Appears in *occam and the Transputer, research and applications*, IOS, September 1988.

The meaning and implementation of PRI ALT in occam

Geoff Barrett
Michael Goldsmith
Geraint Jones
Andrew Kay

Programming Research Group
Oxford University Computing Laboratory
11 Keble Road, Oxford OX1 3QD, U.K.

28 July 1988

Abstract

This paper describes what we believe to be an error in the implementation of `PRI ALT` on the inmos transputer, and exhibits anomalous behaviour of that implementation. A correct implementation of comparable complexity is described, and some of the properties of the construct are described. Finally, an attempt is made to describe circumstances in which the behaviour of the existing implementation is adequate for the correctness of a program which uses it.

Concepts

An *alternation* is a process which is simultaneously prepared to communicate with a number of other processes, and which selects exactly one of those that are prepared to communicate with it. In the occam languages[1, 2], the simplest form of alternation consists of a number of branches each beginning with an input process. The alternation executes exactly one of these inputs and the process which follows it, making its selection from amongst those inputs using channels on which another process is already prepared to output.

The initial input of each branch of an alternation is called its *guard*, and a guard is said to be *ready* when another process has committed itself to performing an output on that channel. Two observations are worth making about readiness. Firstly, since output processes are unconditional, a guard which becomes ready remains ready until discharged by the corresponding input. Secondly, it is possible for several branches of an alternation to be guarded by inputs from the same channel; in this case they would all become ready simultaneously. For the most part we confine ourselves in this paper to considering only guards that consist of a simple input process; the treatment can be extended to accommodate `SKIP` guards, delay guards, and guards accompanied by Boolean conditions.

In occam there are two alternation constructs: the *symmetric* construct introduced by the keyword `ALT` gives the weakest useful guarantee about the selection of a guard; the *asymmetric* construct introduced by the keywords `PRI ALT` gives a stronger guarantee which favours some guards over others. In each case there is a *liveness* guarantee that once some guards become ready, a selection will eventually be made and a branch will be executed. The *safety* guarantee of the symmetric construct is only that the selected branch is one of those that is ready. The asymmetric construct gives the stronger guarantee that (there is some time during the negotiation when) the selected branch has a ready guard, and that (at that time) no higher priority branch has a ready guard.

Notation

We are only concerned with the negotiation between potential partners in a communication; once an alternation has selected a guard, the data will be moved exactly as if between an unconditional input and output. The data transfer is irrelevant to our purpose, so we will write just c! for any output on c and c? for any input on c.

Similarly, we want to make no distinction between replicated (intensional) constructs and extensionally presented ones. Anything we say about

```
ALT i
  c_i?
    P_i
```

is going to be true about any alternation, whether written with or without replication, for example

```
ALT
  c?
    P
  d?
    Q
```

Again, since we are going to be particularly concerned with the relative priorities of the branches of alternatives, we will write, for example

```
ALT
  c?
    P
  PRI ALT i
    d_i?
      Q_i
```

when we want to ensure an ordering on the priorities of the guards on d_i, but are not concerned about the priority of the guard on c relative to those on the ds. The concrete syntax of occam does not allow this.

The transputer implementation of `ALT`

The transputer implementation of occam allocates a variable to represent each channel, this variable being shared by the process which sends output over the channel, and the process which takes input from the channel. When a process commits itself to an output, it stores an encoding of its own identity in the channel variable. An alternation polls the channel variables, waiting until at least one of them contains the identity of a committed output process, indicating a ready guard.

In fact, as described in the compiler writers' guide[3], a single pass is made over the branches, executing an 'enbc' instruction for each channel, to enable the channel. An enabled channel contains a pointer to the (unique) alternation which is expecting to perform an input on it. A single 'altw' wait instruction then suspends execution of the alternation until one of the guards is ready. When an output is performed on an enabled channel, it causes the corresponding guard or guards to become ready, and the outputting process reschedules the alternation. In this way the wait is not busy.

Once at least one guard is ready, the alternation makes a single scan of all of the branches in some order, selecting one of the ready guards. This pass over the branches is implemented by a sequence of 'disc' instructions, one for each guard. The disabling instruction removes the pointer from the guard's channel if that channel is still enabled. Additionally, the first disabling instruction to encounter a ready guard records the identity of that guard.

After the end of this scan, a branch has been selected which has a ready guard, so this implementation correctly achieves the precondition (A) for execution of the selected branch of an `ALT`, that its guard should be ready.

The selected branch is one of those with ready guards. ...A

Properties of `ALT`

The essence of alternation is that it acts as an arbiter between a number of concurrently presented alternative communications. If a number of parallel processes are prepared (become committed) to output on channels c_i then an alternation guarded by inputs from these c_i will select exactly one communication. We require that

$$\begin{array}{l} \texttt{PAR} \\ \quad \texttt{PAR}\ i \\ \quad\quad c_i\,\texttt{!} \\ \quad \texttt{ALT}\ i \\ \quad\quad c_i\,\texttt{?} \\ \quad\quad\quad P_i \end{array} \quad \text{can behave like} \quad \begin{array}{l} \texttt{PAR} \\ \quad \texttt{PAR}\ j \neq i \\ \quad\quad c_j\,\texttt{!} \\ \quad P_i \end{array} \quad \text{for any } i$$

which is to say that it cannot be distinguished by its observable behaviour from the process

```
ALT i
  SKIP
    PAR
      PAR j ≠ i
        c_j !
      P_i
```

that makes an uncontrollable choice of i.

This equivalence is certainly true of the transputer implementation of occam, in the following precise sense. An execution of the code compiled for either process will be a possible behaviour of the (same) abstract meaning given to both processes by the standard semantics[4]. Notice that this is not to say that the observable behaviour of a particular execution of the code compiled to implement one program would (or indeed even could) ever be observed as the behaviour of any execution of the code implementing the other.

It is a consequence of the model of synchronization underlying the standard semantics that there is nothing essential about concurrency, as compared with interleaving. To say that two communications are simultaneous is for us only to say that they might be observed in either order. A consequence of this is the following simulation theorem, that

```
PAR i
  SEQ
    a_i ?
    P_i
```

is the same as

```
ALT i
  a_i ?
    PAR
      P_i
      PAR j ≠ i
        SEQ
          a_j ?
          P_j
```

by successive applications of which PAR may be removed from any (loop-free) program. (This law is a fairly immediate consequence of the second expansion law in reference [5] under the assumption that each channel is properly used, by one process for input and one other for output.)

The simulation theorem is also true of ALT as implemented on the transputer.

The (incorrect) transputer implementation of PRI ALT

The transputer implementation of occam executes exactly the same code for PRI ALT as for ALT. It guarantees additionally that the selection (second) pass is made in the order of the branches, which is the order of decreasing priority. It selects the first branch which has a ready guard at the time that the corresponding disable instruction is executed.

In the case of a binary PRI ALT, this means that the second branch could only be selected in case there was a time when the second guard was ready, and the first guard was not. So, for example,

```
PAR
  PRI PAR
    a!
    b!
  PRI ALT                                   ...1
    a?
      Pa
    b?
      Pb
```

is the same as

```
PAR
  b!
  Pa
```

because the the `PRI PAR` causes the a`!` to become committed, and so the a`?` to be ready, before the b`!` is committed and the b`?` ready. (`PRI PAR` differs from `PAR` in that no component process executes while a higher priority process can make progress.)

Pathological behaviour of this implementation

Unfortunately, this implementation allows of some peculiar behaviours in asymmetric alternations with more than two branches. In particular

```
PAR
  PRI PAR
    a!
    b!
  x!                                  PAR
  PRI ALT                               a!
    a?         might behave like        x!
      Pa                                Pb
    b?
      Pb
    x?
      Px
```

even though there is no time at which the b`?` guard is ready and the a`?` guard is not. A sequence of execution which leads to this pathology is:

alternation enables each of a, b, x in some order, and waits
x`!` is committed to output, and reschedules the alternation
a`?` is disabled but this guard is not selected, not being ready
a`!` is committed to output
b`!` is committed to output
b`?` is disabled and this guard being ready is selected
x`?` is disabled but is not selected, not being the first ready guard seen

At the end of this sequence, the guard on b has been selected and the communication between it and the b`!` will happen.

An extreme case of this behaviour is that as implemented by the transputer

```
PAR
  a!
  PRI ALT
    PRI ALT i
      a?
        P_i
    SKIP
      Q
```
...2

may execute any one of the P_i, despite their relative priorities. This is because the `SKIP` guard means that the alternation can start the final selection pass over the guards before the a`!` is committed.

Our expectation of `PRI ALT` is that it should guarantee, at the very least, that only the highest priority guard(s) of a set that become ready simultaneously can possibly be selected. Specifically, we would expect that process (2) would either select Q or the very first of the P_i and only that one.

A correct implementation of `PRI ALT`

Our new algorithm is quite similar to the existing one: a slightly modified disabling action is needed; and the order in which the guards are disabled is reversed. The selected branch is the last one seen to be ready, when they are scanned in order of increasing priority. The intuition is that the highest priority branch must be given the longest opportunity to overcome the claims of those with lower priority. So

```
PRI ALT
  c1?
    P1
  ...
  cn?
    Pn
```

would be translated as

enable $c_1, \ldots, c_n$ (in any order)
wait for one or more guards to become ready
disable c_n, if c_n is ready select branch n
disable c_{n-1}, if c_{n-1} is ready select branch $n-1$, even if branch n was selected earlier
...
disable c_1, if c_1 is ready select branch 1, in preference to any branch already selected
execute the selected input and the corresponding process

Notice that since one of the guards is ready before the sequence of disabling instructions, and that a guard stays ready until this process performs the corresponding input, at least one of the guards will be seen to be ready. That guarantees that a branch is selected by at least one of the disabling actions.

Since a branch is only selected if its guard is ready, the selected branch will certainly have a ready guard, which achieves the safety guarantee (A). Moreover, if the selected branch is the i^{th}, and if $j < i$, then when the j^{th} guard is disabled it is not ready (for if it were, it would be selected in preference to the i^{th}). Since a guard stays ready when once it has become ready, this means that the j^{th} guard was not ready when the i^{th} guard was disabled. So, in order for the i^{th} branch to be selected, there must have been a time when the i^{th} guard was ready and no higher priority guard (one with a lower index) was ready.

Conclude from this argument the stronger safety guarantee which we require of `PRI ALT`, that for a branch to be selected it must have a higher priority than any branch with a guard which is ready when the guard of the selected branch becomes ready. More precisely, it must have a higher priority than any branch with a guard which is ready either before or simultaneously with it.

> The selected branch has a guard which was ready at a time when no higher priority branch had a ready guard. ...B

This is clearly not true of the transputer implementation of `PRI ALT`, as witness the pathology described earlier.

Properties of `PRI ALT`

The semantics in reference [4] does not describe the `PRI ALT` and `PRI PAR` constructors, except to say that an asymmetric process should be an implementation of (should be at least as deterministic as) the symmetric process obtained by cancelling each occurrence of `PRI`. A semantics which distinguishes between symmetric and asymmetric constructions is presented in reference [6], together with an operational semantics describing the (modified) transputer implementation of the relevant parts of **occam**. The equivalence of these two semantic descriptions is a justification that the language is correctly implemented.

In the new semantics, it is possible to distinguish symmetric and asymmetric alternations, in the sense that in general

`PRI ALT` i
  c_i`?`
    P_i
	is different from	`ALT` i
		  c_i`?`
		    P_i

In fact, the first is strictly more deterministic than the second. However, a number of properties of the two constructs are the same; for example it is still the case that

`PAR`
  `PAR` i
    c_i`!`
  `PRI ALT` i
    c_i`?`
      P_i
	is equivalent to	`ALT` i
		  `SKIP`
		    `PAR`
		      `PAR` $j \neq i$
		        c_j`!`
		      P_i

In general, as remarked in reference [6], it is only possible to distinguish between `ALT` and `PRI ALT` by using `PRI PAR` or a fairness property. (The `PRI PAR` may be explicit in the program, or it may be implemented by the user who interacts with the program.) For example,

```
PAR
  PRI PAR 1 ≤ i ≤ n
    c_i !
  PRI ALT 1 ≤ i ≤ n
    c_i ?
      P_i
```

is equivalent to

```
PAR
  PRI PAR 2 ≤ i ≤ n
    c_i !
  P_1
```

where c_1? is, of course, the guard with the highest priority.

In this framework, a simulation theorem can be proved for `PRI PAR`, specifically that

```
PRI PAR i
  SEQ
    a_i ?
    P_i
```

is the same as

```
PRI ALT i
  a_i ?
    PRI PAR
      PRI PAR j < i
        SEQ
          a_j ?
          P_j
      P_i
      PRI PAR j > i
        SEQ
          a_j ?
          P_j
```

Because the modified implementation guarantees that when it selects a branch it selects the highest priority branch with that guard, we can also prove that

```
ALT
  ...
  PRI ALT i
    e_i & x?
      P_i
  ...
```

is the same as

```
ALT
  ...
  ∃ i. e_i & x?
    IF i
      e_i
        P_i
  ...
```

which is the sort of result that would be used in demonstrating the correctness of an optimizing compiler. An immediate consequence of this last observation is that process (2)

```
PAR
  a!
  PRI ALT
    PRI ALT 1 ≤ i ≤ n
      a?
        P_i
    SKIP
      Q
```

is equivalent to

```
ALT
  SKIP
    P_1
  SKIP
    Q
```

and can never execute any other of the P_i than that with the highest priority.

It is the fact that few of these observations are true of the transputer implementation of `PRI ALT` that is the substance of our claim that that implementation is incorrect. Each of these equivalences would be true of our modified implementation.

Limits on the distributability of `PRI ALT`

A few words are in order about what we mean by the order in which the environment of a process makes channels become ready. In a physically distributed system, it is not always clear what constitutes an event, nor when two physically separated events can be said to be simultaneous. Consider process (1) again: recall that it is certainly true that

```
PAR
  PRI PAR
    a!
    b!                          PAR
  PRI ALT   is the same as        b!
    a?                            Pa
      Pa
    b?
      Pb
```

because the `PRI PAR` causes the a`!` to be committed before the b`!`, and this is the same as saying that the a`?` guard is ready before the b`?` guard.

Suppose however that the two processes are distributed onto two processors, separated by a significant distance,

```
PLACED PAR
  PRI PAR
    a!
    b!
  PRI ALT                                                        ...3
    a?
      Pa
    b?
      Pb
```

and connected by independent pieces of wire implementing the channels a and b. On a network of transputers, such channels are implemented by a pair of inmos link controllers, one in each processor. The 'shared variable' which we described as implementing a channel is effectively present in the state of the link controller at the inputting end of a channel.

The negotiation leading up to a synchronization remains unchanged at the inputting end of the channel, indeed the code is identical. The difference is that when a process commits itself to an output to a channel implemented by an inter-process link, it does so by sending a message over the link, and suspending its own execution until both the synchronization and the subsequent data transfer have been completed. Only when the commitment message (actually the first byte of the data) arrives at the inputting processor does any guard on that channel become ready. There is no further communication on that link until the synchronization is complete.

It is quite possible that the message from one processor to the other to make the a`?` guard ready could be overtaken by the signal on another link to make b`?` ready. In this case, it is still possible for process (3) to behave like

```
PAR
  a!
  Pb
```

regardless of the selection strategy of the `PRI ALT`, and we would have to treat it as being the same as

```
ALT                                   PLACED PAR
  SKIP                                  PAR
    PAR                                   a!
      a!                                  b!
      Pb   which is the same as         PRI ALT
  SKIP                                    a?
    PAR                                     Pa
      b!                                  b?
      Pa                                    Pb
```

In effect, the asymmetry of the `PRI PAR` is not visible to the process running on the other transputer.

Ensuring that one commitment signal does not overtake another might prove to be costly: we believe that in general it would require that commitment signals be acknowledged, and that no lower priority process could be allowed to make progress if there were an outstanding acknowledgement of commitment expected at a given priority. Effectively, the a`!` in process (3) would have to wait busily for an acknowledgment that the a`?` guard was ready, to prevent the b`!` from becoming committed too early.

It would seem to be unjustifiably expensive to do this, particularly since we see no particular practical need to have distributed negotiations between `PRI ALT` and `PRI PAR`.

Getting by with the `PRI ALT` as implemented by the transputer

We would expect that anyone constructing an implementation of occam in this style would follow the model of our corrected implementation, but to do so on a transputer would not be easy since we require to replace the *disable channel* instruction:

disc disable the channel;
  if it is ready and if no branch is selected, select this channel

by a new instruction

disc′ disable the channel;
  if it is ready then provisionally select this channel in preference to any already selected

and similarly for each of the other disabling instructions: diss, dist.

Fortunately, it appears that the transputer implementation of `PRI ALT` is adequately asymmetric to be useful in most places where one might need an asymmetric alternation in practice. First of all we note that the selection strategy of the transputer meets the following safety condition: that if the guard of the i^{th} branch is ready before the negotiation begins and the j^{th} branch is eventually selected, then $j \leq i$. For certainly, if the guard of the i^{th} branch is ready before the negotiation, it is still ready when the i^{th} guard is disabled. Unless, for some $j < i$, the j^{th} branch has been selected, the i^{th} branch will be selected.

> The selected branch has a ready guard, and no higher priority branch had a ready guard before execution of the `ALT` began. ...C

In particular, if the guard of the first branch is ready before the negotiation begins, then the first branch will certainly be selected.

Notice, by the way, that because the safety guarantee (B) afforded by our modified strategy is stronger than that (C) of the transputer strategy, the modified strategy gives a more deterministic process. The most important consequence of this is that any reliable property of the transputer implementation of a program is also a property of the modified implementation of the same program. You have nothing to loose by correcting your implementation of occam!

In practice, most uses of `PRI ALT` have a form something like

```
WHILE e
  PRI ALT i
    ci?
      Pi
```

where if c_i and c_j are the same channel, then P_i and P_j are identical processes; and each P_i uses none of the c_j, except perhaps its corresponding c_i. If a particular c_i? is perversely passed over in favour of a c_j?, $j > i$, by the transputer implementation, then it will not be discharged by execution of P_j, and will still be ready when the `PRI ALT` is next executed (assuming that e remains true). This means that the i^{th} branch cannot be starved by any lower priority branches, and in particular

```
WHILE TRUE
  SEQ i
    PRI ALT
      ci?
        Pi
      ALT j
        cj?
          Pj
```

still gives a guarantee of short-term fairness to each of the branches in turn because this guarantee depends only on absolute priority eventually being given to the first branch.

We can also impose conditions on the context of an alternation which ensure that the difference between the two implementations cannot be detected. If in an alternation no pair of the guards using the same channel is followed by a guard using another channel, and if there is no pair of guards, c_i? at a higher priority than c_j?, for which there is an output process c_j! at a higher priority than some other output process c_i!, then the behaviour of the whole program is unaffected by the difference between the two implementations. Effectively, this is to say that if the outputs become committed in the order of decreasing priority of the guards, it is not possible for the transputer implementation to make an anomalous choice. Whenever a `PRI ALT` appears in a context like this where the two implementations behave identically, anything true of the one will be true of the other. In such a case it is safe to use the simpler axiom (B) instead of (C) to reason about the transputer implementation.

Again, if it is possible for the guards of an alternation to become ready in any order, then no property of the alternation which depends on the difference between the two implementations can matter. In this case, neither implementation performs significantly differently from the symmetric `ALT`.

Summary

We have described a number of equivalences which we consider to be essential properties of the `ALT` and of `PRI ALT` constructs of occam. These equivalences can be shown to be true of the abstract meanings ascribed to the processes by a defining semantics.

We described the implementation on a transputer of the `ALT` and `PRI ALT` constructs, and showed that some of the expected properties of `PRI ALT` were not true of this implementation. The slightly modified, indeed slightly simplified, implementation of `PRI ALT` which we suggested corrects the error. There seems, however, to be no simple way of preventing distributed, `PLACED`, programs from behaving differently from single-processor programs.

Our final remarks showed that it is possible in practice to make use of the transputer implementation of `PRI ALT`, at the expense of needing to understand a much subtler specification. The most significant failing of the transputer implementation of `PRI ALT` is that it does not have a simple semantics, indeed that the easiest way of explaining what it means is to describe what it does.

References

[1] inmos Limited, occam *Programming Manual*, Prentice-Hall International, 1984.

[2] inmos Limited, occam 2 *Reference Manual*, Prentice-Hall International, 1988.

[3] *The transputer instruction set – a compiler writers' guide*, inmos document No 72 TRN 119 02, March 1987.

[4] A.W.Roscoe, *Denotational Semantics for* occam, in *Seminar on Concurrency*, ed. S.D. Brookes, A.W. Roscoe and G. Winskel, Springer-Verlag Lecture Notes in Computer Science 197, July 1984.

[5] A.W. Roscoe and C.A.R. Hoare, *The Laws of* occam *Programming*, Programming Research Group technical monograph PRG-53, (*to appear in* Theoretical Computer Science).

[6] Geoff Barrett, *Semantics and implementation of* occam, D.Phil. thesis, Oxford, September 1988.