ISSN 0105-8517

PARALLELISM IN ADA :

PROGRAM DESIGN AND MEANING

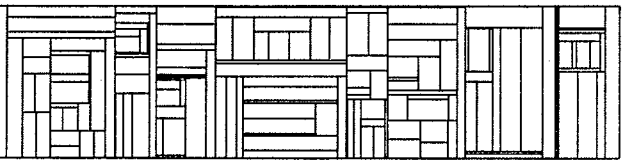
by

Brian Mayoh

DAIMI PB-103

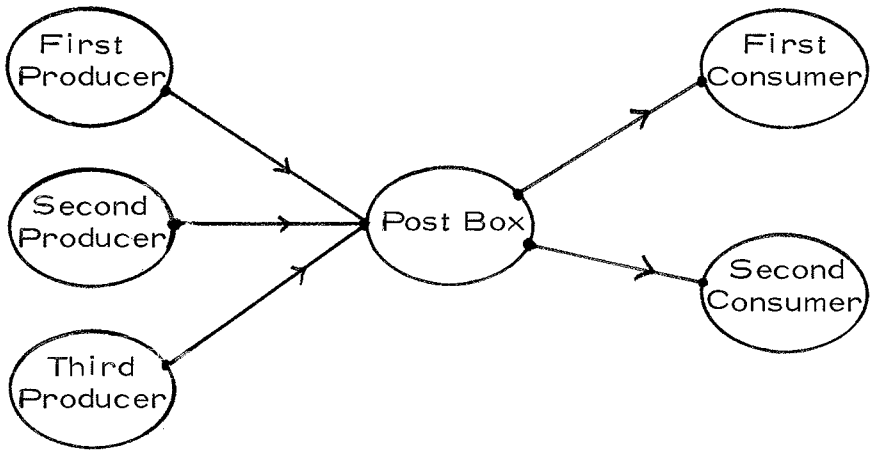
September 1979

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



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 post box 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 named 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

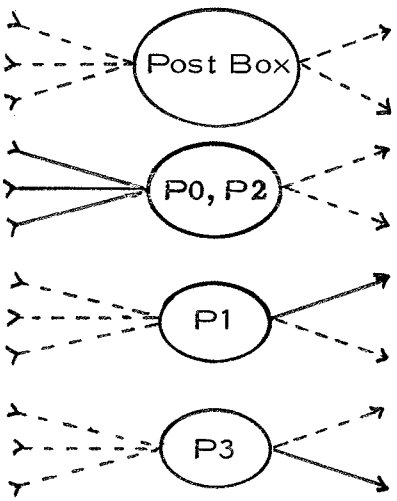
```
task Postbox is
    entry arrive ....
    -- calls entries; depart 1 and depart 2
end task Postbox interface;

task First Consumer is
    entry depart 1 ....
end task First Consumer interface;

task Second Consumer is
    entry depart 2
end task Second Consumer interface;

task First Producer is
    -- calls entry : arrive
end task First Producer;
    -- other producers similar.
```

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 [ ] 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 not ready for any communication

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 "ready to communicate" symbols, but each of them may correspond to many places in the task program; the bound on the number of Milner Symbols is not a bound on the complexity of a task program.

A useful way of describing the task programs during the second phase of our method is to give a set of process equations. (A process equation expresses a process as a sum of capabilities.) The left side is a letter that denotes a process, and the right side is a sequence of zero, one, or more capabilities separated by "+". Any sequence of letters followed by a colon or semicolon is a port, and a port followed by a letter denoting a process is a capability. The set of process equations for the task Post Box in our example might be

p_0 = arrive : p_1
p_1 = depart 1; p_2
p_2 = arrive : p_3
p_3 = depart 2; p_0

The ports for the arrows to (from) Post Box end with a colon (semicolon).

For the producer tasks we might have the one equation

q_0 = arrive; q_0

The task First Consumer might have the two equations

r_0 = depart 1 ; r_1
r_1 = Print Message; r_0

and the task Second Consumer might have the two equations

t_0 = depart 2 : t_∞
t_∞ =

where the last equation corresponds to the death of the task because its right side is empty (consists of no capabilities). If we were concerned by the fact that post box cannot accept more than one communication after the death of Second Consumer, we could change the post box equations to :

p'_0 = arrive : p'_1
p'_1 = depart 1 ; p'_0 + depart 2 ; p'_0

so that Post box can handle any number of communications. Here we are presenting a design method, not trying to design a good producer-consumer program. One aspect of the second phase of our method is deciding which producers are available to a task program and where they are to be used. The port "Print Message;" in the First Consumer process equation could well denote the call of a procedure named "Print Message". For the same reason as the ADA designers, we do not distinguish between entry calls and procedure calls - the ports for both end with semicolons. The ADA equivalent of a set of process equations is a partially defined task body. For the producer-consumer example we would have

```
task body First Producer is
    begin
        loop
            -----
            arrive
            -----
        end loop
    end
end First Producer.

    -- other producers similar

task body Postbox is
    begin
        loop
            accept arrive ...
            depart 1 ...
            accept arrive
            depart 2 ...
        end loop
    end Postbox
```

```
task body First Consumer is
    procedure Print Message....
    begin
        loop
            accept depart 1 ...
            Print Message;
        end loop;
    end First Consumer

task body Second Consumer is
    begin
        accept depart 2 ...
    end Second Consumer;
```

If we were concerned about the death of Second Consumer we would rewrite the Postbox as

```
task body Postbox is
    begin
        loop
            accept arrive
            depart 1 ...
        end loop
    end Poxtbox;
```

(Note for ADA experts : we cannot express depart 1; p'_0 + depart 2; p'_0 clearly because select only works for a task's own entries.)

Let us return to the sets of process equations for a task. If we suppose that a process is a predicate on sequences of parts, we can solve a set of process equations. The solutuion to the equations will give a predicate Path Set(p) to each right-side p; Path Set(p) τ will be true if the path sequence τ enjoys this predicate, otherwise Path Set(p) τ will be false. The equation sets for our producer consumer example have the solution :

Path Set (p_0) τ $\equiv$ τ is an infinite sequence of repetitions of arrive : depart 1 ; arrive : depart 2.

Path Set (p_1) τ $\equiv$ τ is an infinite sequence of repetitions of depart 1 ; arrive : depart 2 ; arrive :

Path Set (p_2) τ $\equiv$ τ is an infinite sequence of repetitions of arrive : depart 2 ; arrive : depart 2;

Path Set (p_3) τ $\equiv$ τ is an infinite sequence of repetitions of depart 2 ; arrive : depart 1 ; arrive :

Path Set (q_0) τ $\equiv$ τ is an infinite sequence of repetitions of arrive ;

Path Set (r_0) τ $\equiv$ τ is an infinite sequence of repetitions of depart 1 : print message ;

Path Set (r_1) τ $\equiv$ τ is an infinite sequence of repetitions of print message ; depart 1 :

Path Set (t_0) τ $\equiv$ τ is the sequence (depart 2 :)

Path Set (t_1) τ $\equiv$ τ is the empty port sequence ()

The reader should not confuse the predicate HALT that is only true for the empty port sequence with the predicate DEADLOCK which is never true.

The third and last phase of our design method is to complete the task program by describing variables and procedures, inserting statements and so on. We can use process equations to document this design phase, too. Suppose we replace the procedure Print Message in our task First Consumer by the ADA statements :

```
loop
    S1;
    exit when B;
    S2;
end loop;
S3;
```

We can document this by replacing the process equation

$$r_1 = \text{Print-Message}; r_0$$

by the four equations

$$r_1 = [S1]; r_2$$
$$r_2 = [\underline{\text{not}}\ B]; r_3 + [B]; r_4$$
$$r_3 = [S2]; r_1$$
$$r_4 = [S3]; r_0$$

In these new equations we use ports that begin with "[", continue with program, and end with "];". If a new equation contains more than one capability, then the choice between them is determined by the value of the variables, when the thread of control is at the corresponding place in the program. When a program equation is replaced, we may get much useful information about possible computation paths by solving the new set of equations with the predicate HALT assigned to the processes that are not on the left side of a new equation. In our example solving the new equations with HALT assigned to r_0 gives

Path Set $(r_1)\tau \equiv (\tau$ is $[S_1]; \tau')$ and Path Set $(r_2)\tau'$

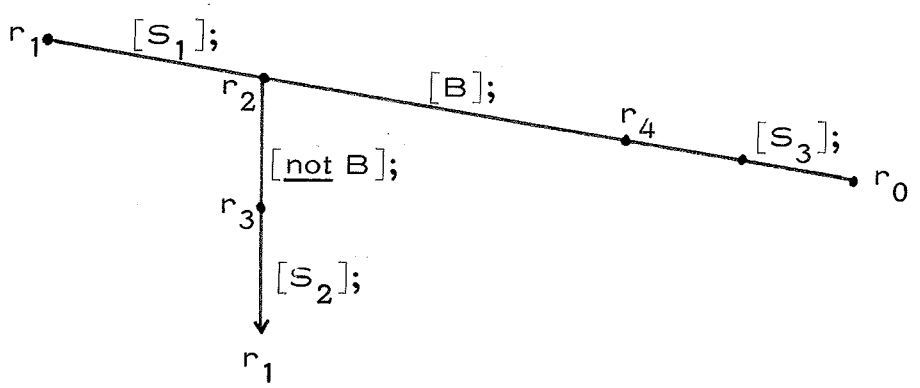
Path Set $(r_2)\tau \equiv \tau$ is infinite sequence of repetitions of
$[\underline{\text{not}}\ B]; [S_2]; [S_1];$

or τ is finite sequence of repetitions of
$[\text{not } B]; [S_2]; [S_1];$ followed by $[\quad]; [S_3];$

Path Set $(r_3)\tau \equiv (\tau$ is $[S_2]; [S_1]; \tau')$ and Path Set $(r_2)\tau'$

Path Set $(r_4)\tau \equiv \tau$ is the sequence $([S_3];)$

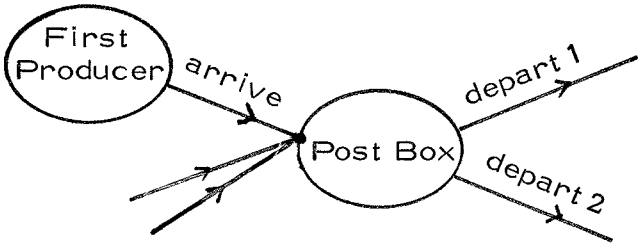
Sometimes trees are a useful way of representing path set solutions of equations :



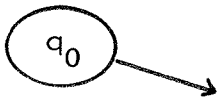
It does not seem unreasonable to assume that the reader knows how to convert a sequential program or flow diagram into a set of equations because she has met the method in the context of program verification, predicate transformer semantics, or dynamic logic. Processes in our equations correspond to other authors' "program control points", the ports in our equations correspond to other author's "elementary actions".

Part 2 : MEANING

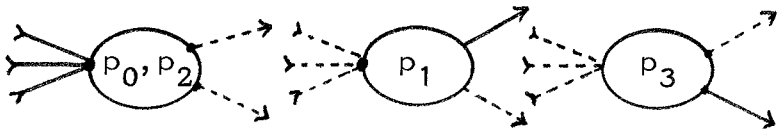
How can we give a precise meaning to a network of tasks with named arrows between them, when we have a set of process equations for each task in the collection ? By building one big network equation set from the task equation sets, then solving the network equation set. Let us begin by describing how equation sets are combined. Suppose we have equation sets for the tasks, First-Producer and Post Box, and these tasks are combined in the network



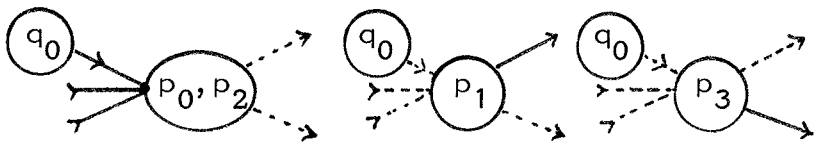
Suppose the equation set for First-Producer gives the Milner symbol



and the equation set for Post Box gives the Milner symbols



The network equation set will give the Milner symbols



where the full arrow in the first symbol shows the possibility of communication, but communication may not actually happen because the first symbol also has half arrows that are not dotted. Note the labels in our symbols, each label in a one task symbol denotes a process in the task, each way of choosing a label at a task in a network symbol denotes a process in the network.

For our network consisting of First Producer and Post Box we have

two processes which we denote $q_0|p_0$ and $q_0|p_2$ from the first symbol,

and two processes which we denote $q_0|p_1$ and $q_0|p_3$ from the other symbols.

The network equations for $q_0|p_1$ abd $q_0|p_3$ are no problem, but the network equations for $q_0|p_0$ and $q_0|p_2$ must reflect the possibility of communicating through the entry "arrive", they must have the rendezvous port "arrive!". If the task equation sets were

$$\{q_0 = \text{arrive} ; q_0\}$$

$$\{p_0 = \text{arrive}:p_1 , p_1 = \text{depart 1};p_2 , p_2 = \text{arrive}:p_3, p_3 = \text{depart 2}; p_0\}$$

then the network equations would be

$$(q_0 \mid p_0) = \text{arrive!}\ (q_0 \mid p_1) + \text{arrive} : (q_0 \mid p_1)$$

$$(q_0 \mid p_1) = \text{depart 1} ; (q_0 \mid p_2)$$

$$(q_0 \mid p_2) = \text{arrive!}\ (q_0 \mid p_1) + \text{arrive} : (q_0 \mid p_1)$$

$$(q_0 \mid p_3) = \text{depart 2} ; (q_0 \mid p_0)$$

The port "arrive:" is in these equations, but the port "arrive;" is not - the equations reflect the assymmetry of ADA's rendezvous concept. Note also that the ports "depart 1;" and "depart 2;" remain in the network equations because they correspond to calls on entries outside the network.

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

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

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

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

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

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

Because these rules are associative and symmetric in p and q, 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 the network process variables, i.e. $((p \mid q) \mid r)$ for $(p \mid (q \mid r))$, $(p \mid q)$ for $(q \mid p)$]. Several applications of the rules in our producer-consumer example give the equations.

$(p_0 \mid r_0 \mid t_0)$ = arrive! $(p_1 \mid r_0 \mid t_0)$ + arrive : $(p_1 \mid r_0 \mid t_0)$
+ depart 1 : $(p_0 \mid r_1 \mid t_0)$ + depart 2 : $(p_0 \mid r_0 \mid t_\infty)$

$(p_0 \mid r_0 \mid t_\infty)$ = arrive! $(p_1 \mid r_0 \mid t_\infty)$ + arrive : $(p_1 \mid r_0 \mid t_\infty)$ + depart 1 : $(p_0 \mid r_1 \mid t_\infty)$

$(p_0 \mid r_1 \mid t_0)$ = arrive! $(p_1 \mid r_1 \mid t_0)$ + arrive : $(p_1 \mid r_1 \mid t_0)$
+ Print Message ; $(p_0 \mid r_0 \mid t_0)$ + depart 2 : $(p_0 \mid r_1 \mid t_\infty)$

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

$(p_1 \mid r_0 \mid t_0)$ = depart 1 ! $(p_2 \mid r_1 \mid t_0)$ + depart 1 : $(p_1 \mid r_1 \mid t_0)$ + depart 2 : $(p_1 \mid r_1 \mid t_\infty)$

$(p_1 \mid r_0 \mid t_\infty)$ = depart 1 ! $(p_2 \mid r_1 \mid t_\infty)$ + depart 1 : $(p_1 \mid r_1 \mid t_\infty)$

$(p_1 \mid r_1 \mid t_0)$ = Print Message; $(p_1 \mid r_0 \mid t_0)$ + depart 2 : $(p_1 \mid r_1 \mid t_\infty)$

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

$(p_2 \mid r_0 \mid t_0)$ = arrive! $(p_3 \mid r_0 \mid t_0)$ + arrive : $(p_3 \mid r_0 \mid t_0)$
+ depart 1 : $(p_2 \mid r_1 \mid t_0)$ + depart 2 : $(p_2 \mid r_0 \mid t_\infty)$

$(p_2 \mid r_0 \mid t_\infty)$ = arrive! $(p_3 \mid r_0 \mid t_\infty)$ + arrive: $(p_3 \mid r_0 \mid t_\infty)$ + depart 1 : $(p_2 \mid r_0 \mid t_\infty)$

$(p_2|r_1|t_0)$ = arrive! $(p_3|r_1|t_0)$ + arrive : $(p_3|r_1|t_0)$
+ Print Message; $(p_2|r_0|t_0)$ + depart 2 : $(p_2|r_1|t_\infty)$

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

$(p_3|r_0|t_0)$ = depart 2! $(p_0|r_0|t_\infty)$ + depart 2 : $(p_3|r_0|t_\infty)$
+ depart 1 : $(p_3|r_1|t_0)$

$(p_3|r_0|t_\infty)$ = depart 1 : $(p_3|r_1|t_\infty)$

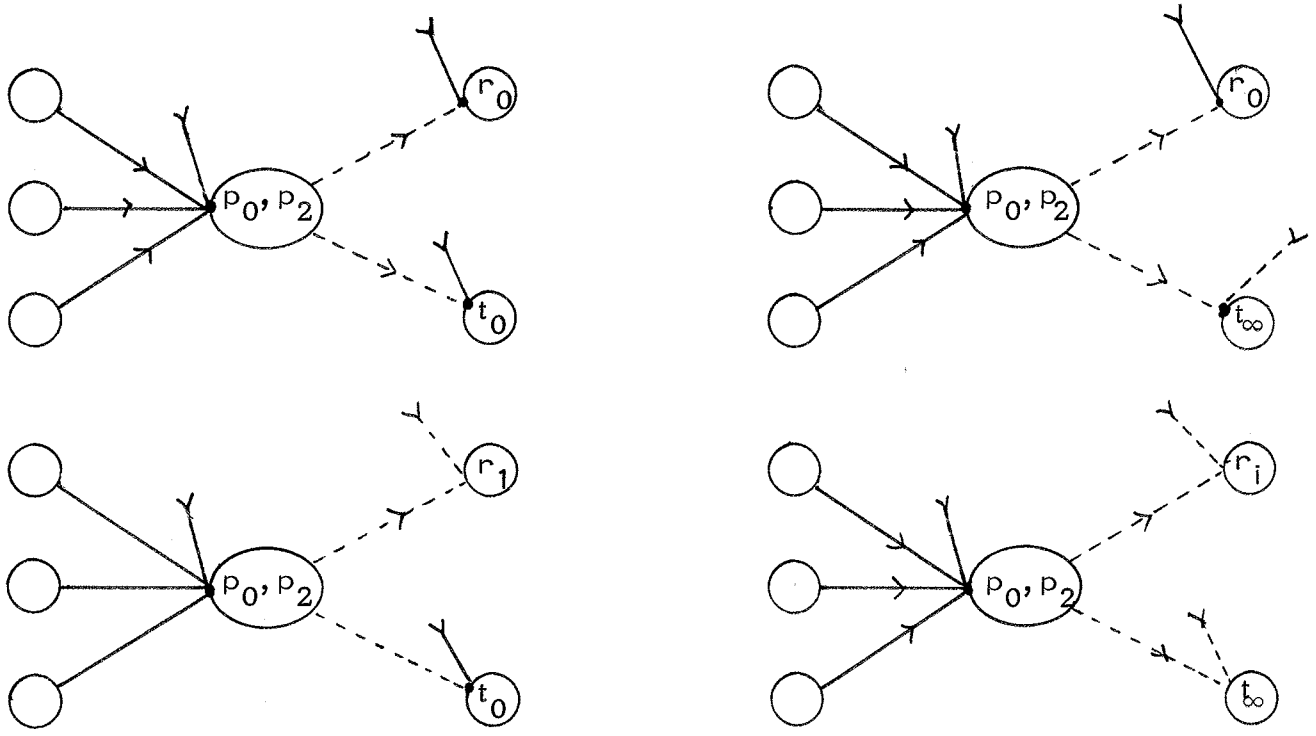
$(p_3|r_1|t_0)$ = depart 2! $(p_3|r_1|t_1)$ + depart 2 : $(p_3|r_0|t_\infty)$
+ Print Message; $(p_3|r_0|t_0)$

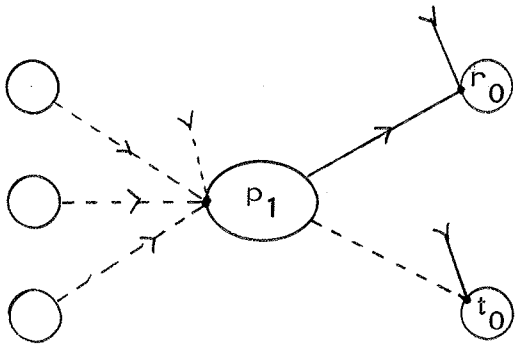
$(p_3|r_1|t_\infty)$ = Print Message; $(p_3|r_0|t_\infty)$

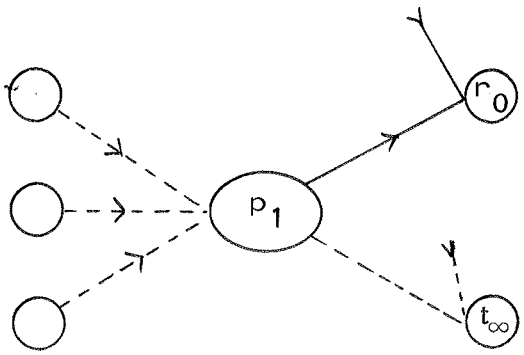
In these equations we have used $(p_0|r_0|t_0)$ instead of the more strict

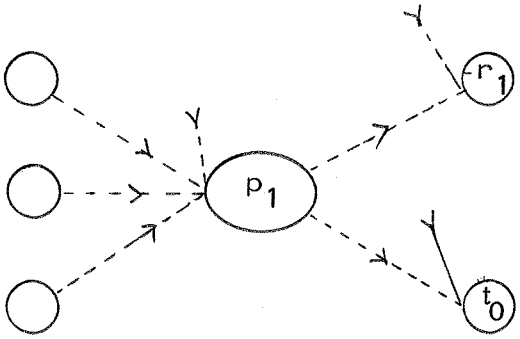
$$(q_0)\ (q_0')\ (q_0'')\ (p_0)\ r_0)\ t_0)))))$$

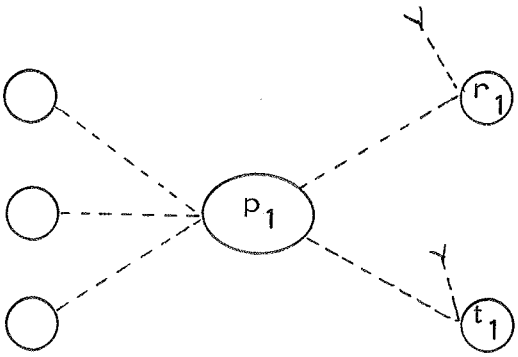
We have shortened names in accordance with the network Milner symbols :

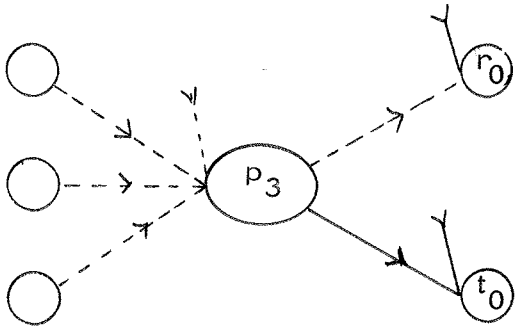


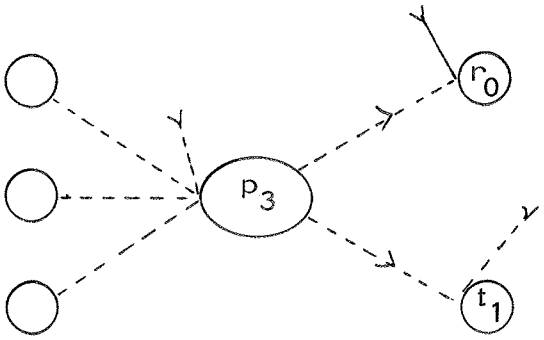

r_0
p_1
t_0

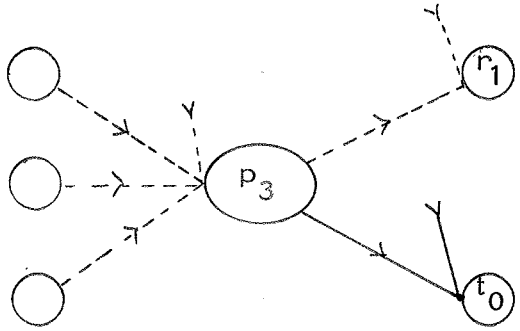

r_0
p_1
t_∞

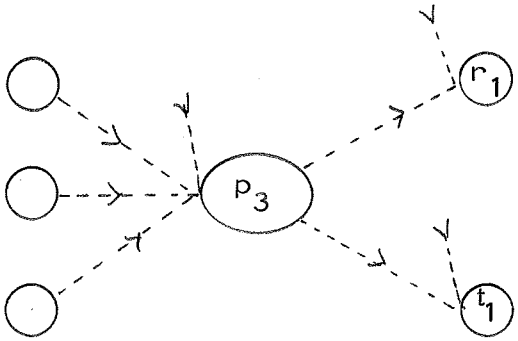

r_1
p_1
t_0


r_1
p_1
t_1


r_0
p_3
t_0


r_0
p_3
t_1

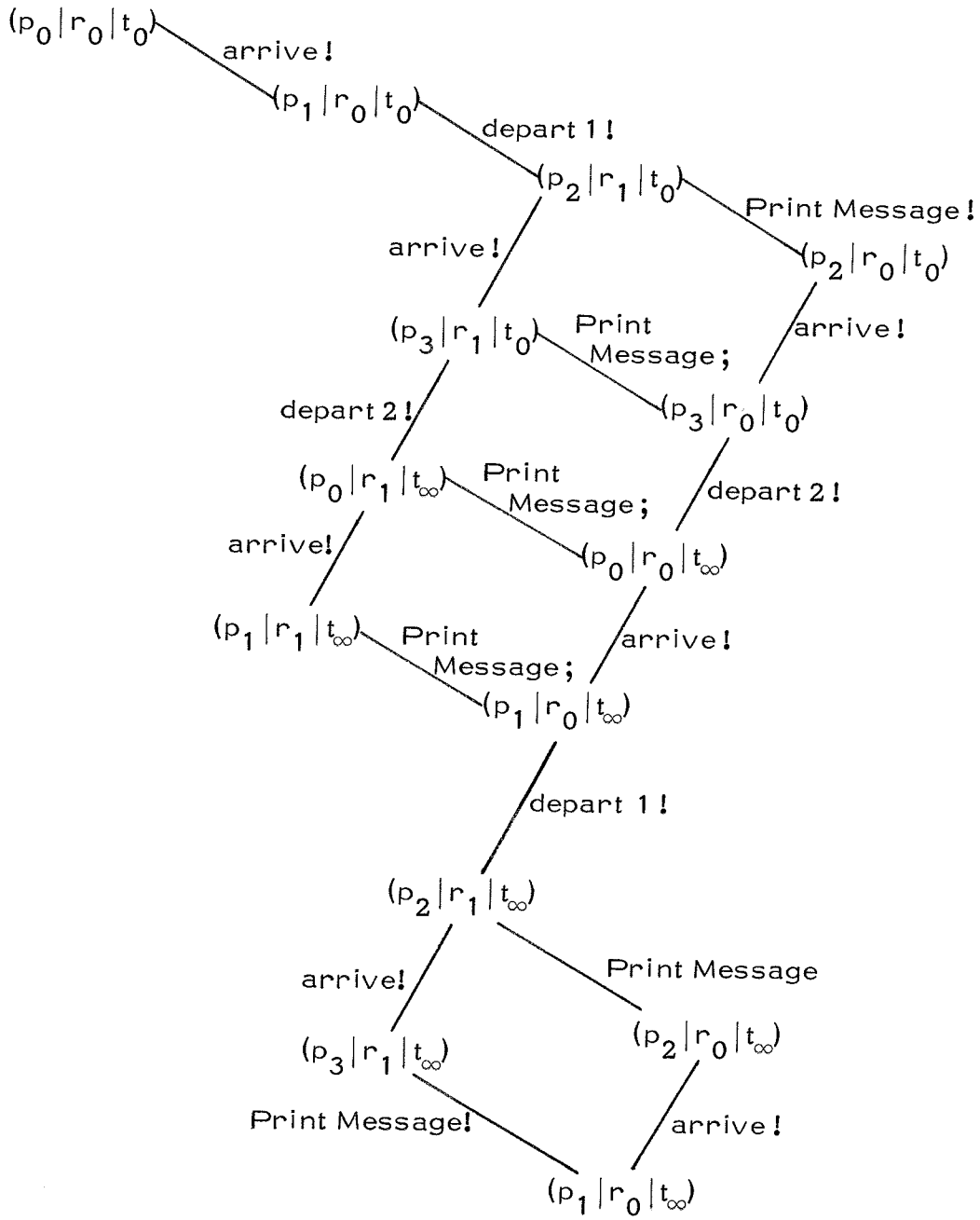

r_1
p_3
t_0


r_1
p_3
t_1

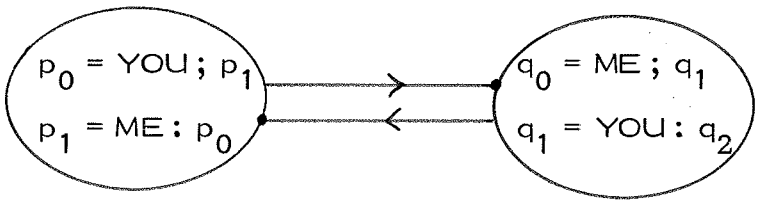
How do we change the network equations, when we realize that there are no more tasks in the network ? We remove all capabilities of the form $\alpha : p$, we destroy all the half-arrows in the Milner symbols. This changes the network equations for our producer-consumer example to

$$(p_0|r_0|t_0) = \text{arrive!}\ (p_1|r_0|t_0)$$
$$(p_0|r_0|t_\infty) = \text{arrive!}\ (p_1|r_0|t_\infty)$$
$$(p_0|r_1|t_0) = \text{arrive!}\ (p_1|r_1|t_0) + \text{Print Message};\ (p_0|r_0|t_0)$$
$$(p_0|r_1|t_\infty) = \text{arrive!}\ (p_1|r_1|t_\infty) + \text{Print Message};\ (p_0|r_0|t_\infty)$$
$$(p_1|r_0|t_0) = \text{depart 1!}\ (p_2|r_1|t_0)$$
$$(p_1|r_0|t_\infty) = \text{depart 1!}\ (p_2|r_1|t_\infty)$$
$$(p_1|r_1|t_0) = \text{Print Message};\ (p_1|r_0|t_0)$$
$$(p_1|r_1|t_\infty) = \text{Print Message};\ (p_1|r_0|t_\infty)$$
$$(p_2|r_0|t_0) = \text{arrive!}\ (p_3|r_0|t_0)$$
$$(p_2|r_0|t_\infty) = \text{arrive!}\ (p_3|r_0|t_\infty)$$
$$(p_2|r_1|t_0) = \text{arrive!}\ (p_3|r_1|t_0) + \text{Print Message};\ (p_2|r_0|t_0)$$
$$(p_2|r_1|t_\infty) = \text{arrive!}\ (p_3|r_1|t_\infty) + \text{Print Message};\ (p_2|r_0|t_\infty)$$
$$(p_3|r_0|t_0) = \text{depart 2!}\ (p0|r_0|t_\infty)$$
$$(p_3|r_0|t_\infty) =$$
$$(p_3|r_1|t_0) = \text{depart 2!}\ (p_0|r_1|t_1) + \text{Print Message};\ (p_3|r_0|t_0)$$
$$(p_3|r_1|t_\infty) = \text{Print Message};\ (p_3|r_0|t_\infty)$$

If we assign the predicate HALT to the no capability process $(p_3|r_0|t_\infty)$, we have the path set solution


$(p_0|r_0|t_0)$
arrive!
$(p_1|r_0|t_0)$
depart 1!
$(p_2|r_1|t_0)$
Print Message!
$(p_2|r_0|t_0)$
arrive!
$(p_3|r_1|t_0)$
Print Message;
arrive!
$(p_3|r_0|t_0)$
depart 2!
$(p_0|r_1|t_\infty)$
Print Message;
depart 2!
arrive!
$(p_0|r_0|t_\infty)$
$(p_1|r_1|t_\infty)$
Print Message;
arrive!
$(p_1|r_0|t_\infty)$
depart 1!
$(p_2|r_1|t_\infty)$
arrive!
Print Message
$(p_3|r_1|t_\infty)$
$(p_2|r_0|t_\infty)$
Print Message!
arrive!
$(p_1|r_0|t_\infty)$

We should have solved the equations with the always false predicate DEADLOCK assigned to the process $(p_3|r_0|t_1)$, because the termination of a network should imply termination of every task in the network. Consider the example



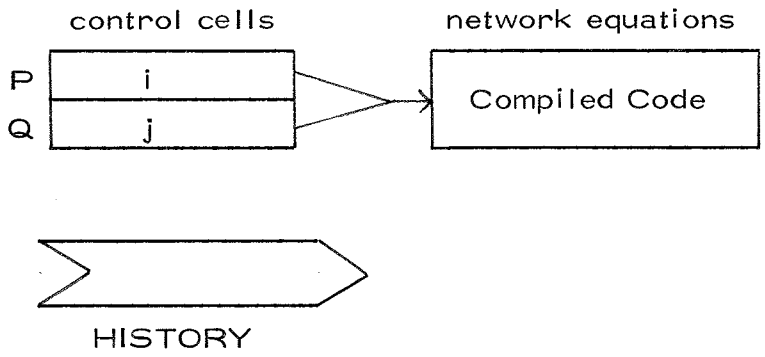
The appropriate path set solution of the network equations

$$(p_0|q_0) =$$
$$(p_0|q_1) = \text{YOU!}\ (p_1|q_0)$$
$$(p_1|q_0) = \text{ME!}\ (p_0|q_1)$$
$$(p_1|q_1) =$$

has "infinite repetition of YOU! ME!" for $(p_0|q_1)$
"infinite repetition of ME! YOU!" for $(p_1|q_0)$
and DEADLOCK for $(p_0|q_0)$ and $(p_1|q_1)$.

The predicate HALT should only be assigned to a no capability network process (a|b ... |c) when none of the task processes a, b ... c has a capability.

There is a close connection between realisation of parallel programs in a computer and path set solution of task and network equations. Suppose we have control cells, P and Q, a (fictive) file HISTORY, and the network equations in the computer.



Suppose the computer is in some "control state" with integers i, j in the cells P and Q. Suppose we say that the control state is jammed, if the first port in HISTORY does not occur on the right side of the network equation for process $(p_i | q_j)$. If the computer is not jammed in our control state $\langle i, j \rangle$ there is a capability

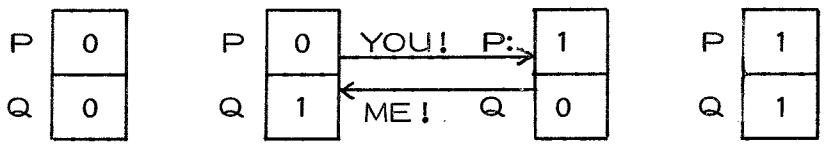
$$\gamma(p_k | q_l) \text{ where } \gamma \text{ is the first port in HISTORY}$$

on the right side of the network equation for process $(p_i | q_j)$. There is a next control state for each of these capabilities

- put the integer k in the cell P
- put the integer l in the cell Q
- remove the first port in HISTORY

Having many next control states is inconvenient and unimplementable; instead we suppose the ports in HISTORY can be decorated with integers that determine which capability should be chosen, e.g. if γ_2 is the first port in HISTORY, then the next control state is given by the second occurence of the port γ on the right side of the relevant equation.

Given the network equations we can draw a control state transition diagram



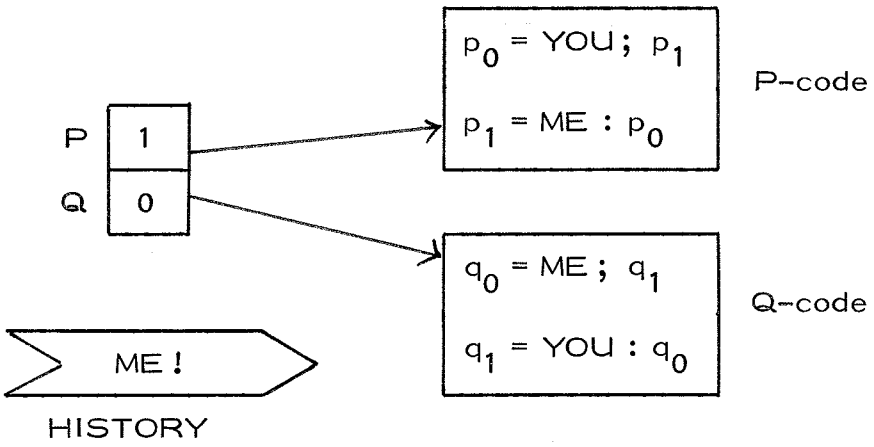
If we start the computer in an initial control state σ we will reach a jammed control state σ, because files in real computers are finite. Some jammed control states are better than others; some of them are acceptable control states : jammed because the HISTORY file is empty; some of them are final control states : not only acceptable but the integers in P and Q satisfy the "termination criteria". If σ is a final control state, then the port sequence given by removing decorations in σ's history should be in the path set solution of the network equations. This path set solution should also include an infinite port sequence

$$(\pi_1 \ \pi_2 \ \pi_3 \ \ldots\ldots)$$

if taking (π_1) then (π_2) then (π_3) ... as the values of the file HISTORY gives only acceptable control states. We shall say no more about infinite path sequences, because they are controversial (Park ...) and they play no part in the semantics we shall give for parallel programs. Instead we describe another computer realization - we replace network equations by task equations. At the beginning of this section we gave five rules that defined the capabilities on the right side of the equation for (p|q) in terms of the capabilities in the task equations for p and q. Instead of matching the first component of HISTORY with the right side of the network equation for (p|q), the computer now matches the first components with the right side of the task equations for p and q. The new computer realization is equivalent to the old in that

1) it jams, when the old realization jammed
2) when it does not jam, it gives the same next control state as the old realization.

The new computer realization is a natural generalization of the usual computer realization of serial programs, task equations correspond to compiled code, and control cells correspond to instruction pointers.



Another close relative of network equations should be mentioned : finite automata. Process variables correspond to automaton states, parts correspond to input symbols, and capabilities correspond to transitions. The transition table for the automaton that corresponds to our producer-consumer example is :

Input Symbol / State	arrive!	depart 1!	depart 2!	Print Message;
$(p_0 \mid r_0 \mid t_0)$	$(p_1 \mid r_0 \mid t_0)$			
$(p_0 \mid r_0 \mid t_\infty)$	$(p_1 \mid r_0 \mid t_\infty)$			(
$(p_0 \mid r_1 \mid t_0)$	$(p_1 \mid r_1 \mid t_0)$			$(p_0 \mid r_0 \mid t_0)$
$(p_0 \mid r_1 \mid t_\infty)$	$(p_1 \mid r_1 \mid t_\infty)$			$(p_0 \mid r_0 \mid t_\infty)$
$(p_1 \mid r_0 \mid t_0)$		$(p_2 \mid r_1 \mid t_0)$		
$(p_1 \mid r_0 \mid t_\infty)$		$(p_2 \mid r_1 \mid t_\infty)$		
$(p_1 \mid r_1 \mid t_0)$				$(p_1 \mid r_0 \mid t_0)$
$(p_1 \mid r_1 \mid t_\infty)$				$(p_1 \mid r_0 \mid t_1)$
$(p_2 \mid r_0 \mid t_0)$	$(p_3 \mid r_0 \mid t_0)$			
$(p_2 \mid r_0 \mid t_\infty)$	$(p_3 \mid r_0 \mid t_\infty)$			
$(p_2 \mid r_1 \mid t_0)$	$(p_3 \mid r_1 \mid t_0)$			$(p_2 \mid r_0 \mid t_0)$
$(p_2 \mid r_1 \mid t_\infty)$	$(p_3 \mid r_1 \mid t_\infty)$			$(p_2 \mid r_0 \mid t_\infty)$
$(p_3 \mid r_0 \mid t_0)$			$(p_0 \mid r_0 \mid t_0)$	
$(p_3 \mid r_0 \mid t_\infty)$				
$(p_3 \mid r_1 \mid t_0)$			$(p_0 \mid r_1 \mid t_\infty)$	$(p_3 \mid r_0 \mid t_0)$
$(p_3 \mid r_1 \mid t_\infty)$				$(p_3 \mid r_0 \mid t_\infty)$

It would be very satisfying if the theory of decomposition of finite automata into cascades and the like could be applied fruitfully to the practical design of parallel programs.

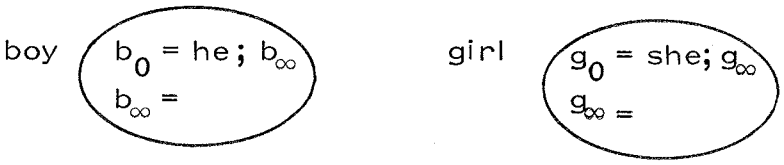
So far we have said nothing about dynamic networks, about the birth and death of tasks. Now we suppose that tasks have parents that can be called by tasks in the network. (Birth) When a task calls a parent, the

network equations are changed to allow for the tasks that are children of the parent (Death). Networks never shrink, but some of the tasks in the network may have no capabilities left.

Remember our equation set for the task First-Consumer

$$r_0 = \text{depart } 1 : r_1$$
$$r_1 = \text{Print Message} ; r_0$$

If Print Message is the parent of two tasks



then the equation set for First-Consumer changes to the infinite set

$$r_0 = \text{depart } 1 : r_1$$
$$r_1 = \text{Message}; (r_0 | b_\infty | g_\infty)$$
$$(r_0 | b_\infty | g_\infty) = \text{depart } 1 : (r_1 | b_\infty | g_\infty)$$
$$(r_1 | b_\infty | g_\infty) = \text{Message} ; (r_0 | b_\infty | g_\infty | b_\infty | g_\infty)$$
$$(r_0 | b_\infty | g_\infty | b_\infty | g_\infty) = \text{depart } 1 ; (r_1 | b_\infty | g_\infty | b_\infty | g_\infty)$$
$$---$$

There would be analogous changes in the network equations for the whole producer-consumer example. (For ADA experts : if the body of Print Message contained initiate boy, girl as well as the call of the procedure Message, these equations would be

$$r_0 = \text{depart } 1 : r_1$$
$$r_1 = \text{Message} ; (r_0 | b_0 | g_0)$$
$$(r_0 | b_\infty | g_\infty) = \text{depart } 1 ; (r_1 | b_\infty | g_\infty)$$
$$(r_1 | b_\infty | g_\infty) = \text{Message} ; (r_0 | b_\infty | g_\infty | b_0 | g_0)$$
$$(r_0 | b_\infty | g_\infty | b_\infty | g_\infty) = \text{depart } 1 : (r_1 | b_\infty | g_\infty | b_0 | g_0)$$
$$---$$

and we would have new equations like

$(r_0|b_0|g_0)$ = depart 1 : $(r_1|b_0|g_0)$ + $(r_0|b_\infty|g_0)$ + she ; $(r_0|b_0|g_\infty)$

$(r_0|b_\infty|g_0)$ = depart 1 : $(r_1|b_\infty|g_0)$ + she ; $(r_0|b_\infty|g_\infty)$

$(r_0|b_0|g_\infty)$ = depart 1 : $(r_1|b_0|g_\infty)$ + he ; $(r_0|b_\infty|g_\infty)$

$(r_0|b_0|g_0|b_0|g_0)$ = depart 1 ; $(r_1|b_0|g_0|b_0|g_0)$

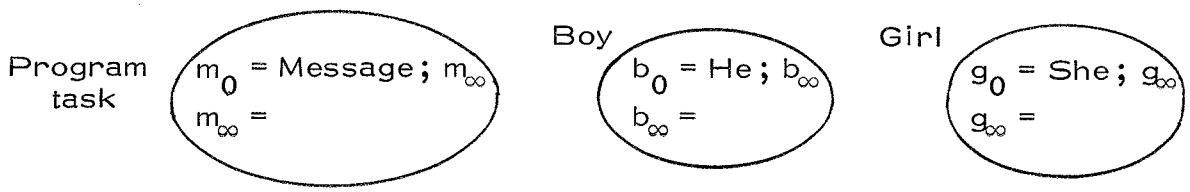
+ he ; $(r_0|b_\infty|g_0|b_0|g_0)$ + she $(r_0|b_0|g_\infty|b_0|g_0)$

+ he ; $(r_0|b_0|g_0|b_\infty|g_0)$ + she $(r_0|b_0|g_0|b_0|g_\infty)$

There would be analogous changes in the network equations for the whole producer-consumer example).

In ADA parents are procedures and the main program is the parent of the program task. If the main program has no other children, and its procedures have no children, then the main program is sequential, the network equations are the program task equations, and we have no rendezvous ports. The program task, like every other task, has a start process; the meaning of a sequential program is the meaning given to this start process when we solve the program task equations.

Let us consider the situation when the main program has more than one child. Suppose Print-Message is the main program, and the initial picture is



The network equations are

$(m_0|b_0|g_0)$ = Message ; $(m_\infty|b_0|g_0)$ + He ; $(m_0|b_\infty|g_0)$ + She ; $(m_0|b_0|g_\infty)$

$(m_0|b_0|g_\infty)$ = Message ; $(m_\infty|b_0|g_\infty)$ + He ; $(m_0|b_\infty|g_\infty)$

$(m_0|b_\infty|g_0)$ = Message ; $(m_\infty|b_\infty|g_0)$ + She ; $(m_0|b_\infty|g_\infty)$

$(m_0|b_\infty|g_\infty)$ = Message ; $(m_\infty|b_\infty|g_\infty)$

$(m_\infty|b_0|g_0)$ = He ; $(m_\infty|b_\infty|g_0)$ + She ; $(m_\infty|b_0|g_\infty)$

$(m_\infty | b_0 | g_\infty)$ = $\qquad$ He; $(m_\infty | b_\infty | g_\infty)$

$(m_\infty | b_\infty | g_0)$ = $\qquad$ She;$(m_\infty | b_\infty | g_\infty$

$(m_\infty | b_\infty | g_\infty)$ =

The meaning of our program is the meaning given to the process variable (startprocess, b_∞, g_∞) when we solve the network equations. The path set solution gives the meaning

"port sequence is (Message;)"

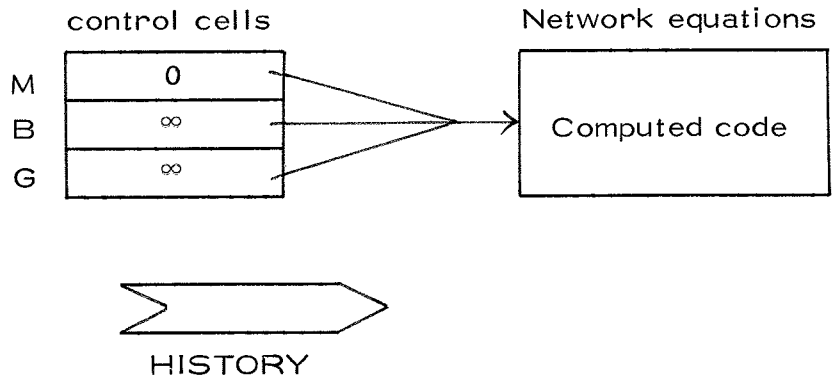
when we take m_0 as the start process (For ADA experts : if we replaced m_0 = Message; m_∞ by m_0 = initiate Boy, Girl; m_0 and added m_1 = m_1 = Message; m_∞ the fourth equation would be

$(m_0 | b_\infty | g_\infty)$ = initiate Boy, Girl; $(m_1 | b_0 | g_0)$

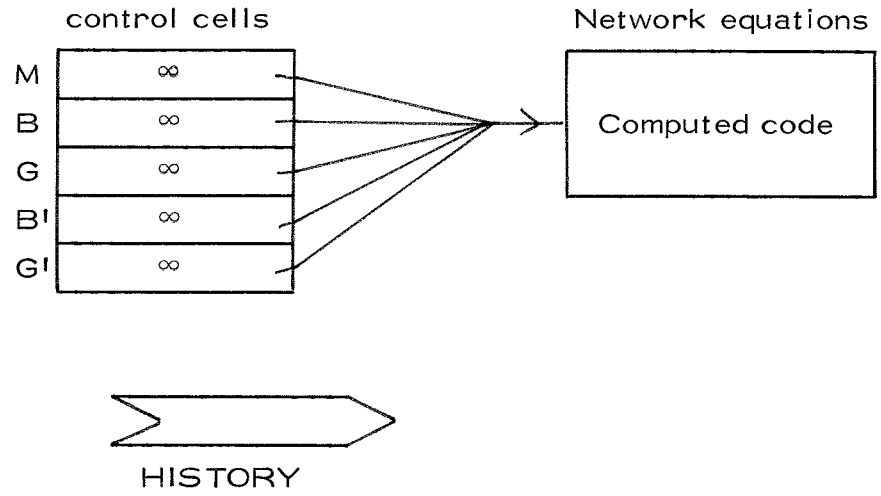
and the path set meaning of the program would be

"port sequence consists of initiate Boy, Girl;
followed by 'Message;', 'He;', 'She;' in any order").

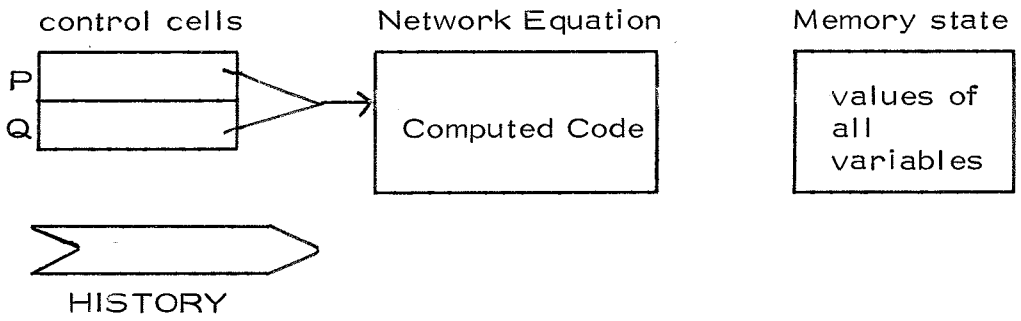
Remember our computer realization. The initial control state for the execution of Print Message would be



and the next control state has ∞ in the three control cells. If we replaced m_0 = Message; m_∞ by m_0 = Print message; m_∞ , the fourth network equation would become $(m_0 | b_\infty | g_\infty)$ = Print Message; $(m_\infty | b_\infty | g_\infty | b_\infty | g_\infty)$ and the next control state would have been



Whenever a computer executes a program from a well defined initial state, it will either terminate in some final state or run for ever (in practice overrun a time limit). Our problem is to give a meaning to a parallel program that pays due attention to the possibility of different executions of the same program from the same initial state giving different final states and/or running for ever. Remember the way we could realize parallel programs in the computer using control cells and a HISTORY file. If we suppose the current values of all the variables (the memory state) are part of the control state, then we can define



the next control state by

1) the first part in HISTORY gives a capability $\gamma(p_k | q_l)$ in the network equation given by the control cells

2) the integers k and l are put in the appropriate control cells

3) the memory state is changed in accordance with γ

4) the first port in HISTORY is deleted.

Because files in a real computer are finite, we will always reach a jammed control state when we start the computer in an initial control state. The semantics of the programming language may give us a function from memory states to memory states for each port that occurs in HISTORY. If it does, our realization gives the function g from initial control states to final control states defined by :

g (old-control-cells, old-HISTORY, old-memory state)

= if old-HISTORY is empty or no capability for first port in old-HISTORY
then (old-control-cells, old-HISTORY, old-memory-state)
else let old-HISTORY = (γ , new-HISTORY)
and new-control-cells be given by capability for γ
and f be the function for γ
in if f is defined for old-memory-state
then (new-control-cells, new-HISTORY, f(old-memory-state))
else (old-control-cells, old-HISTORY, old-memory-state)

We can use the function g to define a predicate-transformer semantics when we have a predicate FINAL on control states :

INITIAL(old-control-cells, old-HISTORY, old-memory-state)
≡ FINAL (g(old-control-cells, old-HISTORY, old-memory-states))

We get a path set solution of our network equation for every choice of old-memory-state by :

If INITIAL(old-control-cells, old-HISTORY, old-memory-state) holds then the port sequence, given by removing decorations in old-HISTORY, is put in the path set for the process "old-control-cells".

The path sets in these solutions are subsets of those in the path set solution we described earlier : there are no infinite port sequences, and some finite port sequence may be absent. These solutions agree when no finite port sequence is absent; in particular they agree when the functions assigned to ports are defined for all memory states and the

truth value of

FINAL(old-control-cells, old-HISTORY, old-memory-state)

does not depend non old-memory-state.

Straightforward development of the ideas in the last paragraph would give a "resumption" semantics for ADA similar to those given for other languages by Plotkin (4). We prefer the flexibility of continuations; we replace a function f from memory states to memory states by the function

$\hat{f}$ (old – local-continuation) = old-local-continuation of;

and we replace a function g from control states to control states by the function

$\hat{g}$(old-global-continuation) = old-global-continuation o g.

These replacements are only meaningful if we define

Local-Continuation = Memory state → Answer
Global-Continuation = Control state → Answer

so the definitions of $\hat{f}$ and $\hat{g}$ can be written

$\hat{f}$(old-local-continuation) old-memory state
= let new-memory-state = f(old-memory-state)
in old-local-continuation (new-memory-state)

$\hat{g}$(old-global-continuation) old-control-state
= let new-control-state = g(old-control-state)
in old-global-continuation (new-control-state).

We assume that all functions are total and all domains have an "undefined" element. We replace the function g from control states to control states by :

$\hat{g}$: Global-Continuations → Global-Continuations

$\hat{g}$ (old-global-continuation) (old-control-cells, old-HISTORY, old-memory-state)
= if old-HISTORY is empty
or no capability for first port in old-HISTORY
then old-global-continuation (old-control-cells, old-HISTORY, old-memory-state)
else let old-HISTORY = (γ, new-HISTORY)
and new-control-cells be given by the capability for γ
and $\hat{f}$ be the function for γ
and old-local-continuation (new-memory-state)
= old-global-continuation (new-control-cells,
new-HISTORY, new-memory-state)
in $\hat{f}$ (old-local-continuation) (old-memory-state).

We get the replaced function g as the least fixed point of $\hat{g}$ containing the identity function if we have

Answer = Control state;

we get the predicate INITIAL as the least fixed point of $\hat{g}$ containing the predicate FINAL if we have

Answer = {true, false}

(assuming History is a "flat domain" and false is the undefined answer element). Our continuation approach is so flexible that it can give good meaning to non-terminating programs; if we have

Answer = Trace*

and a function OUTPUT from memory sets to traces we can replace the last line in our definition of $\hat{g}$ by

in (OUTPUT(old-memory-state), $\hat{f}$(old-local-continuation)
(old-memory-state)

Continuations are used in the draft formal semantics for ADA ( ) because they also give a convenient way of describing exceptions and other language constructs that affect the run-time behaviour of a program.

Now we give a semantics for parallel ADA programs on the assumption that a set of network equations is equivalent to a parallel program, so our function g gives the meaning of the parallel program. Part 3 of this paper is devoted to the justification of this assumption, here we discuss the ADA form of the HISTORY file. The components of HISTORY can give

1) the name of an entry or a procedure;
2) the name of a task calling this entry or procedure;
3) the code to be executed.

The denotational semantics version of this is

HISTORY = PORT* (flat domain)
PORT = Name × Task Name × Code

The semantics of a procedure port (procedure name, T, S) is the function f from local continuations to local continuations, given S by the draft ADA semantics when the variables refer to the T-part of the memory state. In our definition of the function $\hat{g}$ from global continuations to global continuations we had the condition "no capability for first port in old-HISTORY". When this first port is (procedure-name, T, S), this condition is fulfilled when the control cell for T does not indicate a call of a procedure whose body is S.

To deal with such ADA features as - initiate, guarded select statements, and the possibility of escaping from a select statement without making a rendezvous - we introduce special procedure ports. We can think of "initiate T_1, T_2;" as the name of a procedure. The tasks T_1 and T_2 exist when this procedure is called because their parents are in the scope of the calling task. The code to be attached to a port that begins with "initialize T_1, T_2" should be such that the semantics of the port is :

$\hat{f}$: Local Continuation → Local Continuation

$\hat{f}(\theta)(s)$ = if T_1 and T_2 are dead in old-memory-state
then θ (s with T_1 and T_2 alive)
else undefined-answer

As this paper is already long enough, we do not discuss the semantics of exception handling in ADA; we have used undefined-answer, where we should have raised an exception. We have also used the memory state to remember whether a task is alive or dead. To maintain this information we suppose that we have procedure ports (kill; , T, S_T) where S_T is such that we have the semantics

$\hat{f}(\theta)(s)$ = if T is alive but all its children are dead in s
then θ (s with T dead)
else undefined-answer

This semantics reflects the ADA rules about children of terminating tasks (first sentence on page 11-4 of SIGPLAN Notices June 1979, part B).

Now consider the case then the first port in HISTORY is an entry port : (entry-name, T, S). the $\hat{g}$ condition "no capability for first port in old-HISTORY" is fulfilled when

either the control cell for T does not indicate the call of entry name
or the control cell for the owner of entry-name does not indicate the possibility of a rendezvous on accept entry-name do S end

The semantics of our entry port is the function $\hat{f}$ from local continuations to local continuations, given S by the draft ADA semantics for the part of memory state for the owner of entry name. When we have a select statement with N guards there will be 2^N network equations (2^N possible values in the control cell), one for each assignment of truth values to the guards. For the case when all guards evaluete to FALSE, the network equation will allow us to escape from the guarded select statement if the statement has an else port. We can escape from an unguarded select statement if we have procedure ports (delay; , T, S) with the semantics, given S by the draft ADA semantics when the variables refer to the T-port of the memory state. The $\hat{g}$ condition "no capability for first port in old-HISTORY" is fulfilled when the first port is (delay; T, S) but the control cell for T does not indicate the possiblity of delay(...) do S end.

A brief look at our producer-consumer example will illustrate our semantics and suggest a way of treating ports with parameters. Suppose we have

```
task body   Post Box is
    begin
        loop
            accept arrive do count : = count + 1 end ;
            depart 1 ;
            accept arrive do count : = count + 1 end ;
            depart 2 ;
        end loop ;
    end Post Box ;

restrict (TEXT_IO)
task body First Consumer is
    begin
        loop
            accept depart 1 ;
            TEXT_IO. PUT(count) ; -- Print Message
        end loop
    end First Consumer;
```

If we assume count is the only global variable, we can make the definition

Memory State = Integer Answer = Integer*

so

Local Continuation = Integer → Integer*.

The function from local continuations to local continuations for "count : = count + 1"

$$\hat{f}(\theta)(count) = \theta\ (count + 1)$$

would be the semantics of the port (arrive!, T, count : = count + 1). The corresponding function for TEXT_IO. PUT(count) :

$$\hat{f}(\theta)(count) = (count, \theta(count))$$

would be the semantics of the procedure port

(Print Message; First Consumer, TEXT_IO.PUT(count)).

What should change if we had had

```
task body Post Box is
   begin
      loop
         accept arrive (in out formal : INTEGER)
            do count := count + formal ; formal := count end
         depart 1 ;
         accept arrive (in out formal : INTEGER)
            do count := count + formal ; formal := count end
         depart 2 ;
      end loop ;
end Post Box ;
```

We should redefine Memory State = Integer^3

Local Continuation = $\text{Integer}^3 \to \text{Integer}^*$

and we should introduce the entry name "arrive (actual $\Leftrightarrow$ formal) ! ".
The semantics of the port

(arrive (actual $\Leftrightarrow$ formal), T, count := count + formal ; formal := count)

should be

$$
\begin{aligned}
&\hat{f}(\theta)(\text{count, formal, actual}) \\
&= \underline{\text{let}}\ s = (\text{count, actual, actual}) \\
&\quad \underline{\text{and}}\ \hat{h}(\theta'')(c,f,a) = \theta''(c+f_1 c+f_1 a) \\
&\quad \underline{\text{and}}\ \theta'(c,f,a) = \theta(c,f,f) \\
&\quad \underline{\text{in}}\ \ \hat{h}(\theta')(s)
\end{aligned}
$$

We set the in and inout parameters, we obey the code between do and end, we copy the out and inout parameters. If we define the semantics of a port with parameters by :

$\hat{f}$(old-local-continuation)(old-memory-state)

= let new memory-state be the result of setting
in and inout parameters in old-memory-state
and $\hat{h}$ be the semantics of the port without parameters
and new local-continuation (some-memory-state)
= old-local-continuation (result of copying out and inout)
parameters from some-memory-state)
in $\hat{h}$(new-local-continuation)(new-memory-state)

then our semantics of parallelism in ADA is complete.

Part 3 : JUSTIFICATION

How can we justify our assumption that solving network equations gives the semantics of parallelism in ADA ? Let us start with the ADA requirements :

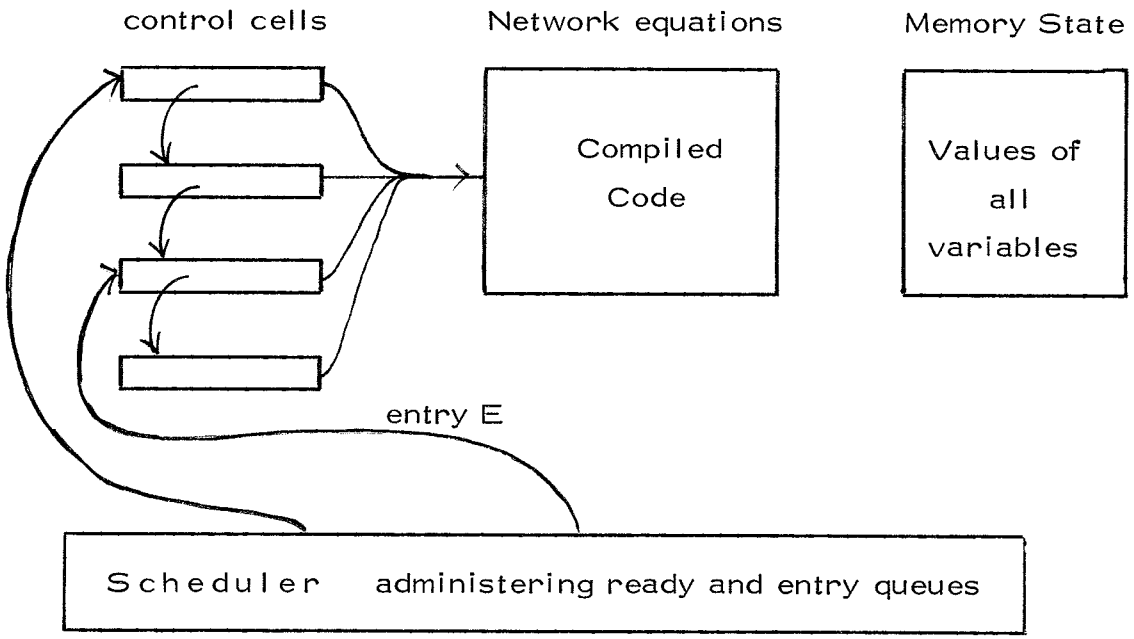
"If several tasks call the same entry before a corresponding accept is reached, the calls are queued; there is only one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in order of arrival. Each task can only be on one queue". (Preliminary ADA reference manual 9.5).

"There may be several tasks that are ready to be executed by the system processors. In choosing the processes to be executed, processes with the highest priority are treated on a first-in-first-out basis. The language does not specify when a scheduling decision is made" (Preliminary ADA reference manual, section 9.8)

These are captured by placing a restriction on initial control states : control cells have a queue structure, this structure may change in the course of the computation from an initial control state σ, but the sequence of ports in the HISTORY port of σ must respect the control cell structure.

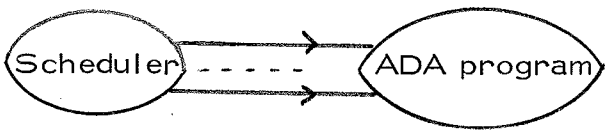
More formally : this requirement is incorporated in the condition : "no capability for first port in old-HISTORY" that occurs in our definitions of the function g from initial control states to final control states and the function $\hat{g}$ from global continuations to global continuations.

More practically : this requirement restricts the choice of next control state in an implementation where HISTORY is replaced by a scheduler :



The ADA requirements rule out certain port sequences as values of HISTORY, some other port sequences may not be relevant for particular scheduling algorithms, still more port sequences may be ruled out by the choice of processors because of the delay statements in ADA. The network equations give the meaning of a parallel program for a particular port sequence, but the question of whether or not this port sequence can actually occur depends on the choice of processors and scheduling algorithm.

In this paper we have used the condition "no capability for first port in old-HISTORY" to decide whether or not a port sequence can occur. Another approach is shown in the diagram



with an arrow for every port in the network equations for the ADA program. Each network equation gives a Milner symbol

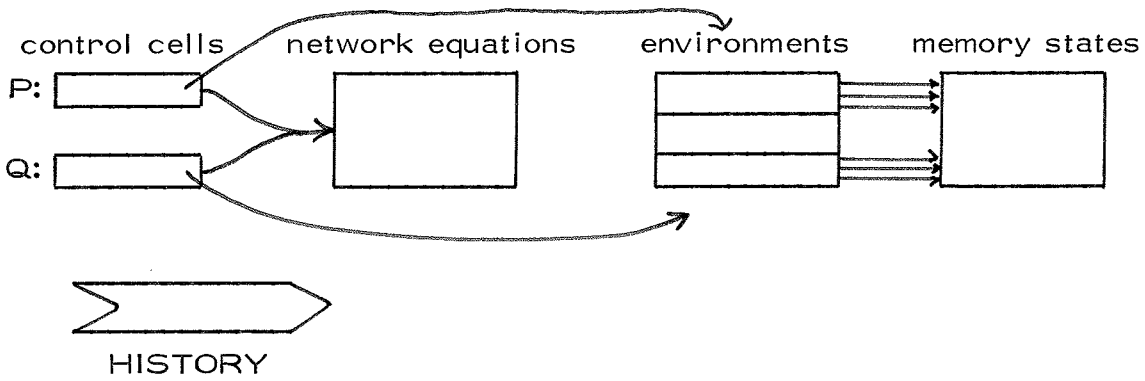
with dotted half arrows for ports that are not on the right side of the network equation. The semantics in part 2 of this paper assumes that Scheduler has just one Milner symbol (no dotted half arrows); the ADA

requirements at the beginning of this part, and the choice of processors and scheduling algorithms force Scheduler to have more than one Milner Symbol. We can even replace network equations for an ADA program by equations for the individual tasks, if we can accept a complicated Scheduler. The resulting semantics would be very close to a possible implementation of ADA.

Our approach to scheduling in ADA is similar to our approach to scope and visibility in ADA. Variables and other named quantities have a procedure or module as parent. Because ADA permits recursion, parents may exist in many incarnations. If there is an incarnation of the parent in an executing task, then a name refers to the latest incarnation of its parent. We can use the usual display mechanism to divide our memory state into sections for each parent incarnation. More formally : each executing task has an ENVIRONMENT

$$\text{Memory State} = \text{Locations} \to \text{Values}$$

and part of the environment gives a map from variables to locations (we can also handle exceptions if another part of the environment gives a map from exceptions to global continuations).



In part 2 we have assumed that an environment determines a function $\hat{h}$ from local continuations to local continuations for each code S. This assumption is true for the draft formal semantics for ADA

There is a more fundamental doubt about the validity of our assumption that solving network equations can give the semantics of parallelism in ADA; how can non-deterministic equations reflect the possibility of

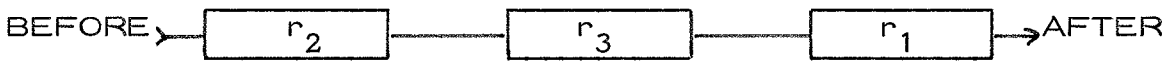
true parallelism; when an executing ADA program can use more than one processor ? We begin our argument for the validity of this assumption by considering the network equations

$$r_1 = [S1]\,;\, r_2$$
$$r_2 = [\underline{not}\ B]\,;\, r_3 + [B]\,;\, r_4$$
$$r_3 = [S2]\,;\, r_1$$
$$r_4 =$$

for the ADA program fragment

```
loop
    S1 ;
    exit when B ;
    S2 ;
end loop ;
```

The process variables r_1 r_2 r_3 r_4 represent periods (closed intervals of time) when the fragment is "resting at a semicolon". During execution intervals between the rest periods, the process variables and network equations do not apply. Any execution of the fragment gives a time chart



with lines for execution intervals and boxes for periods. If our program fragment is part of an executing task, we still have such a time chart even although

1) when the task is in a box (at points in the time chart box) there need not be a processor assigned to the task ;
2) different processors can be assigned to the task for different execution intervals in the time chart.

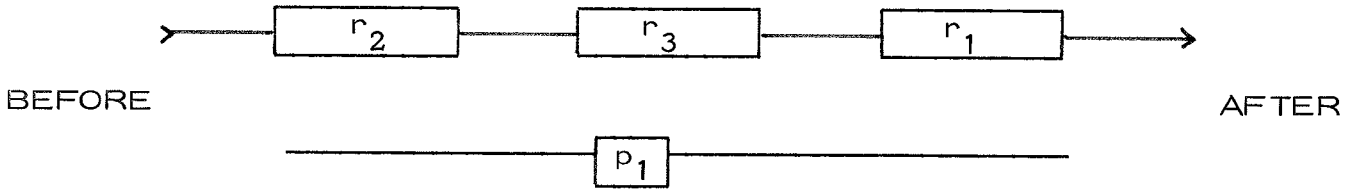
Now suppose our task is running in parallel with another task whose equations are

$$p_1 = [S3] ; p_1.$$

The network equations become

$$(r_1|p_1) = [S3] ; (r_1|p_1) + [S1] ; (r_2|p_1)$$
$$(r_2|p_1) = [S3] ; (r_2|p_1) + [\underline{\text{not}}\ B] ; (r_3|p_1) + [B] ; (r_4|p_1)$$
$$(r_3|p_1) = [S3] ; (r_3|p_1) + [S2] ; (r_1|p_1)$$
$$(r_4|p_1) = [S3] ; (r_4|p_1)$$

and execution might give a <u>malign</u> time chart

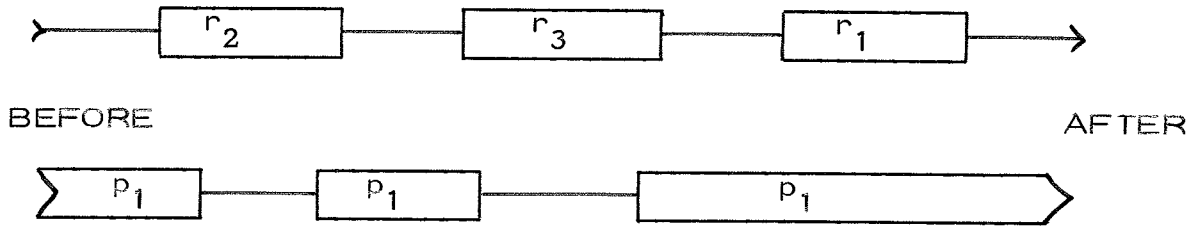


in which the two tasks are never in a box at the same time. It does not seem unreasonable to assume that

> a task can only acquire or relinguish a processor when it is not executing.

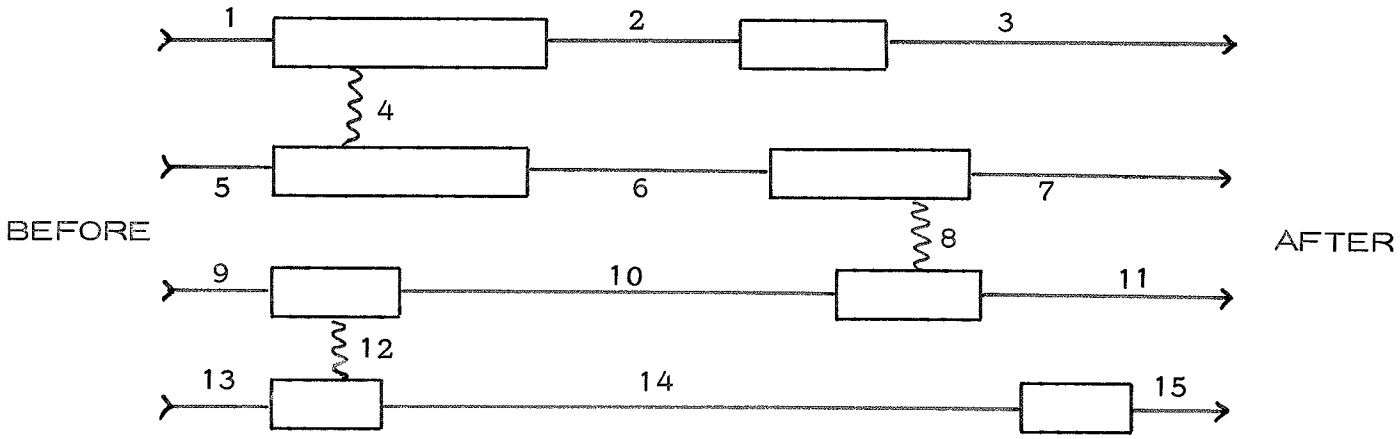
> two processors can only exchange information when neither is assigned to the execution interval of a task.

so our two tasks are running on two processors that never exchange information. Because of this the speed of the processors has no effect on the results of an execution,and our execution with a malign time chart gives the same result as an execution with a <u>benign</u> time chart



in which the two tasks are never outside a box at the same time. The process variables in the network equations are meaningful for benign time charts. The assumption we are trying to justify asserts that every malign time chart is equivalent to some benign time chart. This is obviously true when we have only one processor.

Now consider the case of many tasks on many processors. An execution may well have a malign time chart

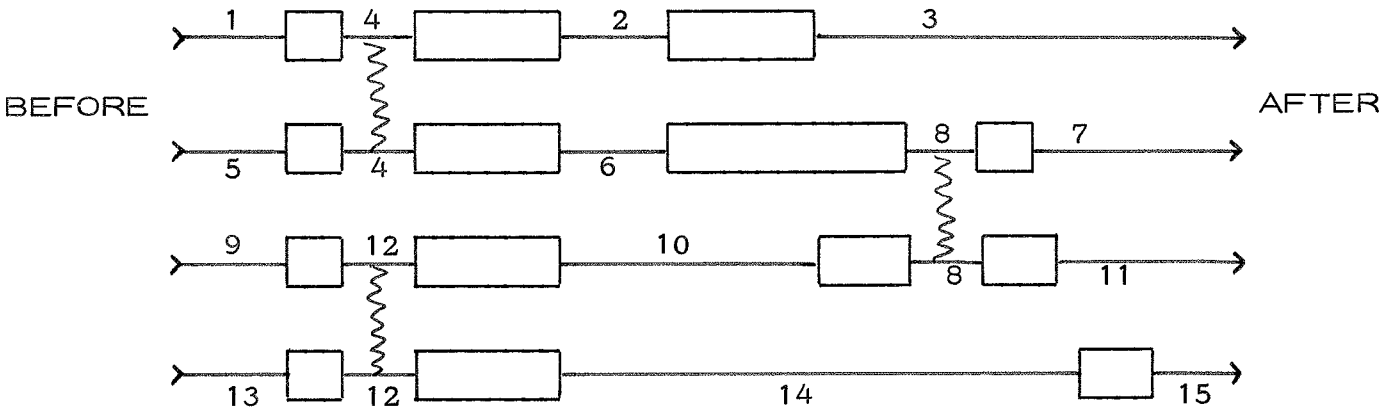


with vertical zig-zags representing rendezvous and more than one rendezvous at the same time.

In ADA simultaneous rendezvous cannot share a task, because tasks call and accept entries sequentially

two tasks can only make a rendezvous when neither is in an execution interval

so zig-zags in a time chart join boxes. Now suppose that rendezvous are a special kind of execution interval so that zig-zags in a time chart give lines with the same start and finish instants



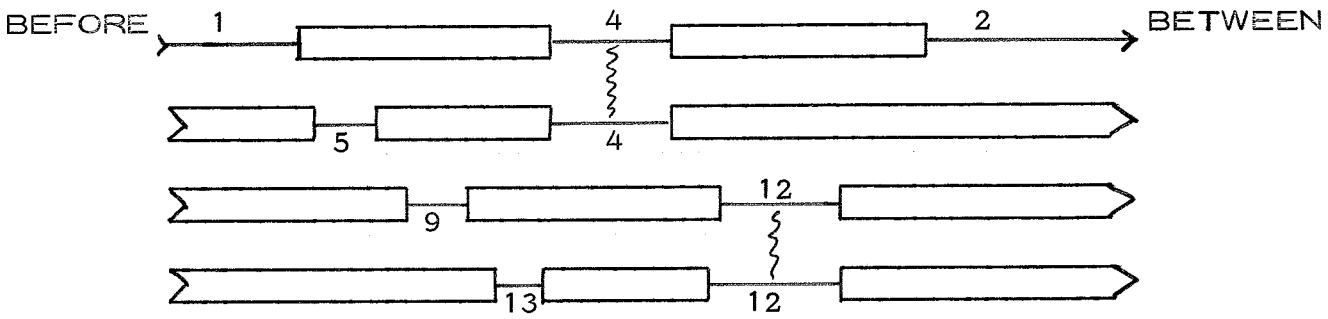
The time charts for the individual tasks always give a partial ordering of execution intervals

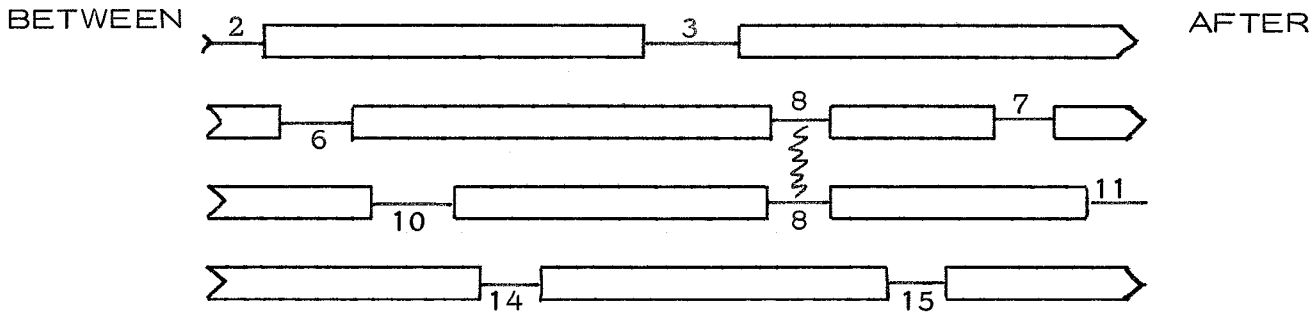
$$1 < 4 < 2 < 3$$
$$5 < 4 < 6 < 8 < 7$$
$$9 < 12 < 10 < 8 < 11$$
$$13 < 12 < 14 < 15$$

which we can embed (by topological sorting) in a linear order

$$1 < 5 < 9 < 13 < 4 < 12 < 2 < 6 < 10 < 14 < 3 < 8 < 15 < 7 < 11.$$

We now have a benign time chart





If you agree that

- a task only acquires or relinguishes a processor when it is not executing
- processors only exchange information when they are not executing some task

then you should agree that a malign time chart is equivalent to the benign time chart given by linearization, the corresponding program executions give the same result. Our assumption that the semantics of "true parallelism" in ADA can be captured by solving "non-deterministic" network equations is thereby justified.

REFERENCES

(1) Preliminary ADA Reference Manual
SIGPLAN Notices 14 (1979), nr. 6, part A

(2) Rationale for the design of the ADA programming language
SIGPLAN Notices 14 (1979), nr. 6, part B

(3) R. Milner : "An Algebraic Theory for Synchronization"
GI 4 Conference, Springer Lecture Notes 67.

(4) G. Plotkin : Lectures at Copenhagen 1979 Winter School.