1

Sequential processes

1.1 EVENTS AND PROCESSES

Any approach to describing the world must concentrate on features of interest. Architects, engineers, economists, cartographers, biologists, physicists, and computer scientists all categorize and describe the world from their own particular point of view, appropriate to the phenomena they are trying to understand and control. They will focus on those aspects of the world relevant to their study.

This book is concerned with the description and analysis of systems which consist of interacting components. In such systems it is the myriad possibilities for interaction between components that are difficult to understand. Since we are interested not only in understanding such systems, but also in designing them, the description language used will influence how we think about systems, and will dictate the way in which these systems will be designed.

The language of Communicating Sequential Processes (CSP) was designed for describing systems of interacting components, and it is supported by an underlying theory for reasoning about them. The conceptual framework taken by CSP is to consider components, or *processes*, as independent self-contained entities with particular interfaces through which they interact with their environment. This viewpoint is compositional, in the sense that if two processes are combined to form a larger system, that system is again a self-contained entity with a particular interface—a (larger) process. This is the framework provided by CSP for analysing the world.

Example 1.1.1 The kitchen of a fast-food outlet might be considered as a process. Its interface will include the door through which the ingredients come in, the counter where the cooked food is passed to the till staff, and the tannoy on which orders come in.

Another process within the fast-food outlet is the customer serving area. The interface here will include the tills, the till counter where the customer's food is placed, the tannoy for relaying orders to the kitchen, the food counter for picking up food placed there by the kitchen.

The kitchen and the customer serving area may be considered as distinct processes, and this separation may be appropriate from the management and company organization point of view. Furthermore, their combination will also be a process, whose interface will include the tills, the till counter, and the door through which ingredients come into the kitchen.

The kitchen itself need not be considered as an atomic process, and may instead be viewed as a combination of more primitive processes, such as a grill process, a deep-fry process, a microwave process, and an ingredients-sort-and-distribute process. ■

Since a process interacts with other processes only through its interface, the important information in the description of a process concerns its behaviour on that interface. In describing systems made up of interacting components and analysing the effects of their interaction, the appropriate level will abstract away the internal workings of the process and will focus on its activity at the interface: its external activity.

The interface of a process will be described as a set of *events*. An event describes a particular kind of atomic indivisible action that can be performed or suffered by the process. In describing a process, the first issue to be decided must be the set of events which the process can perform. This set provides the framework for the description of the process.

Example 1.1.2 A printer can accept jobs, and it can print them. Its interface may be given as the set $\{accept, print\}$. ■

Example 1.1.3 A telephone has 12 buttons, a handset, and a bell. The handset may be lifted or replaced. The telephone's interface might be given as the set

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \#, *, handset.lift, handset.replace, ring\}$$

This set is precisely the ways in which the telephone can interact with its environment. ■

Example 1.1.4 A lift system which serves floors 0 to 3 has an *up* button on each floor (apart from the top), a *down* button on each floor (apart from the bottom) and a *goto.i* button for each floor, within the lift. It also has doors at each floor which can open and close. Finally, it has an emergency *halt* button within the lift. Its interface will be described by the following:

$$\{up.0, up.1, up.2, down.1, down.2, down.3,$$
$$goto.0, goto.1, goto.2, goto.3,$$
$$open.0, close.0, open.1, close.1, open.2, close.2, open.3, close.3,$$
$$halt\}$$

All interaction with the lift is via this set of events. ■

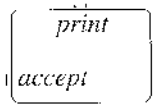


Fig. 1.1 A machine with buttons

Fig. 1.2 A black box with wires

Processes may be thought of in a number of ways: a machine with a collection of buttons corresponding to the events in its interface, as in Figure 1.1; or alternatively as a black box with a collection of wires corresponding to the events in its interface, as illustrated by Figure 1.2.

The interface given for a process can be considered as its static specification. Its dynamic specification describes how it will actually behave at its interface. A process will be willing to engage in interface events only at particular times. More generally, there will be constraints on the sequences of events that it can engage in. For example, the lift in Example 1.1.4 would be required to alternate on the opening and closing of doors. The dynamic part of the process description will describe its permissible patterns of events.

Transitions

An operational semantics provides a way of interpreting a language - of stepping through executions of programs written in that language. It describes an operational understanding of the language. CSP is concerned with the performance of events, so the operational semantics will describe at what stages events may occur.

The operational semantics in fact defines how a CSP interpreter should execute. It provides the first possible execution steps (if any) for any CSP process, together with an expression of the subsequent behaviour in the form of another CSP process. An execution step will be described in terms of *labelled transitions* of the form $P_1 \xrightarrow{\mu} P_2$, where P_1 and P_2 are both processes. This describes a *transition* from P_1 to P_2, or equivalently a change in state. The label μ describes the action which accompanies this transition. It can be either an external event (from P_1's interface), a termination event $\checkmark$ (introduced on page 13), or an internal action τ which indicates that no interface event accompanied the change of state. The set of all possible external events is denoted Σ, so μ will range over $\Sigma \cup \{\checkmark, \tau\}$, which is written $\Sigma^{\checkmark,\tau}$. Variables a, b, c, will be used for events that must be external: they range over $\Sigma \cup \{\checkmark\}$, which is abbreviated $\Sigma^{\checkmark}$.

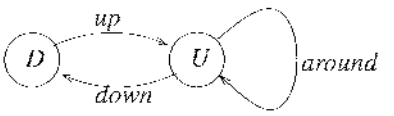


Fig. 1.3 The finite state machine for D and U

The labelled transition $P_1 \xrightarrow{\mu} P_2$ asserts that there is an execution of P_1 which begins with the occurrence of the event μ, and its subsequent behaviour is that of process P_2. The operational semantics offers a way of stepping through executions one step at a time. Since any execution unfolds one step at a time, this operational semantics provides all the information necessary to step through an execution. At every stage, the rules will describe the next possible steps (if any) for the execution.

Example 1.1.5 The following system has two states, D and U, and three transitions between them:

- $D \xrightarrow{up} U$
- $U \xrightarrow{around} U$
- $U \xrightarrow{down} D$

This describes the finite state machine of Figure 1.3. ■

Event names will be written in lower case, and process names in upper case.

Inference rules

An *inference rule* allows the *deduction* of a predicate from a collection of other predicates. It will be of the following general form:

$$\frac{\begin{array}{l}\text{antecedent 1}\\ \vdots\\ \text{antecedent } n\end{array}}{\text{conclusion}} \quad [\text{ side condition }]$$

This rule allows the conclusion to be deduced if all of the antecedents are true, and the side condition is also true. In the special case where there are no antecedents and no side condition, then the conclusion may be immediately deduced.

A number of conclusions which may all be drawn from the same set of antecedents may be listed as conclusions one after the other beneath the line. This provides an alternative to writing a separate rule with the same antecedents and side condition for each conclusion.

Inference rules will be used in two ways in this book. Firstly, rules may be given to formalize inferences concerning particular kinds of predicate. These rules can be independently checked by considering the meaning of the predicate. For example, the rule *modus ponens* can be given in this way:

$$\frac{p \quad p \Rightarrow q}{q}$$

If p and q are both logical statements, then *modus ponens* allows q to be deduced from the pair of statements p and $p \Rightarrow q$. The proof rule can be checked for soundness by considering the possible meanings of p and q: when both antecedents are true, then so too must be the conclusion.

The law of the excluded middle is an example of a rule with no antecedents:

$$\frac{}{p \vee \neg p}$$

The inference rules given in the later chapters concerning **sat** specifications are of this kind: an independent definition of the **sat** relation is given, and the rules are sound with respect to this definition, and provide ways of reasoning about it.

Rules may also be used *axiomatically* to *define* predicates. For example, if the relation 'is a parent of' is already known, then a pair of rules can be used to define the relation 'is an ancestor of':

$$\frac{p \text{ is a parent of } q}{p \text{ is an ancestor of } q} \qquad \frac{p \text{ is an ancestor of } q \quad q \text{ is a parent of } r}{p \text{ is an ancestor of } r}$$

The relation 'is an ancestor of' is defined to hold between two people precisely when the rules can be used to deduce this. Technically, it is the smallest relation closed under these inference rules.

Structured operational semantics are conventionally defined in this way, and this will be the approach taken in this book. The ternary relation $P_1 \xrightarrow{\mu} P_2$ between P_1, P_2, and μ, asserts that there is a transition labelled μ between P_1 and P_2. The relation $- \xrightarrow{-} -$ will be defined axiomatically through inference rules. A process P_1 can perform a μ transition to P_2 precisely when the relation $P_1 \xrightarrow{\mu} P_2$ can be deduced from the rules.

The operational semantics is just this relation between terms of the language and event labels.

1.2 PERFORMING EVENTS

The simplest process of all is *STOP*. This process is never prepared to engage in any of its interface events. It might be used to describe the fast-food outlet after it has closed down, or a broken printer that cannot accept or print jobs.

The operational semantics for *STOP* are extremely simple. It has no event transitions. Any execution of *STOP* will be unable to make any progress, and will remain in the same state for ever. An explicit description of its interface will describe precisely what it is unable to perform.

Event prefix

If P is a CSP process, and $a \in \Sigma$ is an event in the interface of P, then the following new process may be constructed:

$$a \rightarrow P$$

It is pronounced 'a then P'. This process is initially able to perform only a, and after performing a it behaves as P. The labelled transition semantics captures this understanding:

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

There are no antecedents and no side condition to this rule. It is always the case that $a \rightarrow P$ may perform an a transition and subsequently behave as P.

Example 1.2.1 A one-shot printer is described by the process

$$PRINTER0 \quad = \quad accept \rightarrow print \rightarrow STOP$$

Initially it is able only to *accept* a job, after which it will behave as $print \rightarrow STOP$. This subsequent process is able to *print* a job, after which no further action is possible. Its complete maximal execution is described as

$$\begin{array}{l} accept \rightarrow print \rightarrow STOP \\ \quad \downarrow accept \\ print \rightarrow STOP \\ \quad \downarrow print \\ STOP \end{array}$$

The corresponding finite state machine is given in Figure 1.4. ■

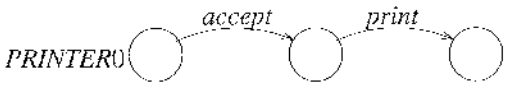


Fig. 1.4 The finite state machine for *PRINTER0*

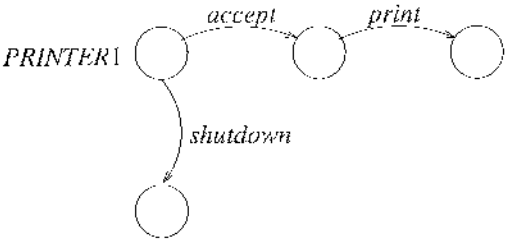


Fig. 1.5 The finite state machine for *PRINTER1*

Choosing between events

If $A \subset \Sigma$ is a set of events, and for each a in A the process $P\langle a \rangle$ is defined, then a new process can be defined:

$$x : A \rightarrow P\langle x \rangle$$

This is called a menu choice, or prefix choice, since a menu of events A is offered as a prefix to the subsequent behaviour. It is pronounced 'x from A then $P\langle x \rangle$'. This process is prepared initially to engage in any of the events in the set A. After an event a is chosen, the subsequent behaviour is that of the process $P\langle a \rangle$ corresponding to the event a.

Example 1.2.2 A printer which initially has a *shutdown* option as well as an *accept* option can be described using this form of choice. The initial choice is between *accept* and *shutdown*. The process following *accept* is to be $print \rightarrow STOP$, and the behaviour subsequent to *shutdown* is simply *STOP*. This situation may be described as follows:

$$\begin{aligned} PRINTER1 &= x : \{accept, shutdown\} \rightarrow P\langle x \rangle \\ &\text{where} \\ P\langle accept \rangle &= print \rightarrow STOP \\ P\langle shutdown \rangle &= STOP \end{aligned}$$

The corresponding finite state machine is given in Figure 1.5. ■

Prefix choice allows a notation for conditional choices to be introduced to CSP. In a choice $x : A \rightarrow P\langle x \rangle$, the definition of $P\langle x \rangle$ might involve a conditional. For example, the

printer of the example above might have P defined by

$$P(x) = \begin{array}{ll} print \rightarrow STOP & \text{if } x = accept \\ STOP & \text{otherwise} \end{array}$$

or even by

if $x = accept$
then $print \rightarrow STOP$
else $STOP$

Neither of these is strictly within the language of CSP. Rather, they are constructions used in the definition of a parameterized process $P(x)$. However, they are conventionally used within CSP descriptions, resulting for example in a description of *PRINTER*1 as follows:

$$PRINTER1 = x : \{accept, shutdown\} \rightarrow \begin{array}{l} \textbf{if } x = accept \\ \textbf{then } print \rightarrow STOP \\ \textbf{else } STOP \end{array}$$

In the case where the choice set A is finite, of the form $\{a_1, a_2 \ldots a_n\}$, the branches of the choice may be listed explicitly as follows:

$$\begin{array}{l} a_1 \rightarrow P(a_1) \\ \mid a_2 \rightarrow P(a_2) \\ \vdots \\ \mid a_n \rightarrow P(a_n) \end{array}$$

Example 1.2.3 The printer above can be written as follows:

$$PRINTER1 = \begin{array}{l} accept \rightarrow print \rightarrow STOP \\ \mid shutdown \rightarrow STOP \end{array}$$

The events offered by the choice are listed explicitly. ■

Example 1.2.4 A printer which begins with a *startup* event:

$$PRINTER2 = startup \rightarrow \begin{array}{l} (accept \rightarrow print \rightarrow STOP \\ \mid shutdown \rightarrow STOP) \end{array}$$

The choice is offered after the first event. ■

In the case where the set A is empty, the choice is equivalent to $STOP$. No initial events are possible, so there can be no subsequent behaviour.

The transitions for $x : A \to P(x)$ are given by the following rule:

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

For each $a \in A$ there is a corresponding transition. There are no other transitions.

Compound events

Events are considered to be atomic and indivisible in their occurrence. However, a single event may still contain various pieces of information, so events can have some structure. An example of this has already been given in Example 1.1.4, where events are structured by the kind of event they are, together with the floor they are concerned with. Another instance of a structured event is given by a communication channel which carries messages. In order to model values v being communicated along channel c, each possible communication is described as a separate possible event $c.v$ in the interface of the process. If a process P has an input channel in that carries 0s and 1s, then both $in.0$ and $in.1$ will appear in the interface set of P. The event $in.0$ describes the appearance of value 0 on channel in. Events $in.0$ and $in.1$ are distinct events, though the intention is to consider them both as inputs of particular values along channel in.

If c is a particular channel name, and T is the *type* of the channel—the set of values that may be passed along it then the set $\{c.t \mid t \in T\}$ will be the set of events associated with c. For convenience this will be denoted $c.T$. More generally, it is often useful to allow a Cartesian generalization of the 'dot' separator to sets. For example, $c.d.S.T = \{c.d.s.t \mid s \in S \wedge t \in T\}$.

Example 1.2.5 The alphabet of the kitchen given in Example 1.1.1 might be given by

$$door.I \cup counter.F \cup tannoy.O$$

where I is the set of all possible ingredients, F is the set of food dishes, and O is the set of possible orders. ■

Input and output

If c is a channel name of type T, and v is a particular value of type T, then the CSP expression

$$c!v \to P$$

describes a process which is initially willing to output v along channel c, and subsequently behave as P. This means that the only event it is initially willing to perform is $c.v$, and its transition semantics is

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

This process has the same behaviour as $c.v \rightarrow P$, but the intention of the designer in considering it as output is made explicit. It is simply a convenient syntactic distinction.

If processes $P(x)$ are defined for each $x \in T$ then the CSP input expression

$$c?x : T \rightarrow P(x)$$

describes a process which is initially ready to accept any value x of type T along channel c. Its subsequent behaviour, described by $P(x)$, is determined by the value v that it receives as input.

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

Example 1.2.6 A 'squaring' server could be described by

$$in?x : \mathbb{N} \rightarrow out!(x^2) \rightarrow STOP$$

The output value is the square of the input. ■

Example 1.2.7 If $JOBS$ is the set of all possible print jobs that can be accepted by a printer, then a more detailed description of a one-shot printer would be

$$PRINTER3 \quad = \quad accept?j : JOBS \rightarrow print!j \rightarrow STOP$$

■

Example 1.2.8 A multiplication server could be described by

$$in?m : \mathbb{N} \rightarrow in?n : \mathbb{N} \rightarrow out!(m * n) \rightarrow STOP$$

or alternatively by

$$in?(m, n) : (\mathbb{N} \times \mathbb{N}) \rightarrow out!(m * n) \rightarrow STOP$$

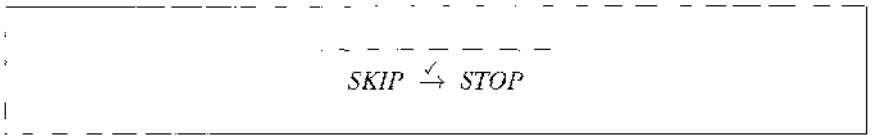
The first process takes in one input followed by another, and then produces an output. The second process requires the pair of numbers to be submitted as a single input. ■

Successful termination

Successful termination is the point that a process reaches when its execution has completed. The process representing this state is

SKIP

which can do nothing except indicate that it has reached termination. It achieves this by performing the special termination event $\checkmark$. It does nothing else.



$$SKIP \xrightarrow{\checkmark} STOP$$

The event $\checkmark$ is a special event used purely to denote termination, so it is not a member of the universal set of events Σ. It therefore cannot appear as an event prefix $a \rightarrow P$, or as one of the choices in a menu choice: such processes describe behaviour subsequent to their events, and this is inappropriate for termination.

1.3 RECURSION

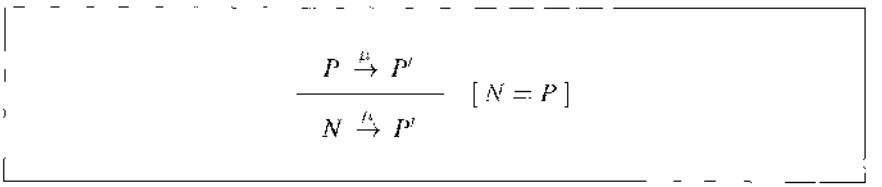
The process constructors introduced thus far allow the construction only of finite processes, which execute for a finite number of steps before stopping. In order to describe infinitely executing processes, a *recursion* construct is introduced. This allows looping executions to be defined. For example, the process $LIGHT = on \rightarrow off \rightarrow LIGHT$, illustrated in Figure 1.6, allows the alternation of the events *on* and *off* indefinitely.

A process *name* N may be used as a component process in a process definition. It is bound by the definition

$$N = P$$

where P is an arbitrary CSP expression which may include process name N. The process expression P is the *body* of the recursive definition.

The rule for unwinding a process name N recursively bound to a process definition P is as follows:



$$\frac{P \xrightarrow{a} P'}{N \xrightarrow{a} P'} \quad [N = P]$$

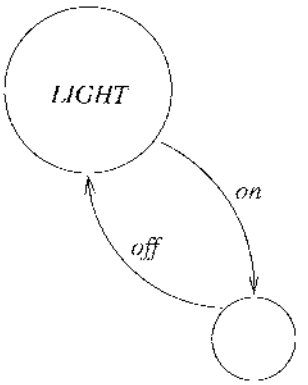


Fig. 1.6 The finite state machine for *LIGHT*

This rule states that any execution of P will be an execution of N.

Another way to consider P is as a process dependent on N. To make this relationship explicit, P may also be written as $F(N)$.

The notation $P_1[P_2/N]$ is used to denote the substitution meta-operation, where all (free) instances of the process name N appearing in P_1 are replaced by the process expression P_2. For example

$$\begin{aligned}(on \rightarrow off \rightarrow LIGHT)[on \rightarrow STOP/LIGHT] &= on \rightarrow off \rightarrow on \rightarrow STOP \\ (on \rightarrow off \rightarrow LIGHT)[Y/LIGHT] &= on \rightarrow off \rightarrow Y\end{aligned}$$

If $N = P$ is a recursive definition, then $F(Y) = P[Y/N]$ is the function (in Y) corresponding to the body of the definition.

Example 1.3.1 The process *LIGHT* is recursively defined as follows:

$$LIGHT = on \rightarrow off \rightarrow LIGHT$$

Equivalently, $LIGHT = F(LIGHT)$, where $F(Y) = on \rightarrow off \rightarrow Y$.

The execution of *LIGHT* unfolds as follows:

$$
\begin{array}{l}
LIGHT \\
\quad \downarrow on \\
off \rightarrow LIGHT \\
\quad \downarrow off \\
LIGHT \\
\quad \downarrow on \\
off \rightarrow LIGHT \\
\quad \vdots
\end{array}
$$

It may alternate between the states *LIGHT* and $off \rightarrow LIGHT$ for ever. ■

Example 1.3.2 The one-place buffer *COPY* is initially ready to accept any message of type T as input, and will then hold it until it is output.

$$COPY \quad = \quad in?x : T \rightarrow out!x \rightarrow COPY$$

After output, it returns to its initial state. ■

Example 1.3.3 A specification of a railway crossing describes the required interactions between the raising and lowering of the gate, and the arrival and departure of a train.

$$
\begin{array}{lll}
CROSS & = & train.approach \rightarrow train.enter \rightarrow train.leave \rightarrow CROSS \\
& & \quad | \; gate.raise \rightarrow train.approach \rightarrow gate.down \rightarrow \\
& & \qquad train.enter \rightarrow train.leave \rightarrow CROSS
\end{array}
$$

The initial state has the gate lowered, blocking road vehicles from crossing the rails. Either the gate is raised, or else a train approaches the crossing. If the gate is raised then it must be lowered on the approach of a train. If the train enters the crossing then it must leave before the gate may be raised. The transition graph for *CROSS* is given in Figure 1.7. Compound events are used here simply to associate each event with either the train or the gate. ■

Mutual recursion

A collection of recursive definitions will bind a number of process names to process definitions. It is often useful to allow the process definitions to contain a number of the names being defined, so that in fact the various processes are defined in terms of each other. This construction is known as *mutual recursion*.

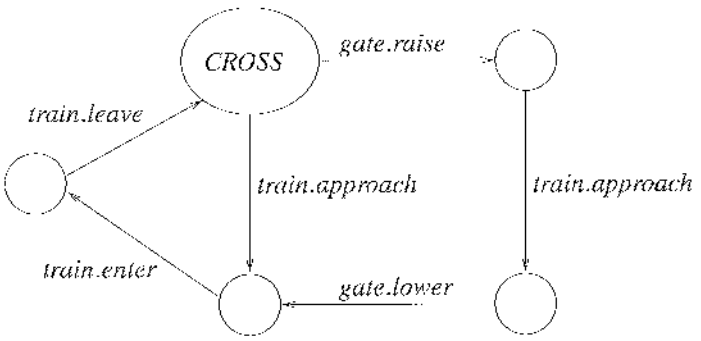


***Fig. 1.7 Transition graph for** CROSS*

Example 1.3.4 The process *LIGHT* may be defined in terms of a process *ON*, which is itself defined in terms of *LIGHT*:

$$\begin{aligned} LIGHT &= on \rightarrow ON \\ ON &= off \rightarrow LIGHT \end{aligned}$$

These recursive definitions define two processes, each in terms of the other. ■

In order for a set of recursive definitions to be a mutual recursion, each name appearing in any of the process bodies must be bound in one of the recursive definitions. The single definition $LIGHT = on \rightarrow ON$ by itself is not suitable as a recursive definition: the process name *ON* must also be bound.

The transition rule for unwinding a recursive definition is exactly the same as that given for a single recursion. The transitions that can be made for a process name N_i in the context of a collection of bindings which binds N_i to P_i are precisely the transitions of P_i.

$$\frac{P_i \xrightarrow{\mu} P'}{N_i \xrightarrow{\mu} P'} \quad [N_i = P_i]$$

The process *CROSS* defined in Example 1.3.3 might also have been given using a mutual recursion:

$$\begin{aligned} CROSS &= gate.raise \rightarrow train.approach \rightarrow gate.lower \rightarrow ENT \\ &\quad \mid train.approach \rightarrow ENT \\ ENT &= train.enter \rightarrow train.leave \rightarrow CROSS \end{aligned}$$

The behaviour of this version of *CROSS* is indistinguishable from the single recursive process given earlier.

One execution of *CROSS* is as follows:

$$
\begin{array}{l}
CROSS \\
\quad \downarrow gate.raise \\
(train.approach \rightarrow gate.lower \rightarrow ENT) \\
\quad \downarrow train.approach \\
(gate.lower \rightarrow ENT) \\
\quad \downarrow gate.lower \\
ENT \\
\quad \downarrow train.enter \\
(train.leave \rightarrow CROSS) \\
\quad \vdots
\end{array}
$$

This is one of the paths through the transition graph shown in Figure 1.7. The names of the recursive processes used in the definition of *CROSS* have been chosen to reflect the important states of the system: *ENT* is the point at which the train will enter the crossing.

This convention may be used more generally with a family of process names $N(i)$ parameterized by $i \in I$. A mutual recursion will bind them to a family of processes containing these names. Alternatively, they will be bound to a family of functions $F(i)$ where each is a function of the family of names $N(i)$.

Example 1.3.5 A heater has four power settings, which can be changed by the events *up* and *down*. We use the four process names $HEATER(0)$, $HEATER(1)$, $HEATER(2)$, and $HEATER(3)$ to describe the four possible states. Their interrelationships are described by mutual recursion:

$$
\begin{array}{lcl}
HEATER(0) & = & up \rightarrow HEATER(1) \\
HEATER(1) & = & up \rightarrow HEATER(2) \mid down \rightarrow HEATER(0) \\
HEATER(2) & = & up \rightarrow HEATER(3) \mid down \rightarrow HEATER(1) \\
HEATER(3) & = & down \rightarrow HEATER(2)
\end{array}
$$

At any point in the execution, the process will be at one of the $HEATER(i)$ nodes. The value of i might be thought of as the state of the heater, corresponding to the setting on a dial. ■

It is appropriate to keep track only of those aspects of internal state that have an impact on the external behaviour patterns of the process. The CSP notation is intended primarily to support description and analysis of processes in terms of their interactions. However, the interactions possible for a process might depend on the value of some internal state variable, and so it is necessary in such situations to keep track of the relevant information, but only in so far as it affects the process's external behaviour. In the *HEATER* example above, the

value of the state determines how many *up* and *down* events are possible. The heater might also contain a thermostat, but if its setting does not have any effect on the behaviour under consideration, then its value is superfluous to the description of the process, and should not be included.

Example 1.3.6 A counter can be *incremented* or *decremented* at any point, provided the total number of *decrement* events does not exceed the number of *increment* events. The family of process names $COUNT(i)$ will be used to define $COUNTER$, where i will track the difference between the number of *increment* events and the number of *decrement* events.

$$\begin{aligned} COUNT(0) &= increment \rightarrow COUNT(1) \\ COUNT(i) &= increment \rightarrow COUNT(i+1) \qquad \text{if } i > 0 \\ &\quad \mid decrement \rightarrow COUNT(i-1) \end{aligned}$$

The counter begins at 0:

$$COUNTER = COUNT(0)$$

If there could be any number of *increment* and *decrement* events, in any order, then it would be unnecessary to keep track of the difference between them, and the process description

$$C = increment \rightarrow C \mid decrement \rightarrow C$$

would be sufficient. State information should be carried only where it affects the possible executions of the process. ■

Example 1.3.7 An *ACCUMULATOR* is used to keep track of running totals of sequences of numbers. It has a *reset* event, a *query* channel on which the current total can be output, and an *add* channel where it is possible to add another number. The family of process names $TOT(i)$ will be used to define this process, where i represents the running total.

$$\begin{aligned} TOT(i) &= reset \rightarrow TOT(0) \\ &\quad \mid query!i \rightarrow TOT(i) \\ &\quad \mid add?x : \mathbb{N} \rightarrow TOT(i+x) \end{aligned}$$

ACCUMULATOR can now be defined:

$$ACCUMULATOR = TOT(0)$$

Its initial state will be 0. ■

Indices for recursive process names need not be restricted to numbers: more generally, any kind of index may be used. This allows processes to be parameterized by more abstract values such as sets or strings. They do not need to be directly representable within a computer.

Example 1.3.8 A process which models a set of elements of type T allows elements to be added, and provides information about whether a particular element is in the set. It will be parameterized by S, the current set of elements:

$$\begin{aligned} SET &= SET(\{\}) \\ SET(S) &= add?x : T \rightarrow SET(S \cup \{x\}) \\ &\quad \mid query?y : T \rightarrow \begin{cases} answer!yes \rightarrow SET(S) & \text{if } y \in S \\ answer!no \rightarrow SET(S) & \text{otherwise} \end{cases} \end{aligned}$$

The response depends both on the parameter S and the input y. ■

Example 1.3.9 A buffer of infinite capacity is always prepared to input a fresh message, and when it is non-empty it is prepared to output the message at the head of the queue. It may be parameterised by the sequence s of messages currently in the buffer.

$$\begin{aligned} BUFFER(\langle\rangle) &= in?x : M \rightarrow BUFFER(\langle x \rangle) \\ BUFFER(\langle y \rangle \frown s) &= in?x : M \rightarrow BUFFER(\langle y \rangle \frown s \frown \langle x \rangle) \\ &\quad \mid out!y \rightarrow BUFFER(s) \end{aligned}$$

x and y range over M, and s ranges over M^*, the (finite) strings of elements of M. The notation $\langle m \rangle$ represents a singleton sequence containing just m, and '$\frown$' is sequence concatenation, discussed in more detail in Section 4.1.

An initially empty buffer is described by

$$BUFFER = BUFFER(\langle\rangle)$$

One execution of $BUFFER$ is

$$\begin{array}{l} BUFFER \\ \quad \downarrow in.3 \\ BUFFER(\langle 3 \rangle) \\ \quad \downarrow in.8 \\ BUFFER(\langle 3, 8 \rangle) \\ \quad \downarrow out!3 \\ BUFFER(\langle 8 \rangle) \\ \quad \vdots \end{array}$$

The parameter consists of the items still in the buffer. ■

Example 1.3.10 The description of a stack is very similar to that of the buffer:

$$\begin{aligned} STACK(\langle\rangle) &= push?x : M \rightarrow STACK(\langle x \rangle) \\ STACK(\langle y \rangle \frown s) &= push?x : M \rightarrow STACK(\langle x \rangle \frown \langle y \rangle \frown s) \\ &\quad \mid pop!y \rightarrow STACK(s) \end{aligned}$$

The only difference is that input messages are placed at the beginning of the string rather than at the end. ■

1.4 CHOICE

Prefix choice has already introduced the possibility of process executions having a number of possible courses of action. Whereas that operator offers a choice between events, this section will introduce choices between processes.

In concurrent systems it is useful to distinguish between the cases where control over resolution of choice resides within a process itself, and where control is outside it. For example, a car showroom advertising cars in any colour might allow either the customer or the manufacturer Henry Ford [1] to make the choice; and these two possibilities are different. The distinction is important in concurrent systems, since problems may arise if two processes have both been given control over a particular choice. If both Henry Ford and the customer are considered to have control over the choice of colour then problems arise if they do not agree. It is therefore important to distinguish between *external choice*, where control over the choice is external to a process, and *internal choice*, where the environment of the process has no such control.

External choice

An external choice between two processes is initially ready to perform the events that either process can engage in. The choice is resolved by the performance of the first event, in favour of the process that performs it. This choice is written

$$P_1 \,\Box\, P_2$$

and is pronounced 'P_1 external choice P_2'.

[1] Ford offered the purchasers of his cars the choice of 'any colour as long as it's black'.

Example 1.4.1 A particular bus journey is covered by two bus routes: the 37 and the 111. The service offered for that journey is then described as the choice between these two bus services.

$$SERVICE \quad = \quad BUS\_37 \,\Box\, BUS\_111$$

This choice can be described even before the initial events of the *BUS* processes are known.

The bus services are used to travel from the bus station at A to a destination at B. The pertinent events for this journey in the description of a *BUS* process are *board*, *alight*, and *pay*.

$$\begin{aligned} BUS\_37 \quad = \quad & board.37.A \rightarrow (pay.90 \rightarrow alight.37.B \rightarrow STOP \\ & \mid alight.37.A \rightarrow STOP) \end{aligned}$$

On boarding the bus at A, a passenger must either pay the fare and then travel to B, or alight again at A without travelling.

The rival bus route charges a lower fare.

$$\begin{aligned} BUS\_111 \quad = \quad & board.111.A \rightarrow (pay.70 \rightarrow alight.111.B \rightarrow STOP \\ & \mid alight.111.A \rightarrow STOP) \end{aligned}$$

The description *SERVICE* describes the situation in the bus station. There is a choice of two buses, and the choice between them is resolved when the first one is boarded. ■

The transition rules for external choice reflect the fact that the first external event resolves the choice in favour of the process performing the event, and that the choice is not resolved on the occurrence of internal events.

$$\frac{P_1 \xrightarrow{a} P_1'}{\begin{array}{l} P_1 \,\Box\, P_2 \xrightarrow{a} P_1' \\ P_2 \,\Box\, P_1 \xrightarrow{a} P_1' \end{array}} \qquad \frac{P_1 \xrightarrow{\tau} P_1'}{\begin{array}{l} P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1' \sqsupset P_2 \\ P_2 \sqcup P_1 \xrightarrow{\tau} P_2 \sqsupset P_1' \end{array}}$$

Control over resolution of the choice is external because the events of both choices are initially available. Considering processes as machines with buttons, the buttons that are initially enabled are those enabled by either of the choice processes. The choice is made externally because the choice of which button to press is not restricted by the process. The choice of buses is not made by the process *SERVICE*, but this choice is instead offered to the customer.

However, if the same event is offered by both of the choice processes, then an external agent will not have control over which process is chosen. An external agent has control only

over the choice of initial event, not over the possible subsequent behaviours in the case where both processes offer the chosen initial event.

Example 1.4.2 The previous description of the bus service provided at the bus station was appropriate in the case where the passenger looks at the number on the front of the bus and so can distinguish the events *board.37.A* and *board.111.A*.

The buses available to a passenger who cannot read the bus number are better described simply as two buses *BUS_1* and *BUS_2*.

$$\begin{aligned} BUS\_1 &= board.A \rightarrow (pay.90 \rightarrow alight.B \rightarrow STOP \\ &\qquad \mid alight.A \rightarrow STOP) \\ BUS\_2 &= board.A \rightarrow (pay.70 \rightarrow alight.B \rightarrow STOP \\ &\qquad \mid alight.A \rightarrow STOP) \end{aligned}$$

The choice $BUS\_1 \,\Box\, BUS\_2$ still offers the option of boarding either bus, but since the two *board* events are not distinguished, the passenger has no control over which bus is boarded. Ignoring the number on the front of the bus results in the inability to distinguish routes. The passenger becomes unable to choose which fare to pay, since no further control over the choice is possible. In the previous case, the passenger could control the fare by ensuring the appropriate bus was boarded. ■

Indexed external choice

The binary form of external choice can be generalized to an external choice between any finite number of processes. If I is a finite indexing set (which can be empty) such that P_i is defined for each $i \in I$, then it may be given an operational semantics as follows:

$$\frac{P_j \xrightarrow{a} P'}{\Box_{i \in I} P_i \xrightarrow{a} P'} \quad [j \in I] \qquad \frac{P_j \xrightarrow{\tau} P'_j}{\Box_{i \in I} P_i \xrightarrow{\tau} \Box_{i \in I} P'_i} \quad [j \in I]$$

where $P'_i = P_i$ for $i \neq j$.

In fact, the relationship between indexed external choice and binary external choice is captured by the following alternative definition of indexed external choice in terms of the binary operator.

$$\begin{aligned} \Box_{j \in \{i\}} P_j &= P_i \\ \Box_{j \in I \cup \{i\}} P_j &= (\Box_{j \in I} P_j) \,\Box\, P_i \end{aligned}$$

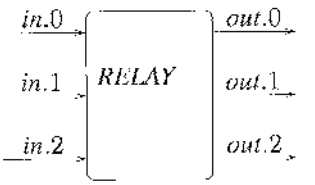


Fig. 1.8 The RELAY process

The external choice operator is associative, in the sense that $P_1 \,\Box\, (P_2 \,\Box\, P_3)$ and $(P_1 \,\Box\, P_2) \,\Box\, P_3$ have the same execution possibilities[2]. It is also commutative: $P_1 \,\Box\, P_2$ and $P_2 \,\Box\, P_1$ also have the same possible execution patterns. These two properties ensure that the order in which components are added to an indexed choice is irrelevant to the resulting behaviour of the choice. The only information required is the identity of the actual processes to be combined.

Example 1.4.3

$$RELAY \;=\; \Box_{i \in I}\, in.i?x : T \rightarrow out.i!x \rightarrow RELAY$$

This process describes a relay service between a number of channels of the form $in.i$ and $out.i$ where i is in some (finite) indexing set I. It is prepared to input a message x along any of the $in.i$ channels, and then output it along the corresponding output channel $out.i$.

Observe that this description has exactly the same transitions as the alternative description

$$RELAY2 \;=\; in?i?x : I \times T \rightarrow out!i!x \rightarrow RELAY2$$

The difference is in the intention of the designer. In the first case, the model is of a process with a number of channels of the form $in.i$ and $out.i$, each of type T. The picture of this process is given in Figure 1.8.

In the second case, the model is of a process with a single input channel and a single output channel of type $I \times T$. This is pictured in Figure 1.9.

An event of the form $in.2.7$ can be considered either as the message '7' on the channel $in.2$, or as the message '2.7' on channel in. The transition system treats these both as the same single event.

[2]Technically, they are *strongly bisimilar* (see [77])—any internal or visible transition that one process can perform can be matched by the other. This is what is meant in this chapter by two processes having the same execution possibilities.

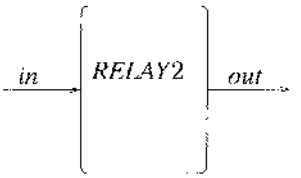


Fig. 1.9 The RELAY2 process

Observe, however, that *RELAY2* is well-defined even in the case where I is infinite, whereas *RELAY* is not well-defined in this case. This is because external choices $\Box_{i \in I} P_i$ are not permitted over infinite sets I. ■

Example 1.4.4 A mail system connects a set of nodes *NODE*. It may accept an input at any node l consisting of a destination and message $(d.m)$. This is captured as an input $in_l?(d.m)$. When there are such pairs $(d.m)$ in the system, then it may also perform $out_d!m$ corresponding to outputting message m at the destination node d.

This specification of a mail system may be described using indexed external choice within a mutually recursive definition. The *MAIL* processes are indexed by multi-sets (or bags), which maintain the number of copies of each element. A fresh copy is added each time a message is input, and one copy is removed when output occurs.

$$
\begin{aligned}
MAIL &= MAIL_{\{\}} \\
MAIL_{\{\}} &= \Box_{l \in NODE}\, in_l?(d.m) \rightarrow MAIL_{\{(d.m)\}} \\
MAIL_B &= \Box_{l \in NODE}\, in_l?(d.m) \rightarrow MAIL_{B \uplus \{(d.m)\}} \\
&\quad \Box \Box_{(d.m) \in B}\, out_d!m \rightarrow MAIL_{B \setminus \{(d.m)\}}
\end{aligned}
$$

where B ranges over non-empty bags, $\uplus$ is bag union, and $\setminus$ is bag subtraction. ■

Internal choice

A process considered as a specification describes a contract between the customer and the system designer. It encapsulates the behaviour of the system that is acceptable to the customer, and gives the designer the requirements that must be met.

The internal choice operator is commonly used as a specification construct. It is a choice over which the user has no control, and for this reason it is often called nondeterministic choice. The process

$$P_1 \sqcap P_2$$

pronounced 'P_1 internal choice P_2', describes a choice between P_1 and P_2, and the choice is resolved by the process itself, without any influence from its environment.

Transition rules given for a specification construct cannot completely characterize the nature of this construct, since they provide a particular approach to implementation. One way of implementing the choice construct is to resolve the choice immediately. This is accompanied by a silent transition, due to the state change from $P_1 \sqcap P_2$ to one of its components. Either of the choices is possible:

$$(P_1 \sqcap P_2) \xrightarrow{\tau} P_1$$
$$(P_1 \sqcap P_2) \xrightarrow{\tau} P_2$$

These rules describe an operational understanding of one way this choice could be implemented, though its use is more often as a specification construct. The process $P_1 \sqcap P_2$ is a process which is guaranteed to behave on any particular execution either as P_1 or as P_2. As a specification, if $P_1 \sqcap P_2$ describes the customer requirement then the implementer is free to provide either P_1 or P_2 for any execution and the customer will find either acceptable.

There are a number of ways a system designer might choose to provide a system which meets the specification $P_1 \sqcap P_2$:

- P_1 and P_2 could both be developed, and whenever the process is run then a coin is tossed to decide which one to provide.
- P_1 and P_2 could both be constructed, and whenever the process is run then resource considerations determine which one is provided.
- P_1 alone is provided.
- P_2 alone is provided.

These possibilities are illustrated in Figure 1.10. In each case a black box labelled with $P_1 \sqcap P_2$ is provided, but the implementations inside the boxes are different.

Example 1.4.5 A mail router program might offer one of two routes: $ROUTER = VIA\_A \sqcap VIA\_B$. Whenever this program is invoked, the choice is resolved at run-time internally by considering the network traffic. The user is not concerned with the route, but simply in the correct delivery of the message, and is therefore happy to devolve responsibility for making the choice to the *ROUTER* program. ■

Example 1.4.6 A bus company guarantees to provide buses between A and B, but does not guarantee any particular route. There are two routes, the 37 and the 111. The passenger is happy to accept either, so the service offered by $BUS\_37 \sqcap BUS\_111$ is acceptable. The bus

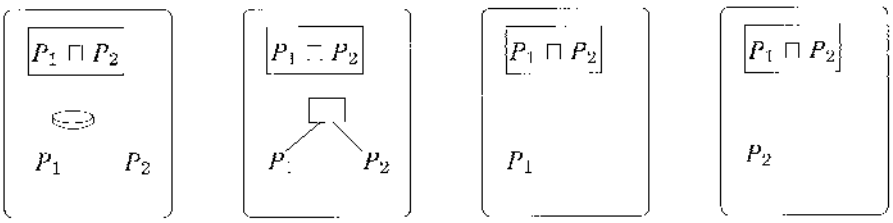


Fig. 1.10 Implementing $P_1 \sqcap P_2$

company decides to scrap the 37 bus service and run only the 111. This is indistinguishable to the customer from the situation where the decision to run the 111 in preference to the 37 is in fact made every morning. ■

Example 1.4.7 A customer who will accept a car of any colour must necessarily find a black car acceptable. A manufacturer who guarantees to provide a car meeting the specification $CAR_{black} \sqcap CAR_{coloured}$ may decide always to provide CAR_{black}. ■

Indexed internal choice

Since the internal choice operator corresponds to the disjunction operator as used in specification, it is natural to generalize it. If J is a set of indices (which may be finite or infinite[3], but must be non-empty) and P_i is defined for each $i \in J$, then the process

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

is a process which can behave as any of the P_i. As a specification this process describes the requirement that any execution should be appropriate to at least one of the P_i.

Operationally the indexed internal choice operator resolves immediately to one of its arguments:

$$\frac{}{(\sqcap_{i \in J} P_i) \xrightarrow{\tau} P_j} \quad [j \in J]$$

Example 1.4.8 The range of possibilities for a random number generator might be described by the infinite choice

$$\sqcap_{n \in \mathbb{N}} outIn \rightarrow STOP$$

[3]Technically, there is a given universal set of indices which contains J.

Any positive integer might be output. CSP does not express the probabilities of the numbers, it simply records the fact that they are possible. ■

Example 1.4.9 A process which can perform some event from the set of events A, but where its environment has no control over which, could be described as follows:

$$\sqcap_{a \in A} a \rightarrow STOP$$

A process D which can repeatedly perform some event from the set A could be defined recursively as follows:

$$D = \sqcap_{a \in A} a \rightarrow D$$

The choice can be resolved differently each time round the recursive loop. ■

Exercises

Exercise 1.1 Give suitable interface sets for the following. In each case you should decide the events that would be required in a description of how the process behaves.

1. A video recorder
2. A vending machine
3. An automated teller machine
4. A personal computer
5. A computer chip
6. A telephone answering machine
7. A multiplexor
8. An analogue to digital converter

Exercise 1.2 Write a CSP description of a square-root server with channels *in* and *out*.

Exercise 1.3 Write a CSP description of a multiplication component which has three input channels in_1, in_2, and in_3, and one output channel *out*. It reads in one number from each input channel (in any order) and outputs their product.

Exercise 1.4 Write a CSP description of a small fast-food outlet which serves only two items: burgers at 75p, and chicken at 95p. The sequence of interactions involves placing an order for

one of the items, paying for the order, and receiving the order. Only one customer at a time can be handled.

Exercise 1.5 Give the transition graph for process *PRINTER2* of Example 1.2.4.

Exercise 1.6 Give the transition graph for the process *HEATER* of Example 1.3.5.

Exercise 1.7 Give the transition graph for the process *COPY* of Example 1.3.2.

Exercise 1.8 What are the possible executions for $X = X$?

Exercise 1.9 Define a variant of the *COPY* process which accepts a value on its input channel, and stops if that value is 0, otherwise it outputs it and begins again.

Exercise 1.10 Define a process with an interface consisting of the events *press* and $finish.\mathbb{Z}$. It accepts a number of *press* events, and then outputs along the channel *finish* the number of *press* events that have occurred, after which it stops.

Exercise 1.11 Are the choices in the following processes internal or external?

1. A shop which offers discounts of 10%, 30% or 50% on sale items.
2. A cafe which offers *tea* or *coffee*.
3. A mail-order book company which offers the choice between sending back the form within two weeks, or receiving the book-of-the-month.
4. A lottery 'lucky dip' machine which gives any 6 numbers of 49 possibilities.

Exercise 1.12 Write a process which offers a choice between three bus routes.

Exercise 1.13 Write a CSP process which describes the choices presented by a sweet trolley containing two pieces of cheese cake, one piece of apple pie and one piece of chocolate cake.

Exercise 1.14 Write a process which describes the pattern of choices presented by the maze in Figure 1.11

1. if the alphabet is $\{east, west, north, south, in, out\}$;
2. if the alphabet is $\{left, right, forward, back, in, out\}$, where for example *right* means 'turn right and then move'.

Exercise 1.15 Give the transition graph of the process *SERVICE* of Example 1.4.1.

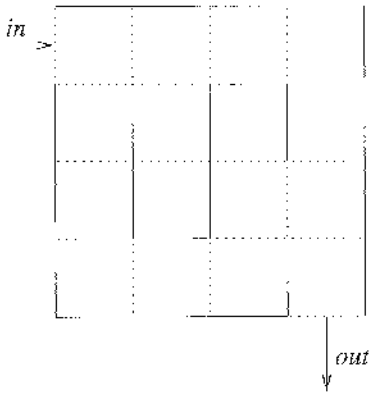


Fig. 1.11 A maze offering choices