Appears in *Parallel Programming of Transputer Based Machines*, IOS, September 1987

On Guards

Geraint Jones
Oxford University Computing Laboratory
11 Keble Road
Oxford OX1 3QD, UK

1 September 1987

Abstract

This is a note about the distinction in *occam* between inputs and outputs, and between those inputs that appear as guards and those that appear as processes in their own right. It explains the restriction that outputs may not appear as guards in *occam* programs, and records programming techniques for coping with this restriction.

An *occam* programmer will from time to time find himself complaining that something would be so much easier if *occam* allowed output guards. There is an apparently arbitrary asymmetry in *occam*'s allowing inputs to guard alternatives, but forbidding outputs.

First I explain why output guards are forbidden in *occam*. Essentially, it is very expensive to implement general input and output guards. There are then presentations of systematic ways of eliminating output guards from a CSP program as a way of translating it into *occam* without incurring the prohibitive costs of general guards. There is no claim that the techniques are new, but experience at previous meetings suggests that they are not well enough known.

Conditional communications

Ignore, for the moment, the obvious difference between inputs and outputs. Apart from the data transfer both parts of a communication are the same, in that neither can happen until the other has happened. Nothing 'after' an input or output can happen until it is also possible for things 'after' its peer to happen. Communication is therefore a means of synchronizing two parallel processes, and viewed solely as synchronization there appears to be no difference between input and output.

There is a distinction which must be made, however: between those communications which appear as guards (which I call *conditional*) and those which are not guards (which are *unconditional*). Strictly speaking, the guard of an `ALT` which contains only one guard can also be treated as unconditional, because

```
ALT
  g
    p
```

is the same as

```
SEQ
  g.
  p
```

The important thing about unconditional communication is that once an unconditional communication has become possible, it must eventually happen. With conditional communications, several become possible together but only one of them will actually happen. A decision has to be made about which one of those possible conditional communications will happen. Not until that decision has been taken is any of them committed to happening.

Now consider two physically separated processes, perhaps processes running on distinct transputers. If each tries to communicate with the other by a conditional communication, then they must both make the same decision about whether they are to communicate. Any attempt to make the decision independently at each process is likely to lead to a disagreement. Although at any time it may appear that a distant peer is prepared to communicate, it may already have committed itself to not doing so. The problem is like that of synchronizing clocks that are separated by distances sufficiently great that relativistic effects are significant.

Naïvely, we might nominate a single process to make all the decisions. Any process which requires to communicate must first ask this global arbiter for permission to proceed. Such a solution becomes more difficult and expensive to implement as the system of processes becomes larger, and the arbiter becomes a bottleneck. Various better schemes (eg [1], [2]) are known, which make the decisions more locally.

Essentially any local solution has to provide an additional process to mediate between any pair of processes that might communicate. This broker process holds successively stronger pledges from each party until either they are both committed or one of them pulls out. The problem with these schemes is the expense: an additional process and several communications with it for each bargain that needs to be struck. Reference [6] contains an excellent survey of these techniques, and a discussion of more efficient solutions which exist in certain restricted cases.

By imposing restrictions on the ways in which conditional communications can legally be used in programs, processes can make stronger assumptions about the behaviour of a peer in the course a negotiation. Under these stronger assumptions the negotiation can be more efficient.

Banning output guards

The solution adopted in *occam* is to ensure that no pair of conditional communications ever meet. Whenever a pair of communications match, at least one is guaranteed to be unconditional. Under this condition there is a simple and distributable implementation of the synchronization.

An unconditional communication, as soon as it becomes possible, makes an offer to its peer. A process receiving an offer knows that the offer can not be withdrawn, that the offerer is already committed to the communication. An `ALT` making a number of conditional communications can safely make its own decision of the basis of the offers which it holds. It chooses one of the peers that has made an offer, commits itself to that peer and makes a reciprocal offer. Any other spare offers are held for possible future use. A process which has both made an offer, and received one, knows that both it and its peer are irrevocably committed to communicating, so can complete the exchange.

Any restriction will do, that will guarantee that two conditional communications can never be matched. In *occam*, output guards are forbidden, and each matching pair of communications consists of an input and an output. This means that in each pair of communicating processes there is an output which is necessarily unconditional. It would be equally possible to forbid input guards so that each pair of communications included an input which would be unconditional. I suspect that programming without input guards would be slightly less natural than programming without output guards.

More generally, both input and output guards might be allowed, provided only that a compiler could statically determine that no communication was between a pair of guards. Such a regime would even work in a language[4] in which there was no distinction between input and output, where data were transferred in both directions at each synchronization. Realistically, whether a channel parameter were used conditionally should probably be declared in interfaces such as the parameter list of a `PROC` declaration (as should whether the channel is used for input or output) to partition the checking. Whether occasionally being frustrated by this more liberal but more subtle restriction would be more annoying, I can only guess.

The simplest and most natural of these solutions would appear to be that adopted in *occam*. The rest of this paper is about coping with this restriction. The restriction may not be as severe as it at first appears since, as observed in [3], some programs which might have been written with output guards are indistinguishable in their execution from others in which there are none. The archetypical example is that

```
ALT
  c!x
    d!y
  d!y
    c!x
```

is equivalent to

```
PAR
  c!x.
  d!y
```

Output guards with unconditional inputs

The output guards can always be eliminated from a program which could have been written in a language with output guards but no input guards. More generally, if a program can be partitioned into components each of which can work under a regime either of output guards only or of input guards only, and there is no conditional communication across component boundaries, then this method will eliminate the output guards.

Consider an array of consumers that periodically become committed to input, and then are busy for a while before becoming ready again. A stream of tasks is to be read from a channel and doled out to these consumers, first come first served. Using output guards this would naturally be written as in Figure 1. In cases such as this, it is simple to separate the data transfer from the synchronization: if each pair of communications on the relevant channels `c!x` in parallel with `c?y` is replaced by

```
SEQ
  c!ANY
  c!x
```

in parallel with

```
CHAN c[n]:
PAR
  WHILE TRUE            -- distributing process
    VAR x:
    SEQ
      in?x
      ALT i = [0 FOR n]
        c[i]!x          -- issue the next task
          SKIP

  PAR i = [0 FOR n]
    WHILE TRUE          -- consumer(i)
      VAR y:
      SEQ
        c[i]?y          -- receive a task
        -- do something with y for a while
```

Figure 1: task distribution using output guards

```
SEQ
  c?ANY
  c?y
```

then nothing is changed about the external behaviour of the program.

There are no variables in the outputting process which can change their values between the pair of outputs, so the same value is transferred as in the original program, and on the same channel. Similarly no variables in the inputting process change their value between the pair of inputs, so it is to the same variable that input is made, and from the same channel.

The state of each process between the two parts of these sequences of communications is transient. Because the second communication is in each case unconditional it will definitely happen once the first has done. Of course, had the original communication been a guard, the rewritten sequence would have to begin with a conditional communication, so I mean that you should rewrite a process with output guards

```
ALT i
  c[i]!x
    p(i)
```

as

```
ALT i
  c[i]!ANY
    SEQ
      c[i]!x
      p(i)
```

.

Since communications on `c` now always happen in pairs, it can do no harm to separate them onto two different channels. For each affected channel `c`, a new channel `d` is declared with the same scope and a name not used in that scope. The first communication of each pair will happen on this new channel, so instead each pair of communications on `c` write

```
CHAN c[n], d[n]:
PAR
  WHILE TRUE            -- distributing process
    VAR x:
    SEQ
      in?x
      ALT i = [0 FOR n]
        d[i]?ANY        -- accept the next request
          c[i]!x        -- and serve it

  PAR i = [0 FOR n]
    WHILE TRUE          -- consumer(i)
      VAR y:
      SEQ
        d[i]!ANY        -- request a task
        c[i]?y          -- and receive it
        -- do something with y for a while
```

Figure 2: task distribution without output guards

```
SEQ                    SEQ
  d!ANY  in parallel with  d?ANY.
  c!x                    c?y
```

Each of the communications on `d` is a simple synchronization; but because of the symmetry between inputs and outputs used only for synchronization, there is no reason why this communication should be implemented either way around rather than the other, so write instead

```
SEQ                    SEQ
  d?ANY  in parallel with  d!ANY.
  c!x                    c?y
```

There are now only input guards left, and the work is done. The example program would have been rewritten as in Figure 2.

In summary, translate

```
                                        CHAN c, d:
    CHAN c:                             PAR
    PAR                                   SEQ
      SEQ                                   ...
        ...                                 d?ANY
        c!x                                 c!x
        ...              ⟹                  ...
      SEQ                                 SEQ
        ...                                 ...
        c?y                                 d!ANY
        ...                                 c?y
                                            ...
```

The cost of this transformation is an extra channel for each channel which was used for conditional output, and an additional communication for each output on such a channel. There is no extra cost for any other communication, and nothing has changed about the observable behaviour of the program apart perhaps from execution time.

Being simultaneously ready for input and output

The previous simple method will not work in case there is a cycle of odd length in the program, and each process in the cycle makes conditional communications with both its predecessor and its successor. An example of this is a routing mechanism made of a ring of processes of the form

```
PROC node(VALUE here, CHAN net.in, env.in, env.out, net.out)=
  ...
CHAN net[n]:
PAR i = [0 FOR n]
  node(i, net[i], send[i], receive[i], net[(i+1) MOD n])
```

where each node inputs from a (local) `send` channel, and by forwarding messages through a succession of `net` channels delivers an output at a `receive` channel. Reference [7] observes that this is free of deadlock provided each node is ready to accept input from its `net.in` unless it has just done so and not subsequently sent output. Clearly `node` must sometimes be prepared to communicate either on `net.in` or on `net.out`.

Because the nodes are connected in a ring, the `net.in` and `net.out` channels must be used one for input and one for output. If there are an odd number of nodes in the ring, the previous transformation cannot be used to eliminate output guards. Since no sequential process can be both ready for input and for output (no `ALT` can have both input guards and output guards) the `node` process must contain at least two processes in parallel. I will describe a way of developing them by eliminating output guards.

The specification of a `node` is that it must behave more or less like a 'long enough' buffer:

- the sequence of messages appearing as output on `net.out` and `env.out` must be a prefix of the sequence appearing as input to `net.in` and `env.in`;
- anything which has been input must eventually appear as output on the correct channel;
- input must be accepted on `net.in` unless the last communication was input from `net.in`.

If you abstract from the network just the sequence in which the communications happen, ignoring for a moment which values are communicated, then a node has to have the following form. Write *net.in* for an input from the predecessor via `net.in`, *env.in* for an input from `env.in`, *out* for an output either to `net.out` or the `env.out` as required by the data,

$$\begin{aligned} NODE &= env.in \rightarrow FULL \mathbin{[]} net.in \rightarrow out \rightarrow NODE \\ FULL &= out \rightarrow NODE \mathbin{[]} net.in \rightarrow out \rightarrow FULL \end{aligned}$$

then *NODE* is a satisfactory implementation of `node`. As a step towards turning the recursion into loops, these equations can be transformed into

$$\begin{aligned} NODE &= (env.in \rightarrow FULL' \mathbin{[]} net.in \rightarrow out); NODE \\ FULL' &= out \rightarrow SKIP \mathbin{[]} net.in \rightarrow out \rightarrow FULL' \end{aligned}$$

Translating $FULL'$ into *occam*, a problem occurs with

$FULL'$ =

```
VAR full:
SEQ
  full := TRUE
  WHILE full
    body
```

$$BODY = out \rightarrow DONE \mathbin{[]} net.in \rightarrow out \rightarrow SKIP$$

$DONE$ = `full := FALSE`

The problem is that *BODY* would translate into an `ALT` with a branch guarded by *out* which is an output.

Notice that *BODY* is committed eventually to doing an *out*, and that it remains ready to do an *out* until that *out* has happened. I shall say that the *out* is *inevitable* once it has become possible. Replace each occurrence of an inevitable event, e, by a pair of new events one of which happens each time that e becomes possible and the other of which happens after the completion of e.

Replacing *out* by *fwd* (for forward) followed by *ack* (for acknowledge), I get

$$BODY' = fwd \rightarrow (ack \rightarrow DONE \mathbin{[]} net.in \rightarrow ack \rightarrow SKIP)$$

Notice that the *fwd* in $BODY'$ is unconditional, and that if *ack* is an input in $BODY'$, then all conditional events are inputs.

Where $BODY$ would have performed an *out*, $BODY'$ is able to do a *fwd*, and having once done a *fwd*, an *ack* is inevitable. To guarantee that an *out* event happens at the same place as in $BODY$, all that needs to be done is to sandwich it between the *fwd* and *ack* events. A process is needed which makes an *out* possible as soon as *fwd* happens, and an *ack* possible only when the *out* has happened

$$BUFF \quad = \quad fwd \rightarrow out \rightarrow ack \rightarrow SKIP$$

Then if $BODY'$ is run in parallel with $BUFF$, with the *fwd* and *ack* events hidden,

$$\begin{aligned} BODY' \parallel BUFF \quad = \quad & out \rightarrow DONE \\ [] \quad & out \rightarrow net.in \rightarrow SKIP \\ [] \quad & net.in \rightarrow out \rightarrow SKIP \end{aligned}$$

Substituting $BODY' \parallel BUFF$ for $BODY$ makes the $NODE$ less determined: it can arbitrarily choose to stay $FULL$, so it can sometimes delay acceptance of $env.in$ inputs that would have been accepted by the original program. The specification was, however, sufficiently liberal that the new process still meets it.

There are no output guards left so the translation can be completed. It was by this method that I arrived at the coding of `node` in Figure 3. This `node` makes a ring which causes the output `receive[i]!j` to happen after each `send[j]!i`.

The cost of this rewriting is greater than that of the previous one. Each affected communication has been replaced by three communications. each affected channel has required an additional process and two additional internal channels. The behaviour of the program has been changed, although in this case keeping within its specification. In summary, try translating

```
ALT
  c!x
    p
  g
    SEQ
      q
      c!x
      r
```

$\Longrightarrow$

```
CHAN d, e:
PAR
  SEQ
    d!x
    ALT
      e?ANY
        p
      g
        SEQ
          q
          e?ANY
          r
  VAR x:
  SEQ
    d?x
    c!x
    e!ANY
```

and then see what changes have been made to the meaning of the program.

```
PROC node(VALUE here,
          CHAN net.in, env.in, env.out, net.out) =

  CHAN fwd, ack:

  PROC out(VALUE from, to) =
    IF
      to = here
        env.out ! from
      to <> here
        net.out ! from; to :

  WHILE TRUE            -- NODE
    VAR from, to:
    ALT
      env.in ? to
        VAR full:
        SEQ
          from := here
          full := TRUE
          WHILE full            -- FULL
            PAR
              SEQ                         -- BODY'
                fwd ! from; to
                ALT
                  ack ? ANY
                    full := FALSE
                  net.in ? from; to
                    ack ? ANY

              VAR from, to:               -- BUFF
              SEQ
                fwd ? from; to
                out(from, to)
                ack ! ANY

      net.in ? from; to
        out(from, to)
```

Figure 3: `PROC node` coded in *occam*

```
PROC buffer(CHAN in, out) =
  VAR head, tail, count, queue[n] :
  SEQ
    head := 0     -- index of head of queue
    tail := 0     -- index of tail of queue
    count := 0    -- number of items in queue
    WHILE TRUE
      ALT
        count < n & in ? queue[tail]
          PAR
            tail := (tail + 1) \ n
            count := count + 1
        count > 0 & out ! queue[head]
          PAR
            head := (head + 1) \ n
            count := count - 1
```

Figure 4: `n`-place buffer with output guards

As a second example, this same method can be used to turn the obvious coding of a multi-place circular buffer, Figure 4, into a legal occam program. In this case, the reasoning is that as soon as `count` has been incremented the first time, an output of `out!queue[0]` is inevitable. Each subsequent output on `out` becomes possible (and inevitable) as soon as its predecessor has happened, provided that `count` is not then zero.

The program arrived at, after a bit of tidying up, is shown in Figure 5. This latter program is an `n+1` place buffer, rather than an `n` place one, so the change in behaviour can this time be accommodated by decreasing `n`.

Discussion

This note has been about writing programs without output guards to do things that might have been easier were output guards allowed. The question naturally arises of what sorts of things are difficult or impossible without output guards. Certainly, it is not possible to implement in *occam* a ring consisting of `node`'s which behave exactly as described by *NODE*. On the other hand I was working to a more liberal specification and was able to implement that.

Programs can certainly be constructed in which general guards are essential, for example

```
VAR x:
ALT
  c!32
    SKIP
  d?x
```

```
PROC buffer(CHAN in, out) =
  CHAN fwd, ack:
  PAR
    VAR head, tail, count, queue[n] :
    SEQ
      head := 0     -- index of head of queue
      tail := 0     -- index of tail of queue
      count := 0    -- number of items in queue
      WHILE TRUE
        ALT
          count < n & in ? queue[tail]
            PAR
              tail := (tail + 1) \ n
              count := count + 1
          count > 0 & ack ? ANY
            SEQ
              fwd ! queue[head]
              PAR
                head := (head + 1) \ n
                count := count - 1
    WHILE TRUE
      VAR x:
      SEQ
        ack ! ANY
        fwd ? x
        out ! x
```

Figure 5: `n+1`-place buffer without output guards

```
        SKIP
```

which offers its environment both an input and an output, neither of which is inevitable. There is no way of eliminating either the conditional input or the conditional output from this program.

Again, even if all the communications of a program with its environment are inevitable, it is possible to construct programs in which the general guards can only be eliminated by gross changes to the structure of the whole program. (All the examples that I have been able to construct are pretty bizarre, like this one.)

```
CHAN c[4]:
PAR i = [0 FOR 4]
  VAR x:
  SEQ
    ALT
      c[i]!ANY
        x := i
      c[(i+1) MOD 4]?ANY
        x := (i+1) MOD 4
    p(i, x)        -- x's are all even or all odd together
```

It follows that no local methods such as are presented here can eliminate output guards from all programs. On the other hand I have yet to be stumped by an actual programming problem. Once familiar with these ideas, their application becomes, like all programming tricks, second nature. I did, however, consciously use (on the back of an envelope) a manipulation essentially like the one quoted here to code `PROC node` for the first time to the specification of [7]. I did not think I would otherwise get it right first time.

References

[1] R. Bornat, 'A protocol for generalised *occam*', in Software Practice and Experience, Vol 16 No 9, September 1986.

[2] Buckley and Silberschatz, 'An Effective Implementation for the Generalised Input-Output Construct of CSP', in ACM TOPLAS 5.2, April 1983, pp 223–235.

[3] C.A.R. Hoare, 'Communicating Sequential Processes', in Commun ACM, Vol 21 No 8, August 1978, pp 666–677.

[4] G. Johnson, 'Bidirectional I/O', (Colorado State University) presented at OUG TM-4, Manchester, March 1986.

[5] G. Jones, A.W. Roscoe, *et al*, 'Distributed Communications', Programming Research Group, Oxford (to appear).

[6] K. Mitchell, 'Implementations of process synchronisation and their analysis', D.Phil thesis, University of Edinburgh Department of Computer Science, 1986.

[7] A.W. Roscoe, 'Routing messages through networks: an exercise in deadlock avoidance', Proc. OUG TM-7, Grenoble, 1987.