PARALLELISM IN ADA:
PROGRAM DESIGN AND MEANING

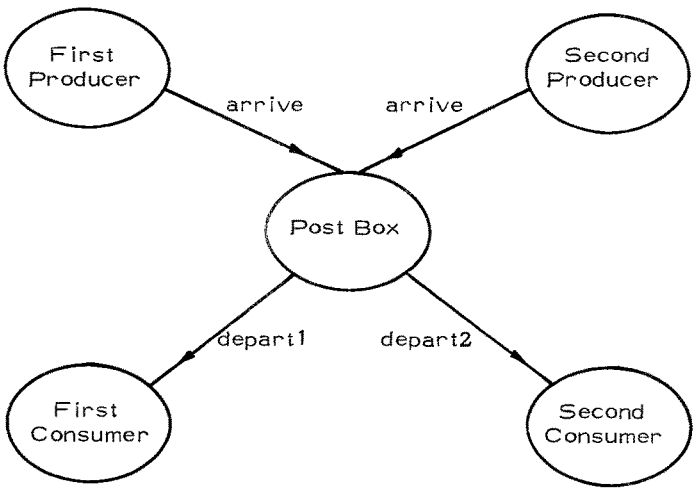
Brian H. Mayoh

Abstract: The art of designing parallel programs is underdeveloped because we do not understand parallelism clearly. This paper suggests a programming methodology and it gives a precise definition of the ADA form of parallelism. The methodology is based on ideas of Milner and it can be used when designing parallel programs in languages other than ADA.

Computer Science Department, Aarhus University, Ny Munkegade,
8000 Aarhus C, Denmark.

Part 1: DESIGN

The art of designing parallel programs is underdeveloped because we do not understand parallelism clearly. This paper suggests a programming methodology and it gives a precise definition of the ADA form of parallelism. The methodology is based on the ideas of Milner and it can be used when designing parallel programs in languages other than ADA. For us a parallel program consists of one or more tasks and several arrows from one task to another. We shall use the example of producers and consumers, communicating through a postbox, to illustrate our design method.



In all our pictures different arrows may have the same head but they always have different tails. From our picture for producers and consumers we see that the postbox decides which consumer to communicate with. If we reversed the arrows to the consumers, we would have the usual interplay between producers and consumers: a shared buffer.

The first phase of our design method is to draw a picture of tasks with names arrows between them. Arrows with the same head must have the same name. Because the ADA equivalent of a first phase picture is a list of partially defined task interfaces whose entries correspond to the heads of arrows, we shall hereafter use the word "entry" instead of "name of arrow".

For our example we have

```
task First Producer is
 -- calls entry: arrive
end;
```

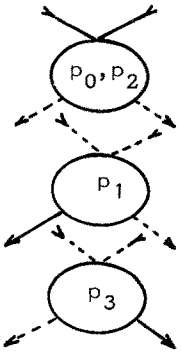
```
task Second Producer is
 -- calls entry: arrive
end;
```

```
task Post Box is
  entry arrive ....
   -- calls entries: depart1, deaprt2
end;
```

```
task First Consumer is
  entry depart1 ...
end;
```

```
task Second Consumer is
  entry depart2 ...
end;
```

The second phase of our design method is to describe the programs for each task to the extent of fixing the places where the task may communicate with another task. We can use a convention of Milner's (Milner (1978)) to indicate which task arrows are ready to communicate and which are not: dotted half arrows correspond to arrows that are not ready to communicate.



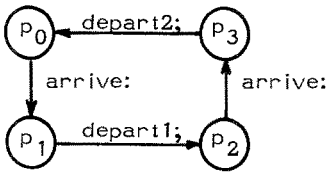
a task waiting for an entry

a task calling an entry

a task calling another entry

If a task has N arrows, then there are at most 2^N of these "Milner symbols", but each of them may correspond to many places in the task program. We can describe the interrelation of the places in a task program by giving a set of place equations. If we decide that our Post Box task should alternate between the two consumers, the set of place equations would be:

p_0 = arrive: p_1
p_1 = depart1; p_2
p_2 = arrive: p_3
p_3 = depart2; p_0



There are colons in the first and third equations because the entry "arrive" is i the interface of Post Box; there are semicolons in the other equations because the entries depart1 and depart2 are in the interface of other tasks. The formal definition of a place equation is

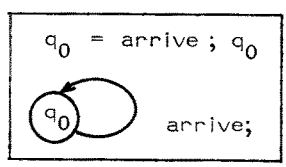
- a separator is a colon, semicolon or exclamation mark
- a guard is a sequence of letters and digits followed by a separator
- a capability is a guard followed by a subscripted letter
- the right side of a place equation is the sum of zero, one or more capabilities
- the left side of an equation is a subscripted letter.

The reader can think of the subscripted letters in place equations as critical places in the task program. The ADA equivalent of a set of place equations is a partically defined task body:

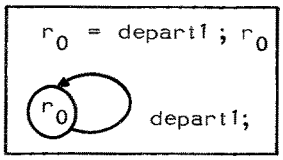
```
task body Post Box is
    begin
        loop          -- p0;
            accept arrive ...      -- p1;
            depart1 ...            -- p2;
            accept arrive ...      -- p3;
            depart2 ...
        end loop
    end Post Box;
```

For the other tasks in our example, the second phase in our design method might give

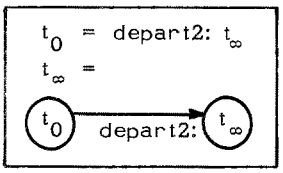
```
task body First Producer is      -- also Second Producer
    begin
        loop          -- q0;
            arrive ...
        end loop
    end First Producer;
```



```
task body First Consumer is
    begin
        loop        -- r0;
            accept depart1 ...
        end loop
    end First Consumer;
```



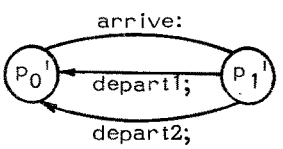

```
task body Second Consumer is
    begin                -- t0
        accept depart2 ...
                         -- t∞
    end Second Consumer
```



As one of the place equations for Second Consumer has an empty right side, this task can die. If we do not want this to cause the death of the other tasks we could rewrite the place equations of Post Box

$p_0' = \text{arrive: } p_1'$

$p_1' = \text{depart1; } p_0' + \text{depart2; } p_0'$



Our place equations are closely related to path expressions (Campbell, Habermann (1974)) and flow expressions (Shaw (1978)), our two sets of equations for Post Box have the flow expression solutions:

(arrive: depart1; arrive: depart2;)*
(arrive: (depart1; ∪ depart2;))*

Path expressions and flow expressions are one way of solving place equations but there are others.

Let us return to our design method. If we finished the second phase with partially defined ADA task bodies, then the third phase consists of programming the task initialisation, and the last phase consists of completing the definition of the task bodies by inserting statements, introducing variables and procedures etc. For reasons of space we shall not discuss these two phases except to say that place equations may be a convenient way of documenting the final program. Suppose we finish with

```
task body First Consumer is
    procedure Print Message is
        begin
            loop
                S1;                 -- r2
                exit when B;        -- r3
                S2;
            end loop;               -- r4
            S3;
        end Print Message;
    begin
        loop                        -- r0
            accept depart1;         -- r1
            Print Message;
        end loop;
    end First Consumer;
```

This can be documented by either

r_0 = depart1 : r_1

r_1 = Print Message; r_0

or

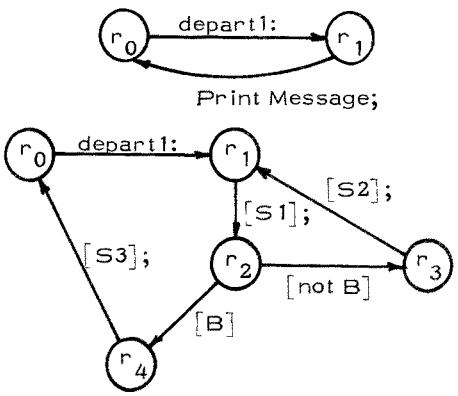
r_0 = depart1 : r_1

r_1 = [S1]; r_2

r_2 = [not B]; r_3 + [B]; r_4

r_3 = [S2]; r_1

r_4 = [S3]; r_0



In these place equations we see a new kind of guard: "[", then a sequential ADA construct, then "];". We can assume that the precise meaning of such guards wil be given by the formal semantics of ADA when it appears.

Part 2: MEANING

We want to give a meaning to a network of tasks with named arrows between them, when we have a set of place equations for each task in the network. We do this converting the place equations into network equations, then solving the network equations. Let us begin by describing how the place equations for two tasks are converted into network equations. Suppose the task First Producer has one place equation and one Milner symbol

q_0 = arrive; q_0

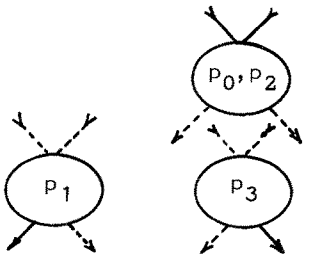
Suppose the task Post Box has four place equations and three Milner symbols

p_0 = arrive : p_1

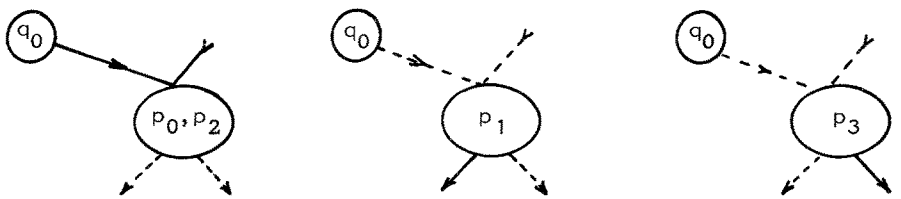
p_1 = depart1; p_2

p_2 = arrive : p_3

p_3 = depart2; p_0



Each choice of a Milner symbol from the two tasks gives a network symbol



The full arrow in the first symbol shows the possibility of communication, but communication may not actually happen because there is an undotted half-arrow. We must explain the significance of the labels in our symbols; each label in a task symbol denotes a place in the task program, each way of choosing a label in a network symbol denotes a configuration of the network. In our example we have:

configurations $q_0|p_0$ and $q_0|p_2$ from the first symbol

configurations $q_0|p_1$ and $q_0|p_3$ from the other symbols.

The network equations for $q_0|p_1$ and $q_0|p_3$ are no problem but the network equations for $q_0|p_0$ and $q_0|p_2$ must reflect the possibility of communication through the entry "arrive", they must have a capability for the rendezvous guard "arrive!". The network equations must be

$(q_0|p_0)$ = arrive! $(q_0|p_1)$ + arrive : $(q_0|p_1)$

$(q_0|p_1)$ = depart1; $(q_0|p_2)$

$(q_0|p_2)$ = arrive! $(q_0|p_3)$ + arrive : $(q_0|p_3)$

$(q_0|p_3)$ = depart2; $(q_0|p_0)$

These equations reflect the asymmetry of ADA's rendezvous concept - the guard "arrive :" is in these equations, but the guard "arrive ;" is not. Note also that the guards "depart1;" and "depart2;" remain in the network equation because they correspond to calls on entries that are not in the interfaces of First Producer and Post Box.

The precise rules for writing down the right side of the network equation for the left side $(p|q)$ are:

1) if the right side of the equation for p has the capability $\gamma p'$ and γ does not correspond to the call of an entry in the q task, then $(p|q)$ has the capability $\gamma(p'|q)$

2) if the right side of the equation for p has the capability $\gamma q'$ and γ does not correspond to the call of an entry in the p task, then $(p|q)$ has the capability $\gamma(p|q')$

3) if the right side of the equation for p has the capability $\alpha:p'$ and the right side of the equation for q has the capability $\alpha;q'$ then $(p|q)$ has the capability $\alpha!(p'|q')$

4) if the right side of the equation for p has the capability $\alpha;p'$ and the right side of the equation for q has the capability $\alpha:q'$ then $(p|q)$ has the capability $\alpha!(p'|q')$

5) the right side of the network equation for $(p|q)$ is the sum of the capabilities given by the other rules.

Because these rules are associative and symmetric, the network equations for a network of many tasks do not depend on the order in which the tasks are combined - except for the names of configurations $((p|q)|r)$ for $(p|(q|r))$, $(p|q)$ for $(q|p)$ etc.

For our producer-consumer example we get the following equations from rules 1) to 5) (and the fact that there are no other tasks in the network so colon capabilities can be dropped):

$(p_0|r_0|t_0)$ = arrive! $(p_1|r_0|t_0)$

$(p_0|r_0|t_\infty)$ = arrive! $(p_1|r_0|t_\infty)$

$(p_0|r_1|t_0)$ = arrive! $(p_1|r_1|t_0)$ + Print Message; $(p_0|r_0|t_0)$

$(p_0|r_1|t_\infty)$ = arrive! $(p_1|r_1|t_\infty)$ + Print Message; $(p_0|r_0|t_\infty)$

$(p_1|r_0|t_0)$ = depart1! $(p_2|r_1|t_0)$

$(p_1|r_0|t_\infty)$ = depart1! $(p_2|r_1|t_\infty)$

$(p_1|r_1|t_0)$ = Print Message; $(p_1|r_0|t_0)$

$(p_1|r_1|t_\infty)$ = Print Message; $(p_1|r_0|t_\infty)$

$(p_2|r_0|t_0)$ = arrive! $(p_3|r_0|t_0)$

$(p_2|r_0|t_\infty)$ = arrive! $(p_3|r_0|t_\infty)$

$(p_2|r_1|t_0)$ = arrive! $(p_3|r_1|t_0)$ + Print Message; $(p_2|r_0|t_0)$

$(p_2|r_1|t_\infty)$ = arrive! $(p_3|r_1|t_\infty)$ + Print Message; $(p_2|r_0|t_\infty)$

$(p_3|r_0|t_0)$ = depart2! $(p_0|r_0|t_\infty)$

$(p_3|r_0|t_\infty)$ =

$(p_3|r_1|t_0)$ = depart2! $(p_0|r_1|t_\infty)$ + Print Message; $(p_3|r_0|t_0)$

$(p_3|r_1|t_\infty)$ = Print Message; $(p_3|r_0|t_\infty)$

These equations have the flow expression solution

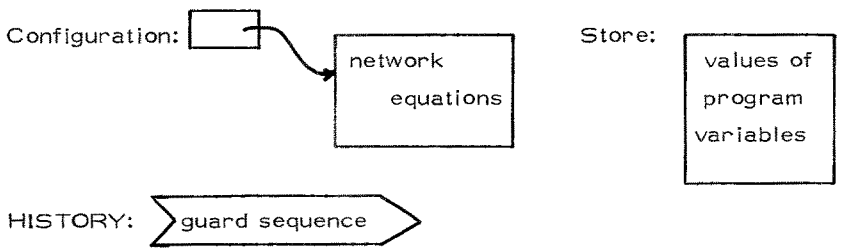
arrive! ((depart1! arrive! depart2! arrive!)$\odot$ Print Message;)
(arrive! $\odot$ Print Message;)

where $\odot$ is the shuffle operator of formal language theory (Ginsburg (1977)). The reader, who recognizes that the concurrency constraints in our example can be captured by the simple path expression

path arrive; depart1; arrive; depart 2 end
-- implicit iteration, and semicolons for sequentiality

may well ask why we have given complicated rules for forming network equations from place equations. The answer is that this enables us to indicate a semantics for ADA parallelism.

Let us begin with an operational semantics for network equations. We allow for equations with more than one capability by using a HISTORY file to resolve non-determinism. We assume that a state is given by the values of variables HISTORY, Configuration and Store:



We say that a state is jammed if HISTORY is empty or the first guard of HISTORY does not occur on the right side of the network equation given by Configuration. If a state is not jammed, the next state is defined by:

1) the first guard γ in HISTORY gives a capability in the network equation given by Configuration;

2) the new value of Configuration is given by this capability;

3) the values of Store is changed as directed by γ;

4) the first guard in HISTORY is deleted.

Example Consider the state where Configuration gives the network equation

$$(p_2|r_1|t_0) = \text{arrive!}\ (p_3|r_1|t_0) + \text{Print Message;}\ (p_2|r_0|t_0)$$

If the first guard in HISTORY is "arrive!", then $(p_3|r_1|t_0)$ is the next value of Configuration; if the first guard in HISTORY is "Print Message;", then $(p_2|r_0|t_0)$ is the new value of Configuration; otherwise the state is jammed.

In the draft formal definition of ADA denotational semantics are used to give a precise meaning to the sequential part of the language. In essence this formal definition gives a function $\mathcal{L}[\![\gamma]\!]$ for every guard γ in the network equations for an ADA program. Because exceptions and other language constructs that affect the run-time behaviour of a program must be defined, the function $\mathcal{L}[\![\gamma]\!]$ maps continuations into continuations, not stores into stores:

$$\text{Local Continuation} = \text{Store} \to \text{Answer}$$
$$\mathcal{L}[\![\gamma]\!] : \text{Local Continuation} \to \text{Local Continuation}$$

We could define parallelism in ADA by introducing the domains

$$\text{State} = \text{History} \times \text{Configuration} \times \text{Store}$$
$$\text{Global Continuation} = \text{State} \to \text{Answer}$$

and defining a function $\mathbb{M}$ from Global Continuations to global continuations for each set of network equations.

For each guard γ we can extend $\mathcal{L}[\![\gamma]\!]$ to:

$\mathcal{G}[\![\gamma]\!]$: Global Continuation → Global Continuation

$\mathcal{G}[\![\gamma]\!]\ (\phi)\ [h, c, s] = \mathcal{L}[\![\gamma]\!]\ (\Theta)\ [s]$ where $\Theta[s'] = \phi[h, c, s']$

For each set of network equations we define

- the predicate Jammed$[h, c]$ as:
  h is not empty and its first guard gives a capability in the network equation given by c;
- the function Next Configuration: Guard x Configuration → Configuration;
- the function $\mathbb{m}$: Global Continuation → Global Continuation as:

$$\mathbb{m}(\phi)[h, c, s] = \underline{\text{if}}\ \text{Jammed}[h, c]\ \underline{\text{then}}\ \phi[h, c, s]$$
$$\underline{\text{else}}\ \mathcal{G}[\![\gamma]\!]\ (\phi)\ [h', \text{Next Configuration}(\gamma, c), s]$$
$$\underline{\text{where}}\ h = (\gamma, h')$$

Example ctd. Consider the state s where Configuration gives the network equation

$$(p_2 | r_1 | t_0) = \text{arrive!}\ (p_3 | r_1 | t_0) + \text{Print Message};\ (p_2 | r_0 | t_0)$$

and the first guard in HISTORY is "arrive!". Our definition gives:

$h = (\text{arrive!}, h')$

Jammed$[h, (p_2 | r_1 | t_0)]$ is FALSE

Next Configuration$[\text{arrive!}, (p_2 | r_1 | t_0)] = (p_3 | r_1 | t_0)$

$\mathbb{m}(\phi)[h, (p_2 | r_1 | t_0), s] = \mathcal{G}[\![\text{arrive!}]\!](\phi)[h', (p_3 | r_1 | t_0), s]$
$= \mathcal{L}[\![\text{arrive!}]\!]\ \Theta\ s$ where $\Theta[s'] = \phi[h', (p_3 | r_1 | t_0), s']$

Now we have defined the function $\mathbb{m}$ for a set of network equations, we can explain how it gives the "meaning" of a parallel ADA program. For any global continuation ϕ_0 the sequence

$$\phi_0, \phi_1 = \mathbb{m}(\phi_0), \phi_2 = \mathbb{m}(\phi_1) \ldots$$

has a least upper bound ϕ_∞ that solves the network equations. Suppose we take the truth values as the domain Answer and we define $\phi_0(h, c, s)$ as the predicate:

the network for c has empty right side & h is empty
& s satisfies some predicate FINAL.

In this case $\phi_\infty(h, c, s)$ will be the predicate:

if we start with store s in configuration c, we will use all of HISTORY h and finish in a no capability equation with a store satisfying FINAL.

This predicate ϕ_∞ is the meaning of the parallel ADA program corresponding to the network equations. There is a close connection between our ϕ_∞-solution of the network equations and the flow expression solution we gave earlier:

> if $\phi_\infty(h, c, s)$ is satisfied, then h satisfies the flow expression solution for c.

The converse of this need not hold because some histories given by the flow expression solution may fail for some values of s.

In the expanded version of this paper (Mayoh (1979)) we indicate

- how some histories are forbidden by such ADA restrictions as first-in-first-out queues on entries;
- how initiation and termination of tasks can be expressed using network equations;
- how scope, visibility and parameters can be incorporated;
- a Petri net justification of our assumption that the true parallelism of ADA can be captured by our non-deterministic network equations.

There seems to be no good reason why a complete formal semantics of ADA can not be based on the approach in this paper: separating the question of what computation histories are possible from the question of defining the result of a computation for a given history.

In the formal semantics of CSP (Francez, Hoare, Lehmann, De Roever (1979)) we find the same approach, but there are important differences. In their terminology we advocate (1) using continuations not power domains - replacing $P_{E-M}((S_1 \times \ldots \times S_m) \cup (\perp, \text{fail, deadlock}))$ by $S_1 \times \ldots \times S_m \rightarrow \text{Answer}$, (2) attaching communication histories to a set of concurrent processes as a whole, not to each concurrent process in the set. Another paper with an approach like ours is (Hoare (1979)); it contains a detailed investigation of histories for communicating processes, our place and network equation sets are special cases of its processes.

References

ADA (1979a) Preliminary ADA reference manual. SIGPLAN Notices 14 no. 6, part A.

ADA (1979b) Rational for the design of the ADA programming language. SIGPLAN Notices 14 no. 6, part B.

Campbell R.H., Habermann, A.N. (1974): "The specification of process synchronization by path expressions". Springer Lecture Notes 16.

Francez, N., Hoare, C.A.R., Lehmann, D.J., De Roever, W.P. (1979): "Semantics of Nondeterminism, Concurrency, and Communication". J. Comp. Sys. Sci. 19, pp. 290-308.

Ginsburg, S. (1977): "The Mathematical Theory of Context Free Languages". McGraw Hill, p. 108.

Hoare, C.A.R. (1979): A model for communicating sequential processes. Program Research Group preprint, Oxford University.

Mayoh, B.H. (1979): "Parallelism in ADA", DAIMI PB-103, Aarhus University.

Milner, R. (1978): "An Algebraic Theory for Synchronization". Springer Lecture Notes 67.

Shaw, A.C. (1978): "Software Descriptions with Flow Expressions". IEEE Trans. Software Engineering Se-4 no. 3.