Pictorial Representation of Parallel Programs

Susan Stepney

GEC-Marconi Research Centre, West Hanningfield Road,
Great Baddow, Chelmsford, Essex, CM2 8HN.

ABSTRACT

The structure of a parallel program is considerably more difficult to visualize and understand than that of a sequential one. Pictorial methods can help make the structure more visible.

In this paper I describe a pictorial representation of occam programs. This representation has been developed as part of the Alvey ParSiFal project, for use by one of its tools, GRAIL.

1 Introduction

Parallel programs have considerably more complicated structures than do sequential programs. A parallel program has many threads to the computation, and each thread can have all the complexity of structure of a sequential program. In addition, there can be a complicated communication structure between these threads. Pictorial methods can help make the parallel structure more visible.

In this paper I describe a pictorial representation of occam programs. This representation has been developed as part of the Alvey ParSiFal project, for use by one of its tools, GRAIL [1].

The *structure* of the program is shown pictorially, by use of nested boxes, and by arrows. This pictorial representation continues down to the level of an individual statement. Statements themselves are shown textually - little is gained, and much is lost, trying to contrive a pictorial representation for these.

2 occam Overview

Before I describe the representation, I give a brief overview of occam for those unfamiliar with it. A full description of the language is given in [2].

The parallel programming language occam is derived from the formal language CSP [3]. occam has parallelism designed in from the start. It is based on the ideas of *concurrency* and *communication*. An occam program consists of a set of concurrent processes that communicate via channels at well defined points in their execution. There is built in synchronization - whichever process is ready to communicate first waits for the other to be ready.

The building blocks of occam processes are three primitive processes; assignment, input and output. They are combined by conventional loop (`WHILE` and replicated `SEQ`) and `IF` constructors, and can be abstracted away as procedures.

If processes are to be run sequentially, this has to be stated explicitly by use of the `SEQ` keyword. Similarly, the `PAR` keyword says the processes are to be run in parallel.

There is one more construct, `ALT`. This means "alternative choice", and allows a process to wait for input from a number of channels, and proceed when any one is ready to communicate. If more than one is ready, the choice of which one proceeds is made non-deterministically.

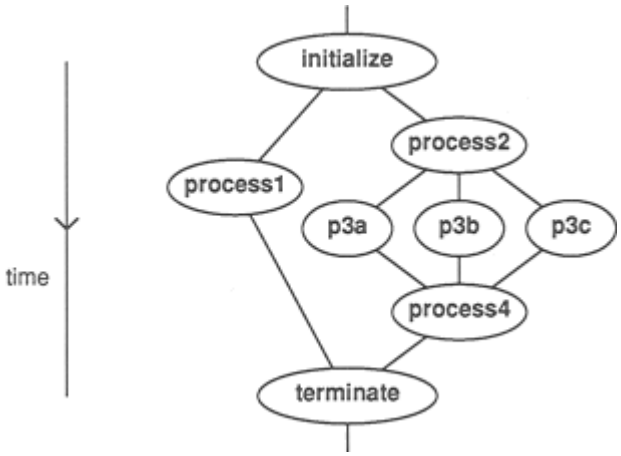
Susan Stepney. "Pictorial Representation of Parallel Programs." A. Kilgour and R. Earnshaw, editors, *Graphical Tools for Software Engineering*, BCS conference proceedings, pages 46-59. Cambridge University Press, 1989.

A collection of occam processes combined with one of the above constructors is also a process. So processes can have internal parallelism and communication, but can also be viewed as black boxes, just in terms of their external communications. This lends itself to a hierarchical process design.

3 Two-Dimensional Display

The pictorial representation used in GRAIL is two dimensional. The vertical dimension (down the page) is used in the conventional way to represent sequential execution, and deterministic choice (`IF`). The horizontal dimension (across the page) is used to indicate parallelism and non-deterministic choice (`ALT`). The various parallel threads of execution are drawn side by side.

Consider the following hypothetical process structure:



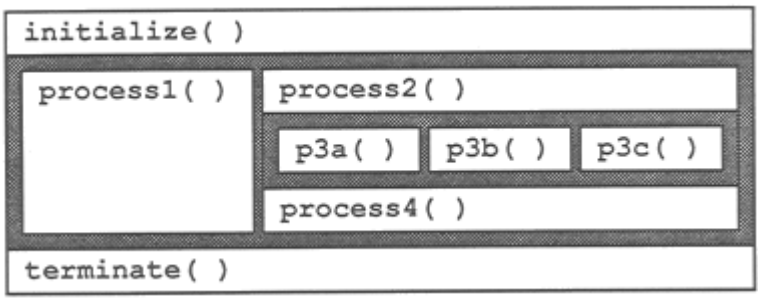
This represents a process that starts off, splits into a few parallel threads, then finishes up.

The occam for such a structure would look like:

```
SEQ
  initialize()
  PAR
    processl()
    SEQ
      process2()
      PAR
        p3a()
        p3b()
        p3c()
      process4()
  terminate()
```

Notice how the clear distinction between parallel and sequential processes has been lost. Also, the scope of the processes is not obvious, for example, is `terminate` in parallel with `process1`, or sequentially after it?

The pictorial representation of the same structure is

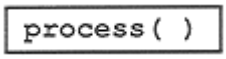


Now the parallel structure, and scope, is visible at a glance.

4 Processes

In this and the following two sections I discuss the GRAIL representation in some detail. I then give a fairly realistic example, illustrating the use of many of the features described.

The GRAIL display uses the process as its fundamental unit. An occam process is drawn in a rectangular box:



Each process can be hierarchically composed of other processes. Each occam construct (`SEQ`, `WHILE`, `IF`, `PAR`, `ALT`) has a different pictorial representation, and is drawn background of a different shade of grey. (I do not discuss `CASE` statements, functions, or variant protocols). The darker the background grey, the more parallel, or less deterministic the construct.

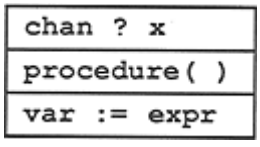
In theory, the picture drawn this way is completely determined by the occam code. In practice, however, the text inside a box can be split over more than one line, in order to maximise the amount displayed.

4.1 Sequential Processes

A sequential process is drawn with the component process boxes stacked vertically. The component boxes all have the same width, but may have different heights. So

```
SEQ
  chan ? x
  procedure( )
  var := expr
```

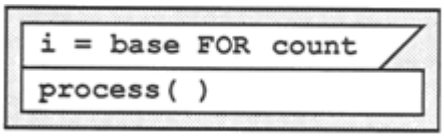
becomes



A replicated `SEQ` is drawn with a replicator bar over the body process, all enclosed in a box with a light grey background.

```
SEQ i = base FOR count
  process( )
```

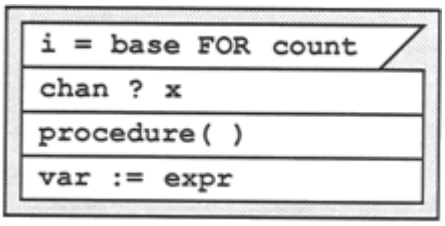
becomes



and

```
SEQ i = base FOR count
  SEQ
    chan ? x
    procedure( )
    var := expr
```

becomes



This type of bar, with a corner chopped off, is used to indicate all replicator and loop constructs.

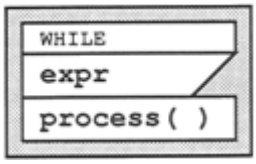
4.2 While Loops

The replicator bar shape is also used in the display of a `WHILE` loop, showing the link :tween replicated `SEQ`s (essentially loops) and `WHILE` loops.

The keyword `WHILE`, and the expression written inside the bar, are drawn above the body process. This is drawn on a light to mid grey background. (The `WHILE` grey is darker an the replicated `SEQ` grey, since the construct is less deterministic - the number of times round the loop is not necessarily known in advance). So

```
WHILE expr
  process( )
```

becomes



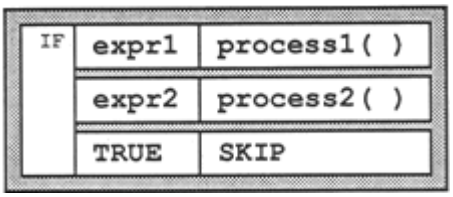
4.3 If Processes

An `IF` process is also drawn vertically, indicating that the first process down the list with a `TRUE` choice is executed.

Each expression is drawn in a box, with the corresponding process box to its right. The expression/process pairs are drawn vertically, with gaps to separate them. They are joined by a box on the left, containing the keyword `IF`. The whole thing is drawn on a mid grey background. So

```
IF
  exprl
    processl( )
  expr2
    process2( )
  TRUE
    SKIP
```

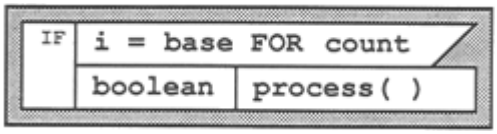
becomes



A replicated `IF` has a replicator bar, drawn over expression/process pair or internal `IF` process.

```
IF i = base FOR count
  boolean
    process( )
```

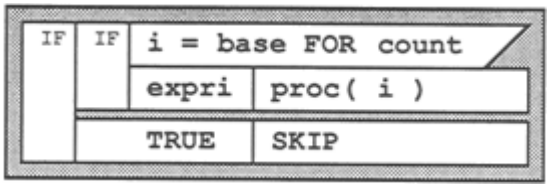
becomes



The fairly common occam cliche

```
IF
  IF i = base FOR count
    expri
      proc ( i )
  TRUE
    SKIP
```

becomes



Note that all the process boxes are the same width, and the expression box widths are altered to take into account the keyword boxes.

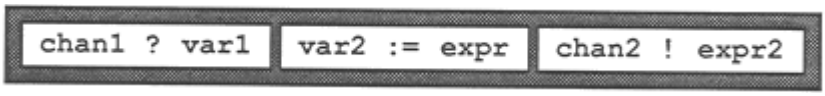
The display highlights the fact that the construct is essentially a single `IF` statement, not two nested ones.

4.4 Parallel Processes

A parallel process is drawn horizontally. The component processes are drawn with gaps between them, on a dark grey background.

```
PAR
  chanl ? varl
  var2 := expr
  chan2 ! expr2
```

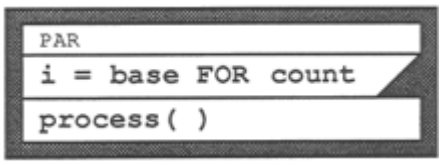
becomes



A replicated PAR has a keyword and replicator bar over it.

```
PAR i = base FOR count
   process( )
```

becomes



A `PRI PAR` is drawn similarly, except the box is a lighter grey, since the construct is more deterministic (if both branches of the `PRI PAR` *can* execute, you know which one is executing).

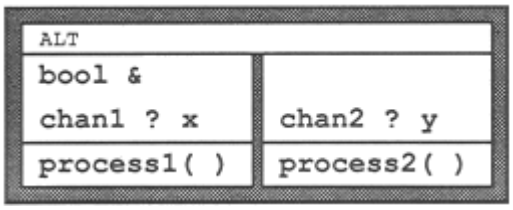
4.5 Alternative Processes

`ALT`s are drawn horizontally, to indicate the non-deterministic choice if more than one guard is ready.

Each guard is drawn in a box, and is split over two lines if it includes a boolean expression. The corresponding process box is drawn beneath the guard. The guard/process pairs are drawn horizontally, with gaps to separate them. They are joined by a box above them, containing the keywords `ALT` or `PRI ALT`. The whole thing is drawn on a mid to dark grey background. So

```
ALT
  bool & chanl ? x
    processl( )
  chan2 ? y
    process2 ( )
```

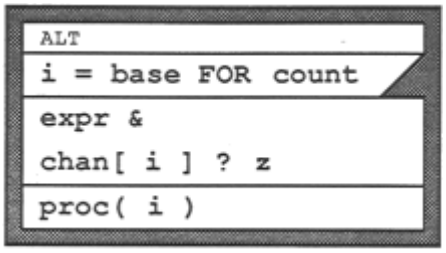
becomes



A replicated ALT has a replicator bar drawn above the guard/process pair. So

```
ALT i = base FOR count
  expr & chan[ i ] ? z
    proc( i )
```

becomes



For more complicated nested ALTs, all the process boxes are the same height, and the guard box heights are altered to take into account the replicator bars.

5 Folds

One of the interesting features on the Inmos TDS editor is its folding ability. A process, or group of processes, can be grouped together and "folded up". The group can be displayed in the normal way, or shown just as a comment summarizing their function.

For example,

```
SEQ
  process ()
  --{{{ fold
  proc.1()
  proc.2()
  --}}}
  another.process()
```

when folded, looks like

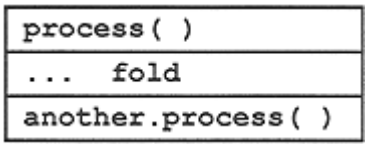
```
SEQ
  process =()
  ... fold
  another.process()
```

The folding allows the amount of information on display to be controlled. It also encourages meaningful comments, since when the fold is closed the comment is the *only* indication of what is inside.

This folding is so useful, and so common among occam programmers, that GRAIL's pictorial representation supports it.

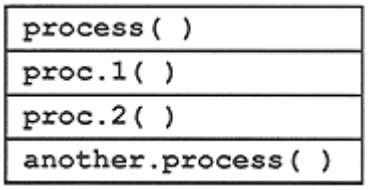
5.1 Closed Folds

When closed, a fold appears as a box containing the fold comment.

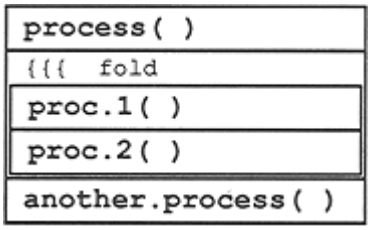


5.2 Open Folds

There are two options for open folds. They can either be indistinguishable from an unfolded process



or have the fold comment displayed in a small font

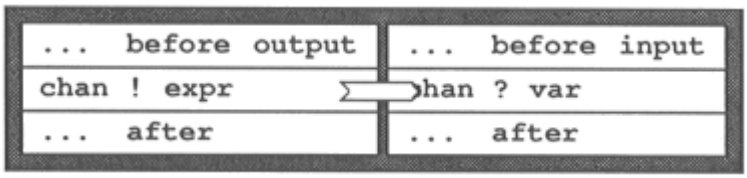


6 Channels

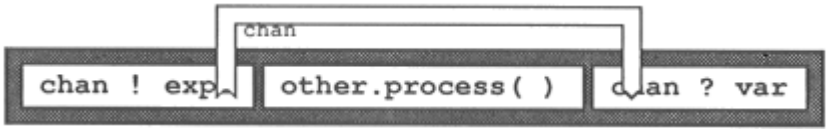
Channels are drawn as arrows between the processes they connect. Drawing channels is not so clear cut as the deterministic drawing of the process structure. A route between the processes needs to be decided. The criterion used is to minimize the amount of picture obscured by the drawn channel.

Since the vertical dimension represents time, and since channel communication synchronizes the processes at either end, channels tend to be horizontal arrows.

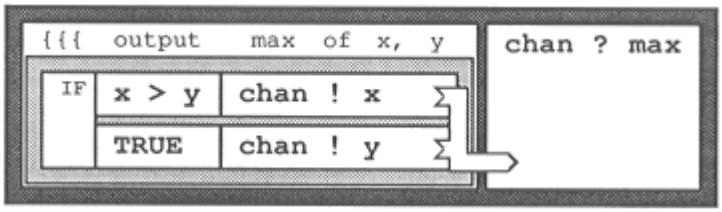
A channel is drawn as an arrow to show the direction of the communication:



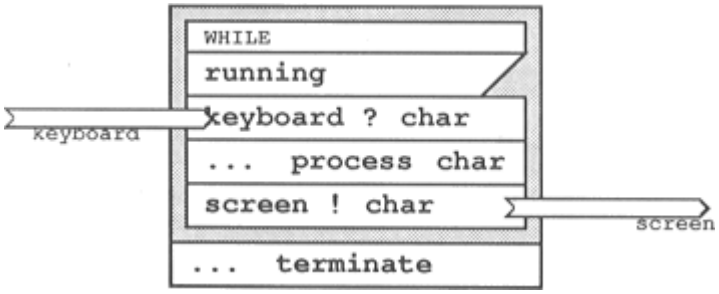
If the input and output processes are far apart in the picture, the channel can bend to avoid other processes:



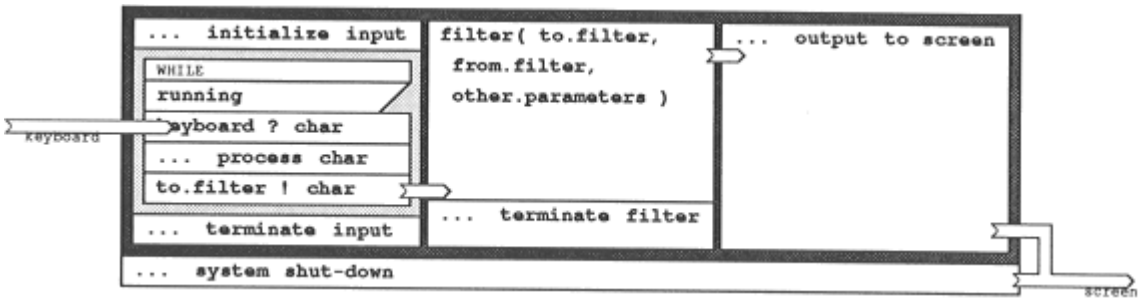
Channels can originate, or end, in more than one place:



or may have only one end attached:



More than one channel may enter or leave a box if it is a procedure call or closed fold:



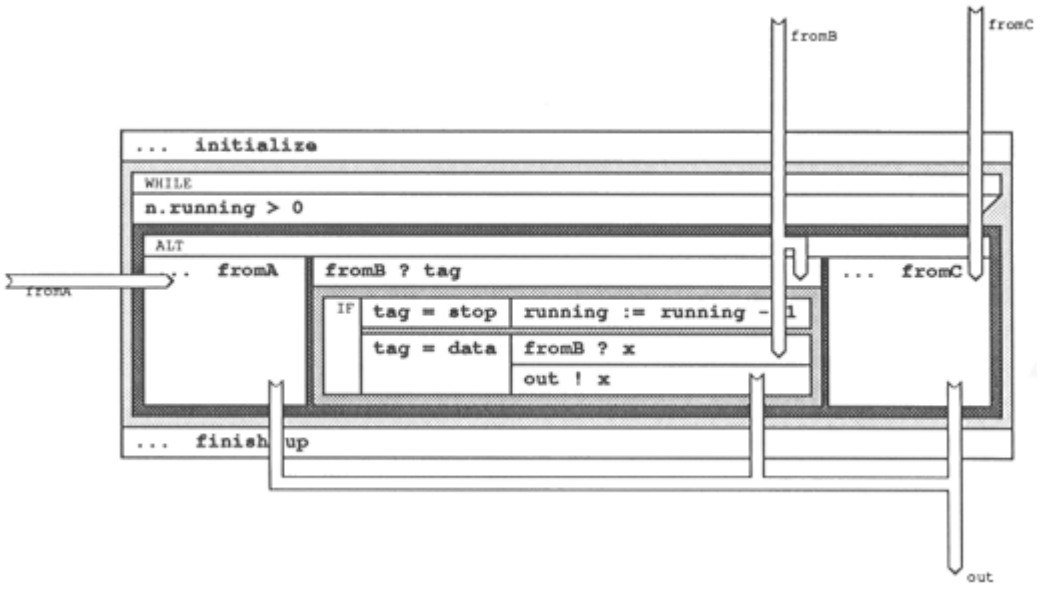
7 Channel Multiplexor

Consider a process that takes input from three other processes, and outputs them on a single channel. The occam for this looks something like

```
SEQ
  ... initialize
  WHILE running > 0
    ALT
      ... fromA
      --{{{ fromB
        fromB ? tag
          IF
            tag = stop
              running := running - 1
            tag = data
              SEQ
                fromB ? x
                out ! x
      --}}}
      ... fromC
  ... finish up
```

It is not possible to see from this text that there are channels entering and leaving the folds, or that the same channel leaves all three folds. In other words, it is not possible to see "at a glance" that this is a multiplexor.

The GRAIL display, with channels, is



Now the communications structure is visible. The fact that the program merges input from three channels onto one is much more obvious.

8 GRAIL and Colour

The ParSiFal occam monitoring tool, GRAIL, uses the representation described above to show the monitored program. The display (currently running on a Sun workstation) is interactive; the user can select which procedure to look at, can manipulate folds, and can display or hide channels, using the mouse.

The representation was developed as a way of displaying the monitoring data. The activity in various parts of the program is overlayed on the relevant process box, or channel,

in colour. Blue indicates inactive processes, with a gradual change in colour to red for the most active processes. The colour allows interesting areas of the program to be found.

As with the pictorial representation of the program, colour is used only where appropriate, for high level information. Once an interesting area of the program has been found, the actual monitoring data can be displayed in a textual form (as numbers).

9 Conclusions

The textual occam representation in particular the folding, directly supports hierarchical process structure. However, the distinction between parallel and sequential processes is obscured by the linear form of the text. Orthogonal to the process structure is the communications. The textual representation does not support any structuring of this.

The representation used by the GRAIL, display also supports the hierarchical process structure. In addition, it highlights the parallelism and scope of processes. The pictorial form, by directly showing the communication paths, encourages simplicity (*i.e.* tidiness) and discourages complexity (*i.e.* spaghetti). It becomes much easier to trace communication routes through the program.

Not only does the GRAIL display help users while developing their own programs, it also comes in very useful when puzzling out a program written by someone else. Also, monitoring (or other) information can be overlayed in colour, allowing interesting parts of the program to be discovered "at a glance". It is important, however, that a textual form is used at the lowest level for detailed information.

GRAIL has been developed for displaying monitoring data. Other uses of the representation, for example, to display debugging data (values of variables) or as a graphical occam editor, would require an extension to the display to include channel and variable declarations.

References

[1] S. Stepney, "GRAIL - Graphical Representation of Activity, Interconnection and Loading", *Proceedings of the occam User Group 7th Technical Meeting*, Grenoble, 1987.

[2] Inmos Ltd, *occam 2 Reference Manual*, Prentice-Hall International, 1988.

[3] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.