4

Processes

Each process may be a primitive process indicating a single action; or it may be a composite process consisting of a number of definitions and simpler component processes bound together by process constructors, indicating a combination of the actions of its component processes. The structure of constructed processes is indicated by a prescribed layout of the source text, with each component appearing on a new line, slightly indented from the keyword that introduced the whole construction.

Processes that do nothing

The simplest of the primitive processes is `SKIP`, which is the process that does nothing at all. In many programming languages, you are obliged to write nothing (that is, not to write anything at all) if you want 'nothing' done; you will see later that `SKIP` serves as useful a purpose in occam as that of zero in the decimal notation for numbers.

The process `STOP` also does 'nothing', but unlike `SKIP` it never terminates. You can think of it as being what happens when something goes wrong, like the program becoming deadlocked, or some illegal operation. Nothing can happen in a sequential process after it has stopped, but things can happen in parallel with a stopped process. You might not expect to write the `STOP` process very often in your programs, but it is the rational thing to do when something unexpected happens, because it ensures that the part of a network of processes that has failed is brought to a standstill without affecting other processes, at least until they come to depend on the broken part.

It is also useful to have `STOP` around so as to be able to describe the effect of compound processes that 'go wrong', for example by becoming deadlocked. A process is said to be deadlocked if there is nothing which it is able to do next, but it has not finished properly. A parallel program which becomes deadlocked typically does so because each of its processes is waiting for one of the others to do something. For example, each component process may be prepared to communicate but only to do so with some of the other processes, and not with any process which is ready to communicate with it.

Sequential processes

In programs which execute sequentially, each process communicates with its successors by passing on a record of its state in variables. The work is done by assigning values to variables, and subsequently basing decisions on the values of those variables. The `occam` assignment has the form

```
variable := expression
```

Each expression has a value, and a type; each variable has a type. The type of the variable in an assignment must be the same as the type of the expression whose value is assigned to it.

A sequence of operations is described by writing them one under the other, under and slightly indented from the keyword `SEQ`. The sequence is executed by executing each of the components in the sequence in which they are written. Of course, if there are no components at all, then the sequence does nothing, and is just like `SKIP`. Thus,

```
SEQ
  x := 3
  y := x + 7
  x := x + 6
  z := x + 1
  x := (y + z) / 2
```

has the overall effect of setting each of the variables x, y, z to 10, albeit in a roundabout way, changing x three times.

Conditional processes

Decisions based on the values of variables are made by conditional processes. A conditional process consists of the keyword `IF` written above a list of components, each slightly indented. Each component is either another conditional nested within the first, or consists of a Boolean expression – the *condition* – and, below the condition and a little further indented, a process.

The whole conditional executes by looking down the list of components, and the components of the nested conditionals, until a condition is found whose value is `TRUE`. The process part of the first component with a

TRUE condition, and only that process, is then executed before the whole conditional terminates. It is wrong for there to be no TRUE condition, and in that case the conditional stops; of course, a conditional with no branches has no true conditions, and so is like STOP.

```
IF
  n < 0
    sign := -1
  n = 0
    sign := 0
  n > 0
    sign := 1
```

sets the value of the variable *sign* to one of minus one, zero, or plus one, according as the variable n has a negative, zero, or positive value.

Since it is defined that the selected process is the first in the order written of those corresponding to a TRUE condition, the process

```
VAL BOOL otherwise IS TRUE :
IF
  n = 0
    sign := 0
  n <= 0
    sign := -1
  otherwise
    sign := 1
```

describes exactly the same effect as the former example, but is much less clear. In general, it is good style to use constraints that describe as precisely as is convenient the conditions under which a process is to be executed. A good conditional has an exhaustive set of conditions, that is one of them is guaranteed to be TRUE; and the truth of a condition is a guarantee that executing the corresponding process would produce the right result.

There is a particularly common form of conditional which can be written more compactly to make a program easier to understand (and, incidentally, to give a hint to a compiler about how to translate it into efficient code). A conditional process selects a branch according to which of a range of constant values is taken by a single expression, for example

```
IF
  (a + b) = (-1)
    x := - a
  (a + b) = 0
    x := 0
  (a + b) = 1
    x := b
```

can be written more clearly as a case discrimination,

```
CASE a + b
  -1
    x := - a
  0
    x := 0
  1
    x := b
```

A case discrimination consists of an expression, the *selector*, written after the keyword `CASE`, followed by a slightly indented list of selections. Most selections consist of a comma-separated list of constant expressions of the same type as the selector, followed by an indented process. At most one of the selections can be a default selection, consisting of the keyword `ELSE` followed by an indented process.

The expressions in selections must all have distinct values, and a case discrimination executes by executing the process in the selection with an expression whose value is the same as that of the selector, if there is one. If there is no matching selection, a case discrimination executes by executing the process in its default selection if there is one; otherwise it stops. Notice that this makes an `ELSE` option different from a constant `TRUE` condition in a conditional:

```
CASE x
  ELSE
    p
  y, z
    q
```

would be the same if the branches were in a different order, and is the same as

```
IF
  (x = y) OR (x = z)
    q
  TRUE
    p
```

but if the first condition of a conditional were `TRUE`, then that branch would be guaranteed to be executed.

Parallel processes

Just as a list of actions can be described as happening in a strict sequence, so it is possible to specify that each of a list of actions is to happen, without specifying an order in which they must happen. Such a parallel composition is indicated by writing the actions one under the other, under and slightly indented from the keyword PAR. The parallel composition executes by executing each of its components until each has terminated. Of course, if there are no components, the effect is the same as SKIP; and if any of the components fails to terminate, for example by stopping, then the composition cannot terminate.

```
PAR
  x := y - 1
  z := y + 1
```

sets the values of x and z to be one less than, and one more than, respectively, the value of y. There is, however, no guarantee that the assignments will not happen in the other order, or indeed at precisely the same moment. Because of this, for a parallel composition to be legal, none of its components may change any variable which is used in any of the other branches: it would be an error, for example, to try to write

```
PAR
  x := y - 1
  z := x + 2
```

because x is used in the second component, but changed in the first component.

Since such mutual interference cannot happen, concurrent processes can only communicate by input and output over channels. In combination, an input and an output behave just like an assignment, except that the expression and variable are in different, concurrently executing processes. In particular, the assignment

```
variable := expression
```

is exactly the same as

```
PAR
  channel ! expression
  channel ? variable
```

provided that it is legal to write the latter in the particular context. Just as there are rules about the use of variables in concurrently executing processes, so also each end – the input end, or the output end – of a channel may be used in only one of the components of a PAR construction.

The rules about the use of channels and variables in parallel processes mean that the branches of a PAR are *disjoint*. In order to make the disjointness of processes mechanically checkable, there have to be quite severe restrictions on the use of arrays. If an array of channels or variables is shared by two or more branches of a `PAR`, and the indices used are not constants, the process is not valid unless the array is divided into non-overlapping segments by abbreviation, and only the abbreviations are used in the different branches.

Arbitrating processes

Decisions may also be distributed across several processes, using an *alternative*, which is like a conditional except in that the choice can depend on whether another process is executing an output. The simplest form of alternative is written with an `ALT` above an indented list of guarded processes; each guarded process is an input – a *guard* – followed by an accompanying indented process.

```
ALT
  up ? increment
    x := x + increment
  down ? decrement
    x := x - decrement
  read ? request
    reply ! x
```

executes by waiting until another process is ready to perform an output on one of the channels, *up*, *down*, *read*. The first of these offers to be made is taken up, and the corresponding input happens, followed by the accompanying process. If the inputs never become ready, the alternative is never completed. If several of the inputs are ready before the alternative is executed or if several become ready simultaneously, for example if several of them use the same channel, then any one – but only one – of the ready inputs is chosen.

In this example, the alternative acts as a guardian for the variable x which is effectively shared by the three (presumably) distinct processes which communicate on *up*, *down*, and *read*. The alternative arbitrates between them, making sure that their accesses to x happen in some sequence, so that only one of them is using the variable at any time.

It turns out that it is often handy to be able to shut out some of the branches of an alternative. A guard that starts with a Boolean condition and an ampersand sign is ready only if the Boolean is true and the input is ready.

```
ALT
  read ? request
    reply ! x
  NOT fixed & write ? change
    x := x + change
```

is a convenient abbreviation for a family of alternatives, in this case

```
IF
  fixed
    ALT
      read ? request
        reply ! x
  NOT fixed
    ALT
      read ? request
        reply ! x
      write ? change
        x := x + change
```

Of course, because the conditions in guards use only variables accessed by the alternative, their values cannot change while the alternative waits for one of the guards to become ready.

A guard in an alternative may also be `SKIP`, which is a guard that is always immediately ready. An alternative with a `SKIP` guard can always choose that guard without considering any of the others. For that reason, `SKIP` guards are almost always accompanied by a condition; indeed a helpful compiler may insist on a condition! `SKIP` guards are used to give an explicit default action in case all of the other branches are closed by false conditions,

```
ALT
  readable & read ? request
    reply ! x
  writable & write ? change
    x := x + change
  NOT (readable OR writable) & SKIP
    x := default.value
```

because if all of the guards in an alternative are closed by false conditions they will remain closed forever, and the alternative behaves like an alternative with no branches – that is, it behaves like `STOP`.

If a guard is an input on a channel with a discriminated protocol, you can use the short form of single-case discriminated input as a guard.

```
ALT
  condition & channel ? CASE tag; variable
    process
```

Notice that a guard which is a discriminated input is ready whenever another process is prepared to output, irrespective of what tag is going to be communicated. If the tag which is sent is not expected then the input stops, but only after the alternative has been committed to that branch.

The full form of a discriminated input takes the place of both the guard and the accompanying process. Both forms of discriminated input appear in

```
ALT
  readable & read ? CASE signal
    reply ! x
  writable & write ? CASE
    increment; i
      x := x + i
    decrement; d
      x := x - d
```

If *write* were a channel with a protocol that had a third tag distinct from both *increment* and *decrement*, and another process were to output that tag to *write*, then this example could opt for the second branch to the exclusion of the first. It would then, of course, stop because the received tag was not one of those present in the discriminated input.

Distinguish carefully between

```
ALT
  write ? CASE increment; i
    x := x + i
  write ? CASE decrement; d
    x := x - d
```

which can arbitrarily decide whether it will stop if sent an *increment* or if sent a *decrement*, and

```
ALT
  write ? CASE
    increment; i
      x := x + i
    decrement; d
      x := x - d
```

which guarantees to accept either tag. The former process is hardly ever what you would want.

Just like conditionals, the other thing that can appear as a branch of an alternative is a nested alternative. An alternative with nested alternatives behaves as though all of the branches of the nested alternatives were written as branches of the outer alternative. That is to say, `ALT` is associative.

Loops

There are two kinds of loop in `occam`: unbounded `WHILE` loops, and indexed, bounded `FOR` loops. Unbounded loops are necessarily sequential, whereas bounded loops need not be.

An unbounded loop is written with the keyword `WHILE`, followed by an expression (the condition), with a process (the body) below and slightly indented from it. It executes by testing the value of the condition and then, provided that its value is `TRUE`, executing the body. When the body has terminated, the condition is re-tested, so that the body is executed a number of times, in sequence, for as long as the condition remains `TRUE`. The whole `WHILE` loop terminates when the condition is tested and found to be `FALSE`.

```
SEQ
  x, v[n] := 0, key
  WHILE v[x] <> key
    x := x + 1
```

sets x to be the index of the first variable in the array v which contains the value *key*, by first posting a sentinel at $v[n]$.

Arrays of processes

A bounded loop may be thought of as being an array of processes, and can be made with any of the `SEQ`, `IF`, `PAR`, and `ALT` constructors, by putting a replicator of the form

```
name = base FOR count
```

after the keyword, and then writing just a single component (of the kind appropriate to the construction) below and slightly indented from the keyword. The *base* and *count* are expressions of type `INT`, and the meaning of such a `FOR` loop is the same as that of a construction formed from the same keyword followed by *count* copies of the component with the *name* taking on the values *base*, *base* + 1, ..., *base* + *count* − 1 in successive copies. In effect, the replicator declares *name* to be an `INT` constant with the body of the loop as its scope.

A `FOR` loop stands for a repetition of the constructor with which it is made. In the same way that

$$\prod_{year=1280}^{1341} entity_{year}$$

stands for the multiplication of sixty-two values,

$$entity_{1280} \times entity_{1281} \times \ldots \times entity_{1341}$$

so too the `SEQ`–`FOR` loop

```
SEQ year = 1280 FOR 62
  celebrate.easter(year)
```

stands for the sequential composition of sixty-two processes

```
SEQ
  celebrate.easter(1280)
  celebrate.easter(1281)
  ...
  celebrate.easter(1341)
```

or, more precisely,

```
SEQ
  VAL INT year IS 1280 :
  celebrate.easter(year)
  VAL INT year IS 1281 :
  celebrate.easter(year)
  ...
  VAL INT year IS 1341 :
  celebrate.easter(year)
```

So `SEQ`–`FOR` loops are quite like `FOR` loops in languages like ALGOL or PASCAL; but notice that you may not assign to the loop index, and it is not declared outside the body of the loop.

The bodies of parallel `PAR`–`FOR` loops are executed concurrently, so such loops are arrays of parallel processes. A conditional loop, written with `IF` and `FOR`, performs a bounded search, so

```
SEQ
  v[n] := key
  IF i = 0 FOR n+1
    v[i] = key
      x := i
```

has precisely the same effect as that of the `WHILE` loop on page 34. The same search can be performed rather more elegantly, without the use of a sentinel, by writing

```
VAL BOOL otherwise IS TRUE :
VAL INT not.found IS n :
IF
  IF i = 0 FOR n
    v[i] = key
      x := i
  otherwise
    x := not.found
```

Here, x is set to *not.found* precisely when there is no occurrence of the value *key* in the array.

In practice, a more useful result from the search would be a Boolean indicating whether or not the *key* was found.

```
VAL BOOL otherwise IS TRUE :
IF
  IF i = 0 FOR n
    v[i] = key
      found, x := TRUE, i
  otherwise
    found := FALSE
```

You should now see the reason for allowing conditionals as components of conditionals! Although you can (and should, where it helps) use redundant nesting to guide a reader through the structure of a large construct, it is necessary in cases like the last example, where one of the branches is a loop. The same applies to alternatives as components of alternatives.

Alternative loops, written `ALT`–`FOR`, give a process an opportunity to listen to an array of channels, for example

```
WHILE TRUE
  ALT i = 0 FOR SIZE c
    INT n :
    c[i] ? n
      shared ! i; n
```

gathers integers from the channels of c and passes them on, with an identifying index, to the *shared* channel.