7

Configuration

Because it was intended as a medium for programming embedded systems, occam includes facilities for describing the way in which the abstract data objects of the program are related to the concrete hardware. Most of the explanations in this chapter have to be in terms of what an implementation would do, and how it would do it. On the whole, you would not expect to have to use the material here unless and until you needed to make your program fit some particular piece of hardware. You may want to pass by this chapter on a first reading.

Biased alternatives

Recall that alternative processes are symmetric: each of the branches of an alternative is given the same chance of being selected. If a number of guards are ready when the alternative is executed, any one of them may be selected.

There are circumstances in which this is inconvenient; in a process

```
WHILE TRUE
  ALT
    interruption ? CASE signal
      unusual()
    INT message:
    input ? message
      usual(message)
```

the intention is that receipt of a signal from *interruption* should cause the execution of the *unusual* procedure, whereas normally a stream of messages from *input* are dealt with by the *usual* procedure. But consider what would happen were the process writing to *input* producing new messages in less time than it would take to execute *usual*: each time the alternative was executed the *input* guard would be ready. Irrespective of the readiness of the *interruption* guard, it would be open to the alternative to choose the *input*, and a ready *interruption* could be ignored indefinitely. Notice especially that this is nothing to do with the order in which the branches of the alternative were written: an alternative is symmetric, and the same argument would hold were the branches written in the other order.

There is, for precisely this reason, an asymmetric alternative process, written with a PRI qualifier. (Some people read this, it appears, as *prioritized*!) An asymmetric alternative favours its earlier branches – those nearer the top of the paper – over the later ones. Like the symmetrical construct, it waits until some of its guards are ready and then selects one for execution, subject to the additional constraint that no branch can be selected which is later in the text than that which first became ready.

In particular, if some of the guards of an asymmetric alternative are ready when the alternative is executed, no branches later than these ready ones can be selected. In the example above, this is just what is needed:

```
WHILE TRUE
  PRI ALT
    interruption ? CASE signal
      unusual()
    INT message:
    input ? message
      usual(message)
```

is obliged to service a signal from *interruption* after the completion of at most one execution of the body of the loop after the signal is sent.

Check that you understand that an implementation of occam would be allowed to ignore the PRI if it implemented *all* alternatives as if they were asymmetric. An implementation must guarantee the asymmetry of a PRI ALT, and a programmer can rely on that; an implementation is free to implement any selection strategy that meets the specification of a symmetric alternative, and a programmer can only rely on the existence of some strategy. There is no guarantee of the fairness of an alternative which happens to have the freedom to make a choice.

Beware the temptation to over-use PRI!

Priority on a single processor

If several branches of a parallel process are to be executed by a single processor, it may be that one of them has to wait for another not because of any communication between them, but because they each need the processor. This is a hidden shared resource, but it does not usually matter because the guaranteed disjointness of parallel processes in occam gives the programmer the illusion that every process has its own machine with its own store (variables) and processor.

The illusion breaks down when the programmer is allowed to measure the performance of a program. There are times when it really matters how long it takes to execute a piece of code, or how long it takes to get a response to an attempt to communicate with a program. One process being executed by a processor shared with another process can be unexpectedly delayed by the other, in a way that cannot be predicted simply by studying the code of the program.

Usually, where performance matters, there is some small collection of processes in a program that must be given priority. There may, for example, be signals from a piece of hardware external to the program to which a response must be given in a bounded time. In checking that a sequential program could achieve the required response time, the programmer – or better still, the compiler – could count the instructions to be executed. The count of instructions executed by a process in a parallel program, however, is only a lower bound on how long it takes to be executed. To achieve that bound the process has to be guaranteed access to the processor when it is needed.

A parallel process is symmetric; the order of the branches is immaterial. Sequential, single-processor implementations are allowed to make progress in executing it by executing any one of the branches that are not blocked waiting for a communication. On the other hand, an asymmetric parallel process, introduced by `PRI PAR`, favours the textually earlier branches – those nearer the top of the paper – over the later ones. If there are two branches that are not blocked, the earlier one must be executed; if an earlier branch becomes unblocked while a later one is executing, the implementation must rapidly switch to executing the higher priority branch.

Once again, an implementation of `occam` would be allowed to ignore the `PRI` if it implemented *all* parallels as if they were asymmetric. An implementation must guarantee the asymmetry of a `PRI PAR`, and a programmer can rely on that; an implementation is free to implement any execution strategy that meets the specification of a symmetric parallel, and a programmer can only rely on the existence of some strategy. There is no guarantee of the fairness of a parallel which happens to have the freedom to make a choice.

Notice that the priority is ascribed dynamically to a piece of code. It is possible for the same piece of code, the procedure p in this example

```
PROC p(...)
  ...
:
PRI PAR
  p(...)
  p(...)
```

to be executed with several different urgencies in the same program.

An implementation will normally impose quite a small limit on the number of branches in an asymmetric parallel construction; on present transputer implementations, the limit is two. In a parallel like

```
PRI PAR
  PAR
    incoming.router(...)
    outgoing.router(...)
  PAR
    user.routine(...)
    background.task(...)
```

the high priority branch encompasses both *—.router* processes; the low priority processes are the *user.routine* and the *background.task*. Within each pair, there is no priority given to the one process over the other.

Once again, beware the temptation to over-use `PRI`! Like an exclamation mark at the end of a sentence, the `PRI` qualifier should be used only when it is necessary. If one exists, the algorithm that solves a problem without requiring asymmetric constructs is likely to be simpler and easier to understand than one which relies on asymmetry.

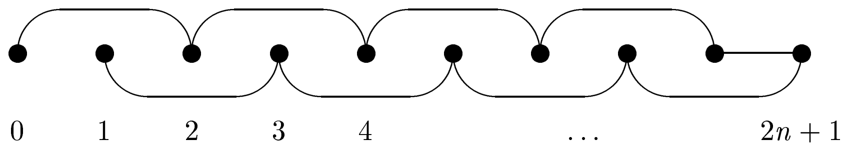
Distributing a process over several processors

If an occam program is to be executed by a multi-processor machine, it is the programmer's responsibility to decide which processor executes which processes. This is in keeping with the philosophy that the programmer should be in control, knowing best what performance measure is important. This mapping of text to processors has no effect on the behaviour of the program, apart from deciding where in a machine a particular variable or channel will exist, or altering the time it takes to execute the program.

The outermost parallel constructor of a program can be annotated `PLACED`, meaning that each branch is to be executed by a different processor. Each processor will be allotted an `INT` identifier in a very machine-dependent way, and the part of the program text which is to be executed by a particular processor is announced by a `PROCESSOR` line with that processor's number. Suppose that the program

```
[(2*n)+1]CHAN OF INT16::[]BYTE c :
PAR
  inpull(c[0])
  PAR i = 0 FOR 2*n
    wurdle(c[i], c[i+1])
  outpush(c[2*n])
```

is to be executed on a linearly indexed array of processors that are connected each to its two second nearest neighbours.



This could be done by grouping the $2n$ processes that *wurdle* into an odd-numbered n and an even-numbered n.

```
[(2*n)+1]CHAN OF INT16::[]BYTE c :
PLACED PAR
  PROCESSOR 0
    inpull(c[0])
  PLACED PAR i = 0 FOR n
    PROCESSOR (2*i) + 2         -- even numbers ascending
      wurdle(c[i], c[i+1])
  PLACED PAR i = n FOR n
    PROCESSOR ((4*n)+1) - (2*i) -- odd numbers descending
      wurdle(c[i], c[i+1])
  PROCESSOR 1
    outpush(c[2*n])
```

The behaviour of this program is indistinguishable from the original, except that each instruction of the code has been allocated to a specific processor.

Variables and timers may not be declared 'outside' a `PROCESSOR`, so only constants, channels, and procedures and functions can be shared by different processors. If executed on a machine constructed from transputers, the channels shared by branches of a placed parallel would be implemented by the physical serial links between the transputers. The mechanism for associating channels with links is described later in this chapter.

In any particular implementation of occam it will usually be necessary to include other information about each processor: for example current transputer implementations require an indication of which type of transputer is to execute the code. Since branches of a placed parallel can be executed by different types of processor, it is perfectly possible for the 'natural' integer type `INT` to have a different range of values in different branches. Not only does this mean that it would be unwise to try to communicate `INT` values between different branches, but it is also necessary to be careful of algorithms that use expressions like `MOSTPOS INT` which might have different values in different parts of the same program. Beware, also, the replicator index of a replicated placed parallel; it is of an `INT` type that may not have the same range of values as the natural `INT` of any of the branches.

Concrete representation of data

A value of one type which is represented by a certain bit pattern can be viewed as if it were the value of another type that would be represented by the same bit pattern, provided the representations are of the same size.

For example, although `#3FF(INT64)<<52` is an `INT64`, whose value is about 4.6×10^{18}, its representation consists of the same sixty-four bits as that of the double-length floating-point number `1.0(REAL64)`. This means that the retyping conversion

```
VAL REAL64 one RETYPES #3FF(INT64) << 52 :
```

is the same abbreviation as

```
VAL REAL64 one IS 1.0(REAL64) :
```

There is an important difference between retyping and type coercion: the `REAL64` with the same representation as an `INT64` is nothing like the `REAL64` of about the same value, in this case

```
VAL REAL64 huge IS REAL64 ROUND (#3FF(INT64) << 52) :
```

which would be approximately the same as `4.6E+18(REAL64)`.

A retyping variable abbreviation is similar; it gives a distorting window through which to view the variable as if it were a variable of a given type, so long as that type has the same sized representation as the type of the variable. For example,

```
INT64 register.dump :
...
[8]BYTE register RETYPES register.dump :
```

allows the program to read and write individual bytes of the `INT64` by indexing the array. The same rules apply to the use of names of retyped values and variables as apply to abbreviation: it is invalid to refer to a retyped variable by its original name; it is invalid to change a variable in a retyped expression.

In general, the meaning of a retyping is allowed to depend on the implementation. For example, different implementations of occam could quite reasonably interpret

```
PAR i = 0 FOR 8
  register[i] := BYTE i
```

as setting the integer *register.dump* either to `#0001020304050607(INT64)` or to `#0706050403020100(INT64)`. Other values are not ruled out either, but an implementation is required to be consistent with itself.

There may be circumstances in which an occam program needs to communicate with a process that is implemented in hardware, and which communicates sequences of bit patterns for which there is no description as a protocol. For example, something like a disk controller may send to the program various sized blocks of data from its disks along a serial link. Such a communication requires that the link be represented by a channel with the anarchic protocol

```
CHAN OF ANY from.disk :
```

Inputs from a channel of this protocol can be made into a variable of any type, not necessarily all of the same type. Each communication is treated as though it were an array of bytes, and is retyped, as if by a `RETYPES` conversion, to be of the type of the variable. All that is necessary is that each input is into a variable represented by the right number of bytes for the value communicated. Similarly, outputs to a channel with an anarchic protocol transmit the byte array which retypes whichever value, of whatever type, is presented for output.

Notice that the use of anarchic channels effectively suppresses checking; if a process makes an anarchic input into a small variable, and receives a larger array of bytes than will fit, anything might happen. The process must be able to determine in some other way what type or size of variable to use. Anarchic communication should only rarely be used on channels between two occam processes, rather than between one software process and one hardware process. Indeed, the only circumstance in which internal anarchic channels are reasonable is when simulating a piece of hardware for testing the process which will drive it.

Attaching a program to hardware

If a program is to be executed on a particular machine, with no operating system to insulate the programmer from the machine's peculiarities, there has to be some way of referring to the parts of the hardware from within the program. The usual thing is to associate each type of hardware object with a corresponding type of abstract object in the program.

The store of each processor is treated as though it were an `[]INT`, and each addressable object will correspond to an element or elements of this array. For example, the serial links of a transputer are located at specific addresses at the bottom of memory; say that one of them corresponds to the $link^{\text{th}}$ location in the memory, where

```
VAL INT link IS ... :
```

A channel, say a channel of counted arrays of characters,

```
CHAN OF INT::[]BYTE c :
```

that is to be implemented by that particular link from a transputer would be placed

```
PLACE c AT link :
```

In the same way, timers can be placed at particular addresses that correspond to hardware real-time clocks.

It is also possible to place variables at specific hardware addresses, although this should not be done in a way that invalidates the anti-aliasing rules. Placing of variables and arrays of variables is allowed principally to let the programmer decide which variables will lie in different types of store. There is, for example, a large performance benefit obtainable from placing heavily used variables in the internal store of a transputer, rather than in the relatively slower external store.

There is an additional abstraction for describing peripherals which are to be handled by the processor's reading and writing particular locations in store. Because accessing a store address which is mapped onto a peripheral device, even if the access is only for reading, can change the state of the peripheral it would be dangerous to treat the peripheral as if it were a variable. An occam implementation is free to access the variables referred to in expressions in any order, even several times as often as they occur in the expression. It would not be possible to explain the behaviour of a program in which the variables changed their values without their being assigned, at least not in terms of the abstraction of the variable.

A *port* of a given type, placed at a given location in store, is a mechanism for controlled reading and storing of values of that type at that location. A port

```
PORT OF [4]BYTE uart :
PLACE uart AT ... :
```

can be used in two sorts of process:

```
uart ? buffer
```

reads four bytes exactly once from the location of the port, and stores them in the `[4]BYTE` variable *buffer*;

```
uart ! table
```

writes the `[4]BYTE` value *table* to the location of the port, writing to it exactly once.

Ports and arrays of ports and so on can be abbreviated, placed, and passed as parameters in the same way as channels.