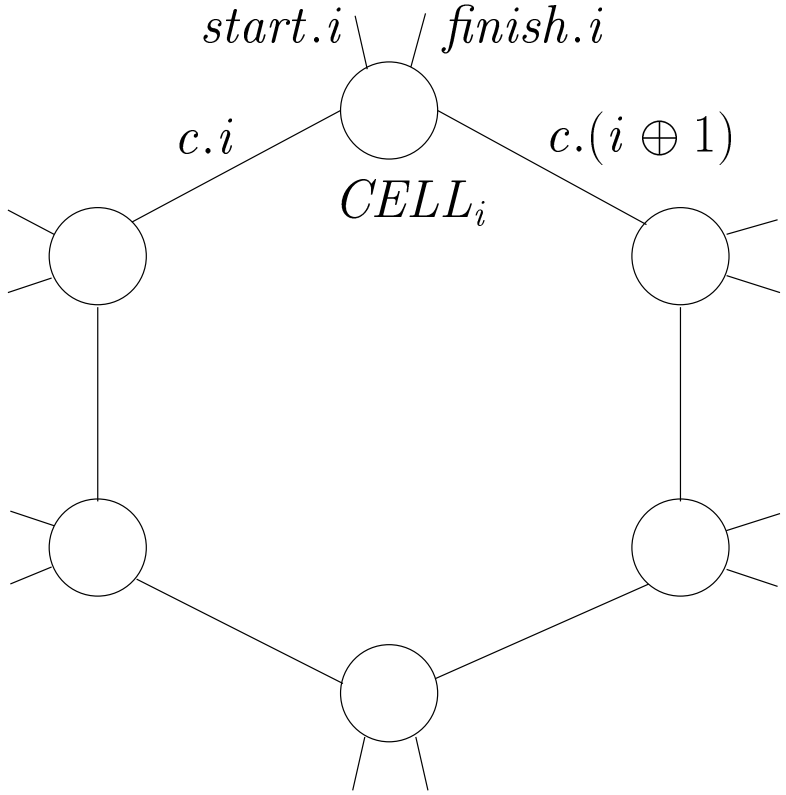
◇ The Cyclic Scheduler ◇

Suppose there are a number of processes which we need to control. Each process can be started by a $start$ event and uses a $finish$ event to indicate that it has finished. Suppose also that we want to start the processes in order, returning to the first when the last has been started; when a process has finished it can be started again, but only when its turn comes round.

We will define a *scheduler*, which uses $start$ and *finish* events to control the processes. The scheduler will be implemented as a collection of cells (processes), each of which communicates with one of the processes being controlled, and also with other cells.

The next slide has a diagram of the case where there are 6 processes to control. $\oplus$ denotes addition modulo the number of processes.

The idea is that each cell waits for a signal on the c channel to its left, which means that the process to its left has been started. Then the cell starts its own process, and sends a signal on the c channel to its right, to tell the next cell that it can start its process. Also, each cell has to wait for its process to do $finish$ before starting it again.



In order to start everything off, one cell must begin by starting its process instead of waiting.

$$
\begin{array}{rcl}
STARTCELL_i & = & start.i \to c.(i \oplus 1) \to \\
 & & (\ \ finish.i \to c.i \to STARTCELL_i \\
 & & \Box\ \ c.i \to finish.i \to STARTCELL_i)
\end{array}
$$

The other processes wait for $c.i$

$$WAITCELL_i = c.i \to STARTCELL_i$$

It is convenient to define

$$
\begin{array}{rcl}
CELL_0 & = & STARTCELL_0 \\
CELL_i & = & WAITCELL_i \quad (i > 0)
\end{array}
$$

and then

$$
\begin{array}{rcl}
SCHED & = & (\|^{0 \leqslant i < n}_{\{start.i, finish.i, c.i, c.(i \oplus 1)\}} CELL_i) \setminus \\
 & & \{c.i \mid 0 \leqslant i < n\}
\end{array}
$$

There are three properties which we would like to verify for the scheduler. The first is that for each i, the events $start.i$ and $\mathit{finish}.i$ happen alternately, beginning with $start.i$. The second is that the events $start.0, \ldots, start.(n-1)$ happen in the correct cyclic order. The third is deadlock-freedom.

For the first property, we can define a process specifying alternation of $start$ and finish for each cell:

$$ALT_i = start.i \rightarrow \mathit{finish}.i \rightarrow ALT_i$$

and combine them in parallel to produce a specification for the scheduler as a whole.

$$ALTSPEC = \|^{0 \leqslant i < n}_{\{start.i, \mathit{finish}.i\}} ALT_i$$

In this parallel combination the alphabets are all disjoint, and no synchronisation is required. It is simply an independent parallel combination of the ALT processes.

The specification

$$ALTSPEC \sqsubseteq_T SCHED$$

can be checked with FDR.

For the second property, define

$$CYCLE_i \ = \ start.i \rightarrow CYCLE_{i \oplus 1} \ (0 \leqslant i < n)$$

and specify

$$CYCLE_0 \sqsubseteq SCHED \setminus \{\mathit{finish}.i \mid 0 \leqslant i < n\}.$$

◇ Traces and Choice ◇

Which traces can be produced by $P \Box Q$ and $P \sqcap Q$? We know that $P \Box Q$ can do the first event of either P or Q, and then behave like the remainder of P or Q. Therefore any trace of either P or Q can be produced by $P \Box Q$, and we have

$$traces(P \Box Q) = traces(P) \cup traces(Q).$$

$P \sqcap Q$ always does τ first, and then behaves like either P or Q. Because τ does not appear in traces, we also have

$$traces(P \sqcap Q) = traces(P) \cup traces(Q).$$

We have previously considered *trace equivalence*, written $P =_T Q$, as a definition of when two processes should be considered equal or interchangeable. However, we can now see that $P \Box Q =_T P \sqcap Q$, even though internal and external choice have been designed to behave in different ways.

In general, trace equivalence is not suitable as a definition of process equivalence.

Before we introduced $\sqcap$ and $\Box$ all processes were deterministic — the internal state was always determined by the observable events. For deterministic processes, *traces* are all we need to know, and trace equivalence is adequate. But the whole point of introducing the $\sqcap$ operator was so that a process could make an internal state change without doing anything observable. Similarly, if P and Q have a common event a available at the first step, then observation of the event a from $P \Box Q$ does not tell us what the internal state has become.

We will now try to say exactly what the difference between $P \sqcap Q$ and $P \Box Q$ is, and develop a new notion of process equivalence accordingly.

$\diamond$ Refusals $\diamond$

Suppose we have the following definitions.

$$\begin{aligned} P &= a \rightarrow P \\ Q &= b \rightarrow Q \end{aligned}$$

What happens if we put each of $P \,\Box\, Q$ and $P \sqcap Q$ in an environment consisting of P? i.e. if we look at $(P \,\Box\, Q) \; {}_{\{a,b\}}\|_{\{a,b\}} \; P$ and $(P \sqcap Q) \; {}_{\{a,b\}}\|_{\{a,b\}} \; P$.

First, we have $P \,\Box\, Q \xrightarrow{a} P$
and $P \xrightarrow{a} P$
so

$$(P \,\Box\, Q) \; {}_{\{a,b\}}\|_{\{a,b\}} \; P \xrightarrow{a} P \; {}_{\{a,b\}}\|_{\{a,b\}} \; P.$$

Also,

$$P \; {}_{\{a,b\}}\|_{\{a,b\}} \; P \xrightarrow{a} P \; {}_{\{a,b\}}\|_{\{a,b\}} \; P$$

so

$$P \; {}_{\{a,b\}}\|_{\{a,b\}} \; P = P$$

(they both satisfy the same recursive definition).

So

$$(P \,\Box\, Q) \; {}_{\{a,b\}}\|_{\{a,b\}} \; P = a \rightarrow P$$

i.e.

$$(P \,\Box\, Q) \; {}_{\{a,b\}}\|_{\{a,b\}} \; P = P.$$

On the other hand,

$$(P \sqcap Q)\ {}_{\{a,b\}}\|_{\{a,b\}}\ P \xrightarrow{\tau} P\ {}_{\{a,b\}}\|_{\{a,b\}}\ P$$

and

$$(P \sqcap Q)\ {}_{\{a,b\}}\|_{\{a,b\}}\ P \xrightarrow{\tau} Q\ {}_{\{a,b\}}\|_{\{a,b\}}\ P$$

so

$$(P \sqcap Q)\ {}_{\{a,b\}}\|_{\{a,b\}}\ P =$$

$$(P\ {}_{\{a,b\}}\|_{\{a,b\}}\ P) \sqcap (Q\ {}_{\{a,b\}}\|_{\{a,b\}}\ P).$$

(This is a loose statement as we haven't decided what "$=$" means yet.)

We know that $P\ {}_{\{a,b\}}\|_{\{a,b\}}\ P = P$
and $Q\ {}_{\{a,b\}}\|_{\{a,b\}}\ P = STOP$
So

$$(P \sqcap Q)\ {}_{\{a,b\}}\|_{\{a,b\}}\ P = P \sqcap STOP.$$

This shows that $P \,\square\, Q$ and $P \sqcap Q$ behave differently when put in parallel with P. One is just P, the other can internally choose to deadlock (become $STOP$).

We can use this observation to develop a general approach to distinguishing between nondeterministic processes. We will consider putting a process P in an environment Q, where the alphabets of P and Q are the same, i.e. constructing $P\ {}_{\alpha(P)}\|_{\alpha(P)}\ Q$.

Let X be a set of events which are offered initially by Q. If it is possible for $P \; {}_{\alpha(P)}\|_{\alpha(P)} \; Q$ to deadlock at the first step, then we say that X is a *refusal* of P. The set of all refusals of P is obtained by considering all possible sets X which could be initial event sets of Q.

Examples:

1. The empty set is a refusal of every process, because if $Q = STOP$ then $P \; {}_{\alpha(P)}\|_{\alpha(P)} \; Q = STOP$.

2. Any set of events X is a refusal of $STOP$.

3. If $a \notin X$ then X is a refusal of $a \rightarrow P$. So if $\alpha(P) = \{a, b, c\}$ then the refusals of $a \rightarrow P$ are $\{\}$, $\{b\}$, $\{c\}$ and $\{b, c\}$. Processes Q causing

$$(a \rightarrow P) \; {}_{\{a,b,c\}}\|_{\{a,b,c\}} \; Q$$

to deadlock include $STOP$, $b \rightarrow STOP$, $c \rightarrow a \rightarrow STOP$, $(b \rightarrow STOP) \,\Box\, (c \rightarrow c \rightarrow STOP)$, etc.

4. The refusals of $(a \rightarrow c \rightarrow STOP) \,\Box\, (b \rightarrow STOP)$ are $\{\}$ and $\{c\}$.

5. The refusals of $(a \rightarrow c \rightarrow STOP) \sqcap (b \rightarrow STOP)$ are $\{\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, c\}$ and $\{b, c\}$.

We can define

$$refusals(P) = \{X \mid X \subseteq \alpha(P) \text{ and } X \text{ is a refusal of } P\}.$$

Note that $refusals(P)$ is a set of sets of events. For example,
$refusals((a \to STOP) \sqcap (b \to STOP)) =$
$\{\{\}, \{a\}, \{b\}, \{c\}, \{a, c\}, \{b, c\}\}$.

In the examples we saw that
$refusals((a \to STOP) \,\Box\, (b \to STOP)) \neq$
$refusals((a \to STOP) \sqcap (b \to STOP))$.

In general, $refusals(P \,\Box\, Q) \neq refusals(P \sqcap Q)$, and this will be the basis for a new definition of process equality which allows us to distinguish between internal and external choice.

We can now define $refusals$ for processes defined in terms of the operators we have seen so far.

$$refusals(STOP) = \{X \mid X \subseteq \Sigma\}$$

where Σ is the set of all events being considered — the universal set of events.

$$refusals(a \to P) = \{X \mid X \subseteq (\alpha(P) - \{a\})\}$$

Both of these definitions are subsumed by the definition for menu choice: if $P = x : A \to P(x)$ then

$$refusals(P) = \{X \mid X \subseteq (\alpha(P) - A)\}$$

If P can refuse X then so will $P \sqcap Q$ if P is selected. Similarly every refusal of Q is a possible refusal of $P \sqcap Q$.

$$refusals(P \sqcap Q) = refusals(P) \cup refusals(Q)$$

$P \;\Box\; Q$ can only refuse X if both P and Q can refuse X.

$$refusals(P \;\Box\; Q) = refusals(P) \cap refusals(Q)$$

$P \;{}_A\|_B\; Q$ can refuse all events refused by P and all events refused by Q.

$$\begin{aligned} refusals(P \;{}_A\|_B\; Q) \;=\; & \{X \cup Y \mid X \in refusals(P) \\ & \quad \text{and } Y \in refusals(Q)\} \end{aligned}$$

Refusals allow us to distinguish formally between deterministic and nondeterministic processes. If a process is deterministic then it can never refuse any event which it could possibly do. In other words, if P is deterministic and a is a possible initial event for P, then a does not appear in any refusal set of P.

Writing $initials(P)$ for the set of possible initial events of P (so $initials(P) = \{x \mid \langle x \rangle \in traces(P)\}$), we can say that if P is deterministic then

$$\begin{aligned} refusals(P) \;=\; & \{X \mid X \subseteq \alpha(P) \text{ and} \\ & \quad X \cap initials(P) = \{\}\}. \end{aligned}$$

Determinism means that any event which is possible cannot be taken away by an internal state transition.

Examples: If

$$P = a \rightarrow c \rightarrow STOP \mid b \rightarrow STOP$$

then $initials(P) = \{a, b\}$ and $refusals(P) = \{\{\}, \{c\}\}$.

If

$$P = (a \rightarrow c \rightarrow STOP) \sqcap (b \rightarrow STOP)$$

then $initials(P) = \{a, b\}$ and (as before)

$$refusals(P) = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, c\}, \{b, c\}\}.$$

Although a is a possible initial event for P, P could also internally choose to be $b \rightarrow STOP$ which refuses a.

To define nondeterminism properly, we need to consider events refused not just at the first step, but after any sequence of events. For example,

$$(a \rightarrow b \rightarrow STOP) \,\Box\, (a \rightarrow c \rightarrow STOP)$$

is nondeterministic, but this does not become apparent until after the first event.

So: P is deterministic if and only if

$$\begin{array}{l} \forall\, tr \in traces(P) \bullet \\ \quad (refusals(P/tr) = \\ \qquad \{X \subseteq \alpha(P) \mid X \cap initials(P/tr) = \{\}\}). \end{array}$$

P/tr is the process whose behaviour is whatever P could do after the trace tr.

◇ Failure Equivalence ◇

A first attempt at a new definition of process equivalence might be to define $P =_r Q$ as

$$\begin{array}{l} traces(P) = traces(Q) \\ refusals(P) = refusals(Q) \end{array}$$

but this is not quite what we want. It would make

$$a \rightarrow ((b \rightarrow STOP) \mathrel{\Box} (c \rightarrow STOP))$$

and

$$a \rightarrow ((b \rightarrow STOP) \sqcap (c \rightarrow STOP))$$

equivalent, which is no better than using trace equivalence. The problem is that looking at *refusals* can only detect differences at the first step. As with the definition of determinism, we need to look at events refused after arbitrary traces have been observed.

The solution is to define $failures(P)$ as follows:

$$\begin{array}{rcl} failures(P) & = & \{(tr, X) \mid tr \in traces(P) \\ & & \text{and } X \in refusals(P/tr)\} \end{array}$$

and then say that $P =_F Q$ means

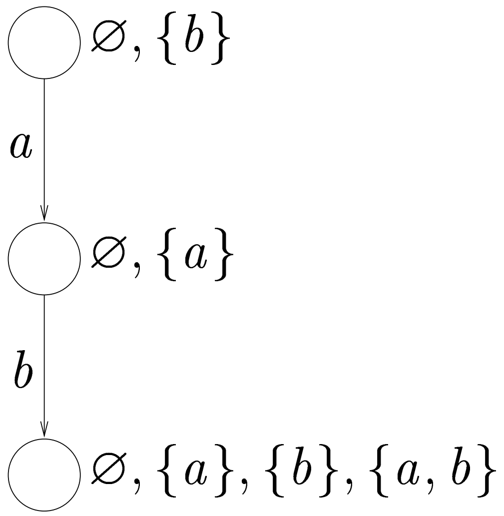
$$\begin{array}{c} traces(P) = traces(Q) \\ \text{and} \\ failures(P) = failures(Q). \end{array}$$

◇ Examples ◇

$P = a \to b \to STOP$

$\alpha(P) = \{a, b\}$

$$
\begin{aligned}
failures(P) \ = \ \{ \ & (\langle\rangle, \varnothing), (\langle\rangle, \{b\}), \\
& (\langle a\rangle, \varnothing), (\langle a\rangle, \{a\}), \\
& (\langle a, b\rangle, \varnothing), (\langle a, b\rangle, \{a\}), \\
& (\langle a, b\rangle, \{b\}), (\langle a, b\rangle, \{a, b\})\}
\end{aligned}
$$

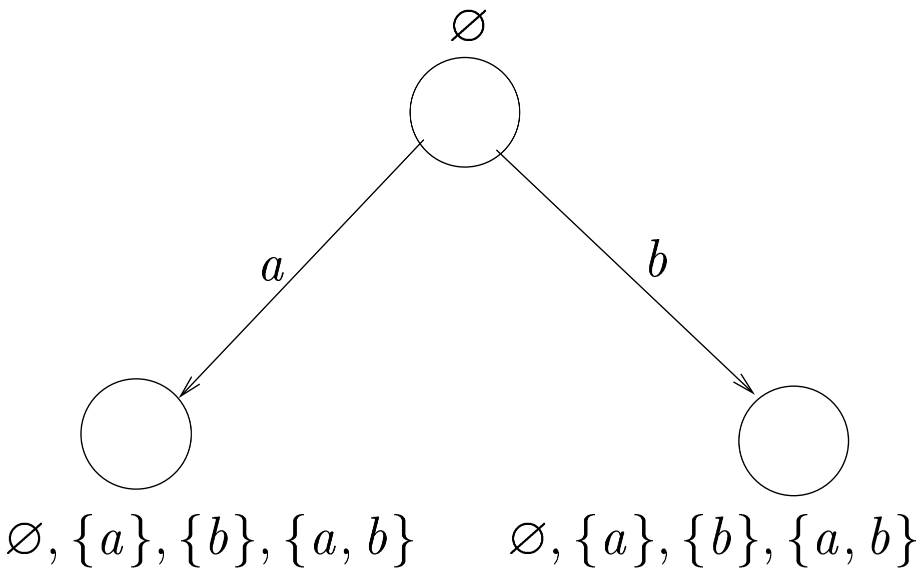


◇ Examples (ctd) ◇

$P = a \rightarrow STOP \mid b \rightarrow STOP$

$\alpha(P) = \{a, b\}$

$$\begin{aligned} failures(P) \ = \ & \{(\langle\rangle, \varnothing)\} \\ & \cup\{(\langle a\rangle, X) \mid X \subseteq \{a, b\} \\ & \cup\{(\langle b\rangle, X) \mid X \subseteq \{a, b\} \end{aligned}$$

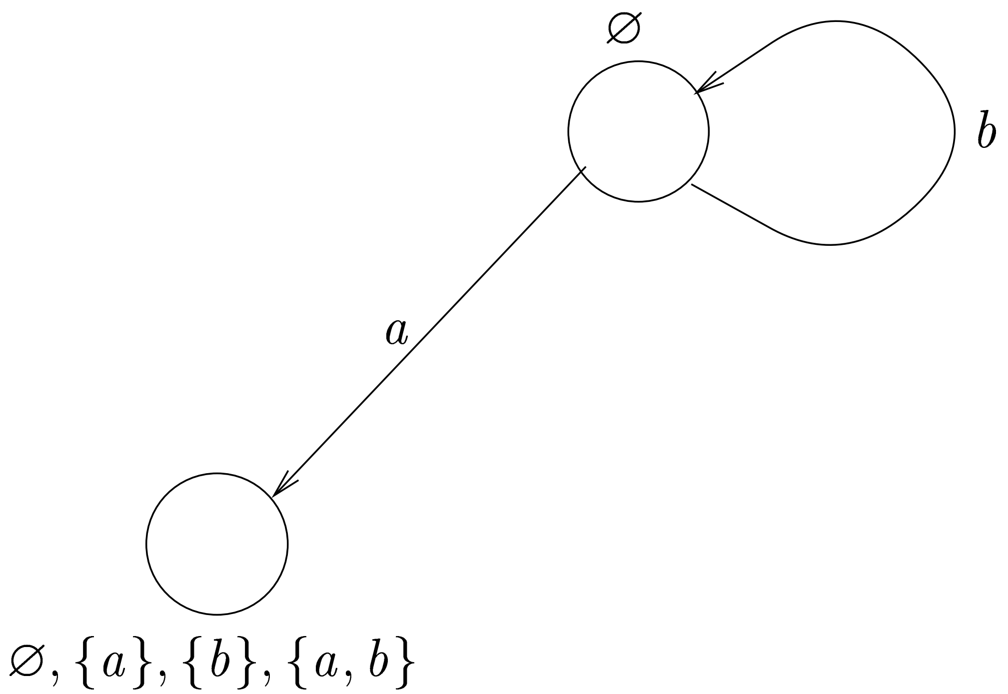


$\diamond$ Examples (ctd) $\diamond$

$P = a \to STOP \mid b \to P$

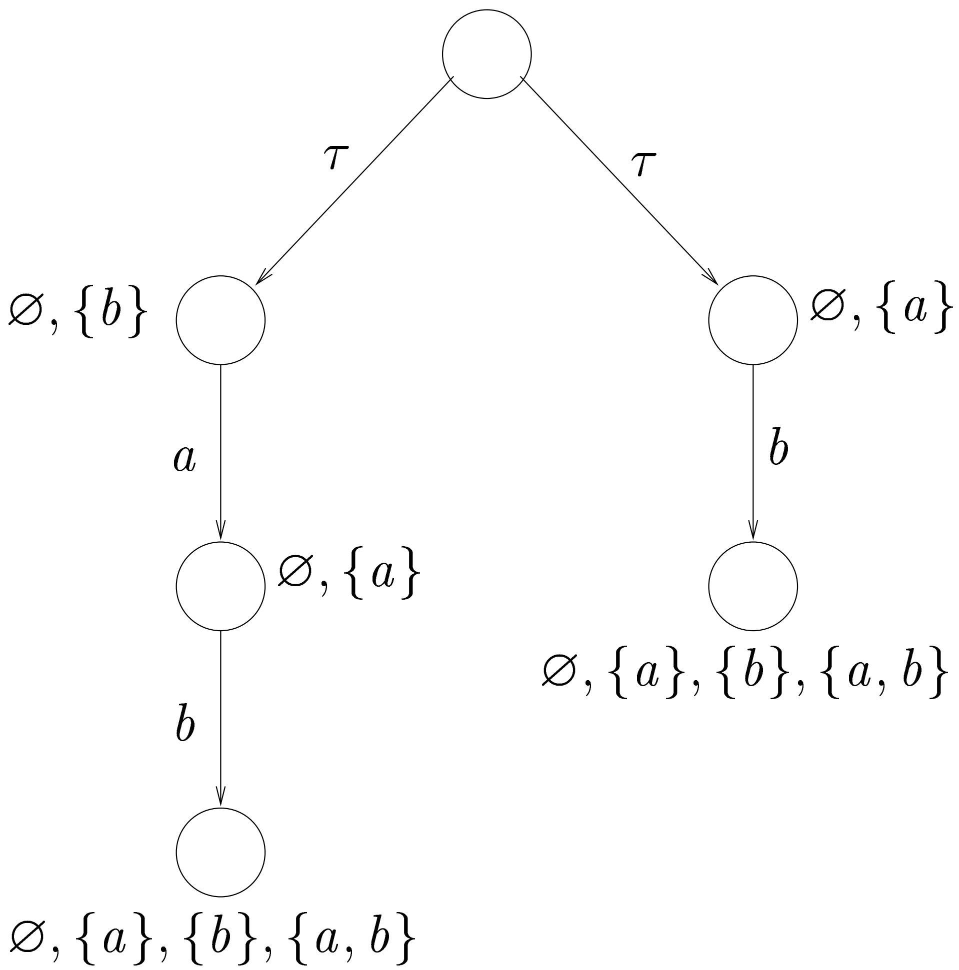
$\alpha(P) = \{a, b\}$

$$\begin{aligned} failures(P) \ = \ & \{(\langle b\rangle^n, \varnothing) \mid n \geqslant 0\} \\ & \cup\{(\langle b\rangle^n \frown \langle a\rangle, X) \mid \ n \geqslant 0 \wedge \\ & \qquad\qquad\qquad\qquad\quad X \subseteq \{a, b\}\} \end{aligned}$$



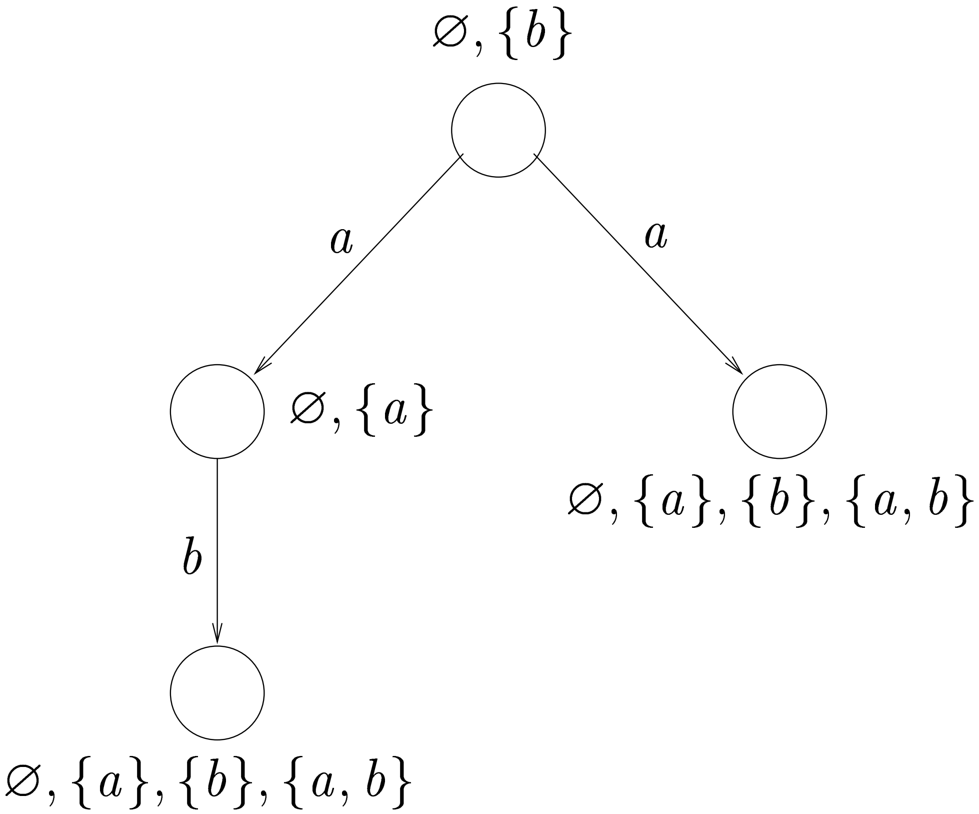
$\diamond$ Examples (ctd) $\diamond$

$P = a \to b \to STOP \sqcap b \to STOP$



◇ Examples (ctd) ◇

$$P = a \to b \to STOP \,\Box\, a \to STOP$$



Recall that $\{\} \in refusals(P)$ for every process P. This means that for every process P and every trace $tr \in traces(P)$, $(tr, \{\}) \in failures(P)$. So $traces$ can be recovered from $failures$ by

$$traces(P) = \{tr \mid (tr, \{\}) \in failures(P)\}.$$

This means that if $failures(P) = failures(Q)$ then $traces(P) = traces(Q)$, so the definition of failure equivalence can be simplified to

$$failures(P) = failures(Q).$$

If P is deterministic, we can analyse $failures(P)$ slightly more.

$$\begin{aligned}
&failures(P)\\
&\quad = \{(tr, X) \mid tr \in traces(P) \text{ and } X \in refusals(P/tr)\}\\
&\quad = \{(tr, X) \mid tr \in traces(P)\\
&\qquad\quad \text{and } X \cap initials(P/tr) = \{\}\}\\
&\quad = \{(tr, X) \mid tr \in traces(P)\\
&\qquad\quad \text{and } X \cap \{x \mid s \frown \langle x \rangle \in traces(P)\} = \{\}\}
\end{aligned}$$

which shows that $failures(P)$ can be defined in terms of $traces(P)$.

So if P and Q are deterministic, and $traces(P) = traces(Q)$, then $failures(P) = failures(Q)$.

Any process defined using just $STOP$, prefixing, menu choice (or $|$), $\|$ and guarded recursion, is deterministic.

◇ Failure Refinement ◇

Failure refinement is defined in a similar way to trace refinement.

$$P \sqsubseteq_F Q$$
if and only if
$$\mathit{failures}(Q) \subseteq \mathit{failures}(P)$$

It is pronounced "P is failure refined by Q".

To see how failure refinement can be used in specifications, consider a very simple example: the process

$$SPEC = a \rightarrow b \rightarrow SPEC$$

Recall that if we use $SPEC$ as a specification with *trace* refinement, we get a *safety* specification. Processes P satisfying the specification

$$SPEC \sqsubseteq_T P$$

include

$$\begin{aligned}
P &= STOP \\
P &= a \rightarrow STOP \\
P &= a \rightarrow (b \rightarrow P \,\Box\, b \rightarrow STOP) \\
P &= a \rightarrow b \rightarrow P
\end{aligned}$$

What is the effect of specifying

$$SPEC \sqsubseteq_F P?$$

We need to calculate $failures(SPEC)$. In words first: the traces of $SPEC$ are alternating sequences of a and b events, starting with a. After a trace ending in a, $SPEC$ refuses the sets $\varnothing$ and $\{a\}$. After a trace ending in b, it refuses the sets $\varnothing$ and $\{b\}$. So:

$$\begin{aligned} failures(SPEC) \;=\; & \{(\langle a, b\rangle^n \frown \langle a\rangle, \varnothing) \mid n \geqslant 0\} \\ \cup\; & \{(\langle a, b\rangle^n \frown \langle a\rangle, \{a\}) \mid n \geqslant 0\} \\ \cup\; & \{(\langle a, b\rangle^n, \varnothing) \mid n \geqslant 0\} \\ \cup\; & \{(\langle a, b\rangle^n, \{b\}) \mid n \geqslant 0\}. \end{aligned}$$

To determine whether $SPEC \sqsubseteq_F STOP$ we need to calculate that

$$\begin{aligned} failures(STOP) \;=\; & \{(\langle\rangle, \varnothing), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}), \\ & (\langle\rangle, \{a, b\})\} \end{aligned}$$

and then we can see that the failure pairs $(\langle\rangle, \{a\})$ and $(\langle\rangle, \{a, b\})$ are in $failures(STOP)$ but not in $failures(SPEC)$. Therefore it is not the case that $SPEC \sqsubseteq_F STOP$. We could also write this as

$$SPEC \not\sqsubseteq_F STOP.$$

Now look at $P = a \rightarrow STOP$.

$$\begin{aligned} failures(a \rightarrow STOP) \;=\; & \{(\langle\rangle, \varnothing), (\langle\rangle, \{b\}), (\langle a\rangle, \varnothing), \\ & (\langle a\rangle, \{a\}), (\langle a\rangle, \{b\}), \\ & (\langle a\rangle, \{a, b\})\} \end{aligned}$$

The failure pairs $(\langle a\rangle, \{b\})$ and $(\langle a\rangle, \{a, b\})$ are in $failures(P)$ but not in $failures(SPEC)$, so again $SPEC \not\sqsubseteq_F P$.

$\diamond$ Exercise $\diamond$

If we define $P = a \rightarrow (b \rightarrow P \Box b \rightarrow STOP)$, is it true that $SPEC \sqsubseteq_F P$? Either show that all the failure pairs of P are also failure pairs of $SPEC$, or find a failure pair of P which is not a failure pair of $SPEC$.

$\diamond$ Liveness $\diamond$

$SPEC \sqsubseteq_F P$ is a *liveness* specification which requires P to do certain events. Which definitions of P satisfy the specification? Obviously

$$P = a \rightarrow b \rightarrow P$$

does, because that is the same process as $SPEC$. In fact this is the only process satisfying this specification. So in this example, the specification is very restrictive indeed: it pins down the implementation precisely.

◇ Safety and Liveness ◇

Saying that $tr \in traces(P)$ is a *positive* statement: it describes something that P can do. A specification of the form

$$SPEC \sqsubseteq_T P$$

puts a limit on the traces that P can do, so it is a specification which restricts behaviour.

Saying that $(tr, X) \in failures(P)$ is a *negative* statement: it describes something that P cannot do. A specification of the form

$$SPEC \sqsubseteq_F P$$

puts a limit on what P can fail to do, so it requires P to accept at least a certain range of behaviours.

Alternatively: P fails a safety (trace) specification by doing too much. P fails a liveness (failure) specification by refusing too much, i.e. by not doing enough.

◇ Another Example ◇

Process P will have alphabet $\{a, b, c\}$, and we want to specify that P must be able to do an infinite sequence of alternating a and b events, starting with a; we do not care when c events occur.

We can use the process

$$ALT = a \rightarrow b \rightarrow ALT$$

as a specification for the a and b events, as before. To allow the c events to occur freely we use hiding, and express the specification as

$$ALT \sqsubseteq_F (P \setminus \{c\})$$

Definitions of P satisfying this specification include

$$\begin{aligned}
P &= a \rightarrow b \rightarrow P \\
P &= c \rightarrow a \rightarrow c \rightarrow c \rightarrow b \rightarrow P \\
P &= a \rightarrow b \rightarrow c \rightarrow P \\
P &= a \rightarrow c \rightarrow b \rightarrow a \rightarrow b \rightarrow P
\end{aligned}$$

because in each case, $P \setminus \{c\}$ is the same process as ALT.

Definitions of P not satisfying the specification include

$$\begin{aligned}
Q &= c \rightarrow b \rightarrow Q \\
P &= a \rightarrow (b \rightarrow P \;\Box\; b \rightarrow Q) \\
P &= a \rightarrow b \rightarrow (P \;\Box\; a \rightarrow c \rightarrow STOP).
\end{aligned}$$

◇ Level Crossing Liveness ◇

In our model of the level crossing, there is an infinite stream of cars trying to cross, and also an infinite stream of trains. We can specify liveness (the requirement that whenever a car approaches it should eventually be allowed to cross, and similarly for the trains) as follows.

$$\begin{aligned} CARSPEC &= car.approach \rightarrow car.enter \rightarrow \\ &\quad car.leave \rightarrow CARSPEC \\ TRAINSPEC &= train.approach \rightarrow train.enter \rightarrow \\ &\quad train.leave \rightarrow TRAINSPEC \end{aligned}$$

The specifications are

$$CARSPEC \sqsubseteq_F (SAFE\_SYSTEM \setminus \{train, gate\})$$

$$TRAINSPEC \sqsubseteq_F (SAFE\_SYSTEM \setminus \{car, gate\})$$

(all the $gate.???$ events are hidden, etc.)

These specifications can be checked using FDR.

◇ Scheduler Liveness ◇

A liveness specification for the cyclic scheduler is that the processes continue to be started, in turn, forever. This can be written

$$CYCLE_0 \sqsubseteq_F (SCHED \setminus \{finish\})$$

where $CYCLE_0$ is the process which was used for the safety specification, and all the $finish.i$ events are hidden. This specification can be checked with FDR.

Another liveness specification might be to pick a particular process i and specify that $start.i$ and $finish.i$ keep happening alternately forever. This can be done with a specification process in which $start.i$ and $finish.i$ alternate, by hiding all the other $start$ and $finish$ events in $SCHED$.