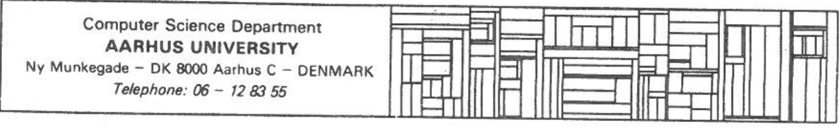
ISSN 0105-8517

A Computational Model for ADA and other concurrent languages

Brian H. Mayoh

DAIMI PB-162
May 1983

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



PB-162 B.H. Mayoh: A model for concurrent languages

TRYK: DAIMI/RECAU

A COMPUTATIONAL MODEL FOR ADA AND OTHER CONCURRENT LANGUAGES

Brian H. Mayoh

An earlier report [Ma1] presented the argument for diagrams as the most suitable models for programs with concurrency. In a diagram the configurations correspond to points of the program, there is a move from m to n if n can be the next program point after m and the move is labelled by the program action when the computation follows the move. This report shows that this computational model is suitable for ADA and other languages with concurrency. In Section 1 we give the computational model for the guarded command language GCL; in Section 2, a powerful specification language SCSS is modelled; in Section 3 we give the model for the language CSP of communicating sequential processes; then the techniques introduced in these sections are used when ADA programs are modelled in Section 4.

CONTENTS

1. COMPUTATIONAL MODEL FOR GCL 1

2. COMPUTATIONAL MODEL FOR SCCS 6

3. COMPUTATIONAL MODEL FOR CSP 10

4. COMPUTATIONAL MODEL FOR ADA 19

SUMMARY 30

REFERENCES 31

1. COMPUTATIONAL MODEL FOR GCL

The guarded command language GCL [Di] is a very simple language with non-determinism. An example of a program is

$$\underline{\text{if}}\ x > y \rightarrow max{:=}x \ \square\ x \leq y \rightarrow max{:=}y\ \underline{\text{fi}}$$

Informally we can describe the behaviour of this program by:

- Suppose we have a pebble placed before _if_
- the pebble is moved after _if_ and the truth values of the guards "$x \geq y$" and "$x \leq y$" are determined
- if either of the guards is true, the pebble may be moved to the corresponding arrow
- after the pebble has been moved to an arrow, the corresponding assignment may be made and the pebble can be moved beyond _fi_.

Sometimes the pebble is at points in the program; sometimes the pebble is in the air while it is passing a program test or command. The pebble movements for the program correspond to traversals of paths in the graph

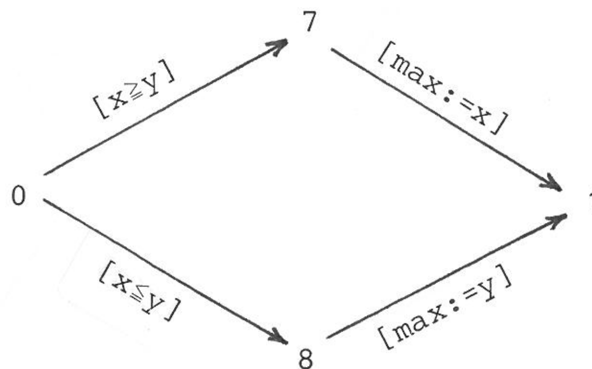


Figure 1
Program Graph

Graphs are not always convenient models of programs, but one can generalise them to diagrams by allowing sums and products of nodes.

Definition

A diagram consists of

- C, a set of configurations, labelled by integers;
- a partition of C into source, sink and internal configurations;
- M, a set of moves labelled by actions.

Each move in a diagram has a source and a sink which are configuration terms. The set T of configuration terms in a diagram (C,M) is given by: $C \subset T$, if $t_1, t_2 \in T$ then $(t_1+t_2), (t_1 \times t_2) \in T$.

Comment

Products are used to express concurrency: sums are used to express program choice.

Example

The graph for our GCL program corresponds to the diagram

$$C = (0,1,7,8)$$

$$M = (0 \xrightarrow{x \geq y, x \leq y} (7+8),\ 7 \xrightarrow{max:=x} 1,\ 8 \xrightarrow{max:=y} 1)$$

Because configurations have integer labels, one can define a neat substitution operator on diagrams:

- one can write "$m_1 + \ldots + m_k \xrightarrow{D} n_1 + \ldots + n_l$" in the specification of a diagram D' when D is a diagram with k source configurations and l sink configurations;
- the source configurations in increasing order are assigned to $m_1 \ldots m_k$ respectively;

- the sink configurations in increasing order are assigned to $n_1 \ldots n_l$ respectively;
- the moves in D are added to D' after the internal configurations are renamed to avoid conflicts in D'.

The substitution operator will be used often in this paper, but its use should be clear if the reader remembers the following translation of configuration labels

source	sinks						internals
0	1	2	3	4	5	6	7,8...,j
normal begin	normal end	aborted	otherwise	select error	failure	tasking error	

Figure 2
Uniform configuration naming convention

We could have modelled our program by a Petri net, and we have borrowed the pebble idea from the net theorists because it will help when we come to describe the diagram model for ADA programs.

The syntax of GCL can be given by the productions

<command>::=<variable>:=<expression>|skip|abort|
<command>; <command>|
if <guarded commands> fi
do <guarded commands> od
<guarded commands>::=<test>→<command>|
<guarded commands>□<guarded commands>

where <variable>, <expression> and <test> are left unspecified. The meaning of each GCL command is given in figure 3.

Command	Configurations	Moves
<variable>:=<expression>	0,1,2	$0 \xrightarrow{\text{[variable:=expression]}} 1$
<u>skip</u>	0,1,2	$0 \xrightarrow{\text{[skip]}} 1$
<u>abort</u>	0,1,2	$0 \xrightarrow{\text{[skip]}} 2$
$\langle command_1\rangle;\langle command_2\rangle$	0,1,2,7	$0 \xrightarrow{command_1} 7+2$
		$7 \xrightarrow{command_2} 1+2$
<u>if</u> <guarded command> <u>fi</u>	0,1,2	$0 \xrightarrow[\text{command}]{\text{guarded}} 1+2+2$
<u>do</u> <guarded command> <u>od</u>	0,1,2	$0 \xrightarrow[\text{command}]{\text{guarded}} 0+2+1$

Figure 3

Semantics of GCL commands

In order to get a diagram meaning for each GCL command we must have diagrams for guarded commands. Clearly a guarded command must be of the form

$$B7 \rightarrow C7 \;\square\; B8 \rightarrow C8 \;\square\; \ldots \;\square\; B_j \rightarrow C_j \qquad \text{where } j \geq 7.$$

The corresponding diagram has the configurations 0,1,2,3,7,8,...j

and the moves $7 \xrightarrow{C7} 1+2$, $8 \xrightarrow{C8} 1+2 \ldots j \xrightarrow{Cj} 1+2$

$$0 \xrightarrow{?\ B7,B8\ldots Bj} 3+7+8+\ldots+j$$

Example

The diagram for "$x \geq y \rightarrow max:=x \;\square\; x \leq y \rightarrow max:=y$" has the configurations 0,1,2,3,7,8 and the moves

$$7 \xrightarrow{[max:=x]} 1, \quad 8 \xrightarrow{[max:=y]} 1$$

$$0 \xrightarrow{?\ x \geqq y,\ x \leqq y} 2+7+8$$

The diagram corresponds to the graph in figure 1, when the alternative 2 is deleted from the last move because one of $x \geqq y$ and $x \leqq y$ is always true.

□

2. COMPUTATIONAL MODEL FOR SCCS

The calculus of synchronous agents SCCS [Mi] is a specification language which exploits non-determinism fully. The SCCS specification for the program in Section 1 is

$$p_0 \equiv [x \geq y] : p_7 + [x \leq y] : p_8$$

$$p_7 \equiv [max:=x] : NIL$$

$$p_8 \equiv [max:=y] : NIL$$

The syntax of SCCS can be given by the productions

```
<specification>::=<name>≡<agent>|<specification><specification>
        <agent>::=<name>|δ<agent>|Δ<agent>|
                  <action>:<agent>|<action>|<agent>|
                  <agent>+<agent>|<agent>×<agent>|
                  <agent>"|"<agent>|<agent>↾<action class> |
                  <agent>[<action morphism>]
```

where <name>, <action>, <action class> and <action morphism> are described in [Mi].

The meaning of each SCCS agent is given in figure 4.

Agent	Configu-rations	Moves
<name>	0	none
δ<agent>	0	$0 \xrightarrow{\text{skip}} 0,\ 0 \xrightarrow{\text{agent}}$
Δ<agent>	as <agent>	explained later
<action>:<agent>	0,7	$0 \xrightarrow{\text{action}} 7,\ 7 \xrightarrow{\text{agent}}$
<action>.<agent>	0,7	$0 \xrightarrow{\text{skip}} 0,\ 0 \xrightarrow{\text{action}} 7$ $7 \xrightarrow{\text{agent}}$
<agent1>+<agent2>	0	$0 \xrightarrow{\text{agent1}},\ 0 \xrightarrow{\text{agent2}}$
<agent1>×<agent2>	explained later	explained later
<agent1>|<agent2>	explained later	explained later
<agent>↾<action class>	as <agent>	explained later
<agent>[action morphism]	as <agent>	explained later

Figure 4

Semantics of SCCS agents

Notice that agent diagrams have no sink configurations, but some of the configurations may come from the agent whose name is NIL.

Example

The agent "[max:=x] : NIL" has the diagram whose configurations are 0 and 7, the only move is

$$0 \xrightarrow{max:=x} 7$$

and the configuration 7 corresponds to the agent NIL. The agent "[x≧y] : p_7 + [x≦y] : p_8" has the diagram whose configurations are 0, 7 and 8, the moves are

$$0 \xrightarrow{x \geqq y} 7,\ 0 \xrightarrow{x \leqq y} 8$$

and the configurations 7 and 8 are different because the corresponding agents have different names.

□

Let us turn to the explanations promised in figure 4. To get the diagram for Δp, one adds $j \xrightarrow{skip} j$ for every internal configuration j in the diagram for p. To get the diagram for $p \restriction A$ one keeps only those moves in the diagram for p, that have labels in A. To get the diagram for $p[\emptyset]$ one relabels all the moves in the diagram for p - $\emptyset(\alpha)$ replaces label α. Suppose the agent p_1 has the diagram $D_1 = (C_1, M_1)$ and the agent p_2 has the diagram $D_2 = (C_2, M_2)$. The diagrams for $p_1 \times p_2$ and $p_1 | p_2$ have $C_1 \times C_2$ as the set of configurations. For both $p_1 \times p_2$ and $p_1 | p_2$ the set of source configurations is (source configuration in C1)× (source configuration in C2) and we have the move $i1 \times i2 \xrightarrow{\alpha \times \beta} j1 \times j2$ whenever $i1 \xrightarrow{\alpha} j1$ and $i2 \xrightarrow{\beta} j2$. In the diagram for $p_1 \times p_2$ we have only these moves; in the diagram for $p_1 | p_2$ we also have the move $i1 \times i2 \xrightarrow{\alpha \times \beta} j1 \times j2$ whenever β = <u>skip</u> & i2=j2 or α = <u>skip</u> & i1=j1.

At last we can define the diagram for a specification of the form

$$x_1 \equiv c_1,\ x_2 \equiv c_2, \ldots,\ x_k \equiv c_k$$

by the rules

- relabel the source configurations 0 of $c_2 \ldots c_k$ by new integers $x_2 \ldots x_k$
- use $0, x_2 \ldots x_k$ to relabel the configurations in the diagrams $c_1 \ldots c_k$ which come from agents named $x_1, x_2 \ldots x_k$

Example

The specification $p_0 \equiv [x \geqq y] : p_7 + [x \leqq y] : p_8$

$p_7 \equiv [max:=x] : NIL$

$p_8 \equiv [max:=y] : NIL$

gives the diagram c_1 with two moves

$$0 \xrightarrow{[x \geqq y]} 7, \quad 0 \xrightarrow{[x \leqq y]} 8$$

the diagram c_2 with one move $0 \xrightarrow{[max:=x]} 7$, and

the diagram c_3 with one move $0 \xrightarrow{[max:=y]} 8$. If we relabel the start configuration of c_2 and c_3 by 17 and 18, then the diagram for the specification has the configurations 0,7,17,18 and the moves

$$0 \xrightarrow{x \geqq y} 17, \quad 17 \xrightarrow{[max:=x]} 7$$

$$0 \xrightarrow{x \leqq y} 18, \quad 18 \xrightarrow{[max:=y]} 8$$

□

In [Mi] Milner has shown that all the SCCS operators are useful and satisfy interesting laws. Note that any finite diagram can be given by an SCCS specification using only the operators ":" and "+". In the next two sections other SCCS operators will help in the building of large diagrams from small.

3. COMPUTATIONAL MODEL FOR CSP

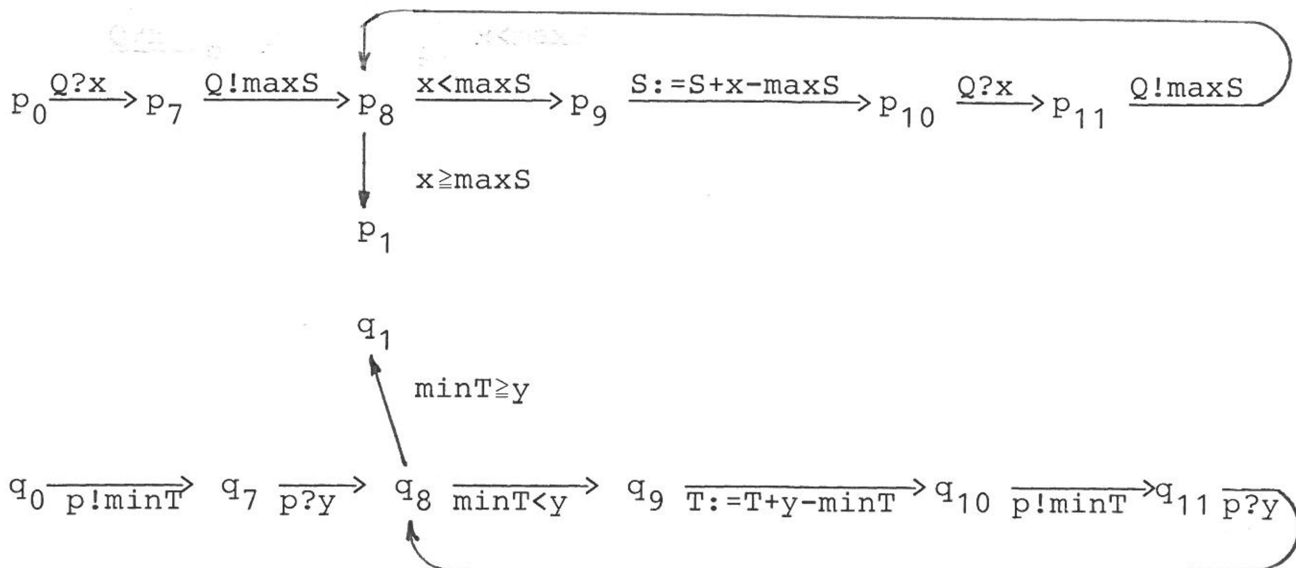
The much studied language for communicating sequential programs CSP [Ho] is an extension of GCL that allows for synchronized communications between processes.

A somewhat intricate CSP program is:

```
[P::Q?x; Q!maxS; do x<maxS→S:=S+x-maxS; Q?x; Q! maxS od
‖Q::P!minT; P?y; do minT<y→t:=t+y-minT; P!minT; P?y od
]
```

Informally, we can describe the behaviour of this program by

1) Suppose we have pebbles at the entry nodes of the two graphs



2) The P-pebble can move freely except on the edges from p_0, p_7, p_{10} and p_{11}.

3) The Q-pebble can move freely except on the edges from q_0, q_7, q_{10} and q_{11}.

4) The P-pebble can move along $p_0 \rightarrow p_7$ or $p_{10} \rightarrow p_{11}$ if and only if the Q-pebble can move at the same time along $q_0 \rightarrow q_7$ or $q_{10} \rightarrow q_{11}$.

5) The P-pebble can move along $p_7 \rightarrow p_8$ or $p_{11} \rightarrow p_8$ if and only if the Q-pebble can move at the same time along $q_7 \rightarrow q_8$ or $q_{11} \rightarrow q_8$.

The pebble movements for the program correspond to the firing of transitions in the Petri net

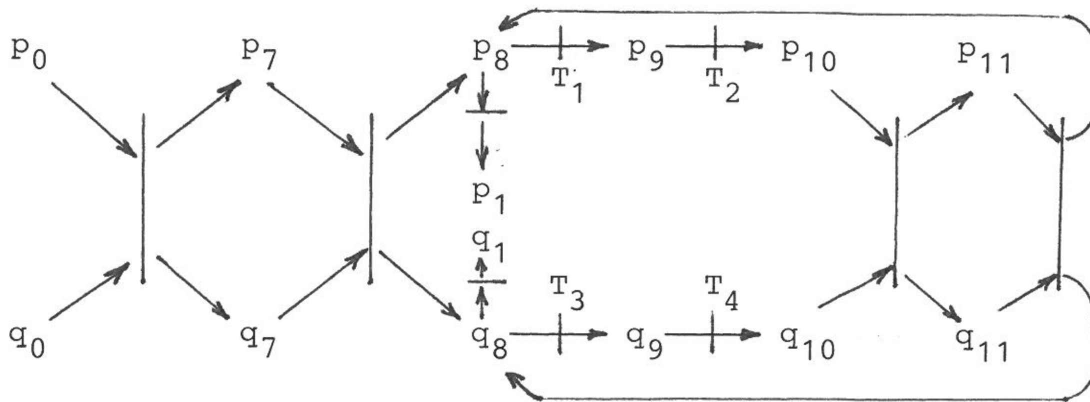


Figure 4
Petri net model for a CSP program

The freedom in pebble rules (2) and (3) is reflected by the fact that the transitions T_1 and T_2 can be concurrent with transitions T_3 and T_4. However, this true concurrency can be reduced to non-determinism because CSP processes do not share variables. For this reason the pebble movements for the program also correspond to traversals of paths in the graph.

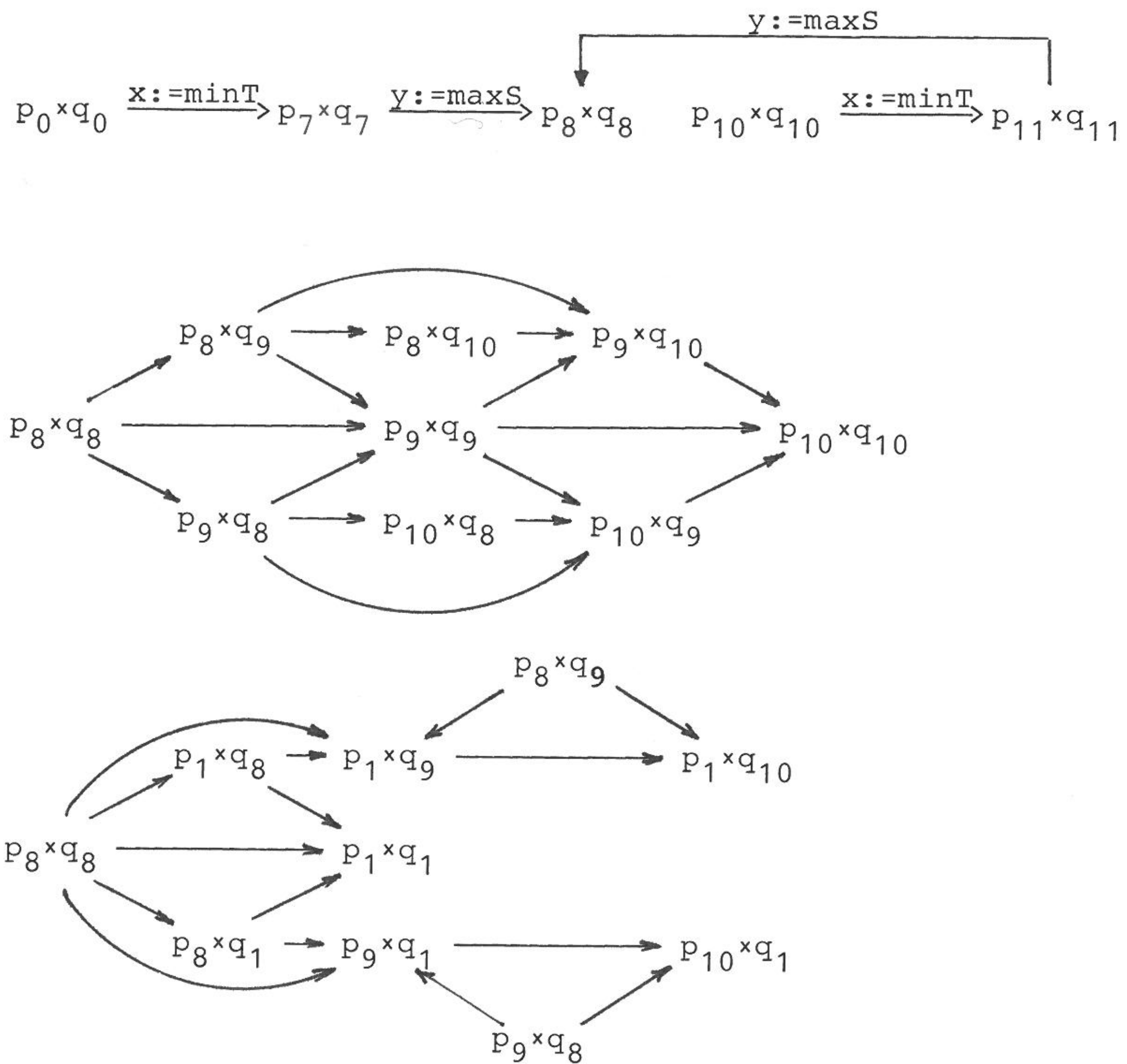


Figure 5

Graph model for a CSP program

The syntax of CSP can be given by adding the following productions to those of GCL:

```
<command>::=<communication>|[<process list>]
<guarded commands>::=<communication>→<command>|
                      <test>; <communication>→<command>
<communication>::=<name>?<variable>|<name>!<expression>
<process list>::=<name>::<command>|
                 <process list>||<process list>
```

Figure 6 is the extension of figure 2 that gives a meaning to communication commands.

command	configurations	move
<name>?<variable>	0,1,2	$0 \xrightarrow{\text{[name?variable]}} 1$
<name>!<expression>	0,1,2	$0 \xrightarrow{\text{[name!expression]}} 1$

Figure 6

Semantics of CSP communication commands

We can take the GCL meaning of guarded commands if we allow ?<communication> and ?<test>;<communication> as move labels.

Example

The CSP command "Q?x;Q!maxS;

do x>maxS→S:=S+x-maxS; Q?x; Q!maxS od"

gives the diagram with configurations (0,1,7,8,9,10,11) and moves

$0 \xrightarrow{\text{Q?x}} 7$, $7 \xrightarrow{\text{Q!maxS}} 8$,

$8 \xrightarrow{\text{?x<maxS}} 1+9$, $9 \xrightarrow{\text{S:=S+x-maxS}} 10$,

$10 \xrightarrow{\text{Q?x}} 11$, $11 \xrightarrow{\text{Q!maxS}} 8$.

□

The diagram for a process list command like

$$[p_1::c_1||p_2::c_2||\ldots||p_m:c_m]$$

is built in two stages: the diagram operator $\textcircled{\mid}$ is used to express the concurrency, then the diagram operator $\textcircled{\upharpoonright}$ is used to capture synchronization and termination in CSP. The concurrency operator $\textcircled{\mid}$ is the one we used to build the diagram or the SCCS agent $p_1|p_2$ from the diagrams for the agents p_1 and p_2. The restriction operator $\textcircled{\upharpoonright}$ is the one we used to build the diagram for the SCCS agent $p\upharpoonright A$ from the diagram for agent p.

Let $D_1D_2\ldots D_m$ be the diagrams for the commands $C_1C_2\ldots C_m$ is our process list command. Let $D'_1D'_2\ldots D'_m$ be the diagrams we get from $D_1D_2\ldots D_m$ by adding the move $1 \xrightarrow{\underline{\text{terminate}}} 1$ and adding <u>in</u> p_j after every communication in C_j. The first stage in our diagram building is completed by defining D_0 as $D'_1\textcircled{\mid}D'_2\textcircled{\mid}\ldots\textcircled{\mid}D'_m$ with $1\times1\times\ldots\times1$ as its sink configuration |. We start the second stage by defining SYNC as the class of actions $\alpha_1\times\ldots\times\alpha_m$ satisfying

$$\alpha_i \text{ is } [P_j!e \ \underline{\text{in}}\ P_i] \quad \text{or} \quad [T_i;\ P_j!e\ \underline{\text{in}}\ P_i]$$

$$\equiv \alpha_j \text{ is } [P_i?v\ \underline{\text{in}}\ P_j] \quad \text{or} \quad [T_j;\ P_i?v\ \underline{\text{in}}\ P_j]$$

We capture the CSP termination convention by defining END as the class of actions $\alpha_1\times\ldots\times\alpha_m$ satisfying

$$\alpha_i \text{ is } [T_i;\ P_j!e\ \underline{\text{in}}\ P_i] \quad \text{or} \quad [T_i;\ P_j!v\ \underline{\text{in}}\ P_i]$$

$$\equiv \alpha_j \text{ is } \underline{\text{terminate}}$$

Now we could define the diagram D for our process list command as $D_0\textcircled{\upharpoonright}(\text{SYNC}\cup\text{END})$. A more precise definition which shows how "process lists are commands" is: 0,1,2 are the configurations and the only move is $0 \xrightarrow{D_0\textcircled{\upharpoonright}(\text{SYNC}\cup\text{END})} 1+2$.

Example

Consider the CSP command

$$[P_1::\underline{do}\ P_2!x\rightarrow\underline{skip}\ \underline{od}||P_2::\underline{skip}]$$

The individual processes have the diagrams

D_1::configurations 0,1,2,7 moves $0 \xrightarrow{[?P2!x]} 1+7,\ 7 \xrightarrow{skip} 0$

D_2::configurations 0,1,2 move $0 \xrightarrow{skip} 1$

The diagram $D_0 = D_1' \textcircled{|} D_2'$ has configurations

0×0, 0×1, 1×0, 1×1, 7×0, 7×1

and moves

(a) $0\times0 \xrightarrow{[?P_2!x \text{ in } P_1]\times skip} 1\times0 + 7\times0 + 1\times1 + 7\times1$

(b) $0\times0 \xrightarrow{skip\times skip} 0\times1 + 0\times0$

(c) $0\times1 \xrightarrow{[?P_2!x \text{ in } P_1]\times st} 1\times1$

(d) $0\times1 \xrightarrow{skip\times st} 0\times1$

(e) $1\times0 \xrightarrow{st\times skip} 1\times1 + 1\times0$

(f) $1\times1 \xrightarrow{st\times st} 1\times1$

(g) $7\times0 \xrightarrow{skip\times skip} 0\times0 + 0\times1 + 7\times1 + 7\times0$

(h) $7\times1 \xrightarrow{skip\times st} 0\times1 + 7\times1$

where st abbreviates "skip or terminate". The action class SYNC is

(skip×skip, skip×terminate, terminate×skip, terminate×terminate)

while "[?P_2!x in P_1]×terminate" is the only action in the class END. The diagram for our CSP command is the same as D_0 except for

- move (a) is dropped
- move (c) has the label "[?P_2!x in P_1]×terminate"
- the configurations are assigned new integers, with 0 being assigned to 0×0 and 1 to |x|.

Perhaps the graph representation of this diagram in figure 6 will be welcome.

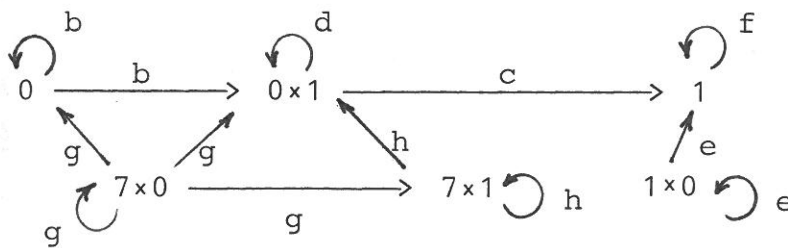


Figure 6

Graph of a diagram

□

We ought to say something about the CSP kind and unkind choice discussion [Fr]. In our view this depends on how one assigns "state transformations" to product moves. When a guarded command has only Boolean tests as guards, one can assign a state transformation to the isolated move

$$0 \xrightarrow{?B_7 \ \dots \ B_j} 3 + 7 + \dots + j.$$

This cannot be done when communications occur in the guards, one must know the product in which the move occurs before it can be assigned a meaning. The same is true when communications occur as commands; one cannot assign a state transformation to the move

$$0 \xrightarrow{P_j?v \quad \text{in } P_i} 1$$

in isolation, only when it occurs in a product with the move

$$0 \xrightarrow{P_i!e \text{ in } P_j} 1;$$

the state transformation assigned to this combination is the one that would naturally be assigned to the move $0 \xrightarrow{v:=e} 1$. When assigning state transformations to moves one has to pay due attention to scope of identifiers and other environment problems. However these problems are of little importance when one is trying to model (understand) a program in a language that allows parallelism.

<u>Example</u>

In the CSP program in the beginning of this section only Boolean tests occur, so all communication commands can be replaced by assignments when we give the diagram for the program

Configurations (0,1,7,8,9,10,11) (0,1,7,8,9,10,11)
source 0×0 sink 1×1

$$0\times0 \xrightarrow{x:=minT} 7\times7 \qquad 7\times7 \xrightarrow{y:=maxS} 8\times8$$

$$10\times10 \xrightarrow{x:=minT} 11\times11 \qquad 11\times11 \xrightarrow{y := maxS} 8\times8$$

$$8\times8 \xrightarrow{?x<maxS \otimes skip} 1\times1 + 9\times1 + 1\times9 + 9\times9$$

$$8\times8 \xrightarrow{?x<maxS \otimes skip} 1\times8 + 9\times8$$

$$8\times8 \xrightarrow{skip \otimes ?minT<y} 8\times1 + 8\times9$$

$$8\times9 \xrightarrow{?x<maxS \otimes T4} 1\times10 + 9\times10$$

$$8\times9 \xrightarrow{?x<maxS \otimes skip} 1\times9 + 9\times9$$

$$8\times9 \xrightarrow{skip \otimes T4} 8\times10$$

$$8\times10 \xrightarrow{?x<maxS \otimes skip} 1\times10 + 9\times10$$

$$9\times 8 \xrightarrow{\text{T2 x ?minT<y}} 10\times 1 + 10\times 9$$

$$9\times 8 \xrightarrow{\text{skip x ?minT<y}} 9\times 1 + 9\times 10$$

$$9\times 8 \xrightarrow{\text{T2 x skip}} 10\times 8$$

$$9\times 9 \xrightarrow{\text{T2 x T4}} 10\times 10 \;,\; 9\times 9 \xrightarrow{\text{T2 x skip}} 10\times 9$$

$$9\times 9 \xrightarrow{\text{skip x T4}} 9\times 10$$

$$9\times 10 \xrightarrow{\text{T2 x skip}} 10\times 10 \quad 10\times 9 \xrightarrow{\text{skip x T4}} 10\times 10$$

$$10\times 8 \xrightarrow{\text{skip x ?minT<y}} 10\times 1 + 10\times 9$$

$$1\times 8 \xrightarrow{\text{skip x ?minT<y}} 1\times 1 + 1\times 9 \qquad 1\times 9 \xrightarrow{\text{skip T4}} 1\times 10$$

$$8\times 1 \xrightarrow{\text{?maxS<x x skip}} 1\times 1 + 9\times 1 \qquad 9\times 1 \xrightarrow{\text{T2 skip}} 1\times 1$$

where T2 ≡ "S:=S+x-maxS" and T4 ≡ "T:=T+y-minT".

Some moves of the diagram are not given because their sources are not accessible from the configuration 0×0. If the reader looks at figure 5 s/he will see that the corresponding edges of the graph model were also omitted.

□

4. COMPUTATIONAL MODEL FOR ADA

The programming language ADA [Ad1] provides powerful mechanisms for parallelism, but these have not yet been defined formally. It is necessary to describe the relation between the formal definition of ADA [Ad2] and our diagram models, before the representation of parallelism can be described. The formal definition gives:

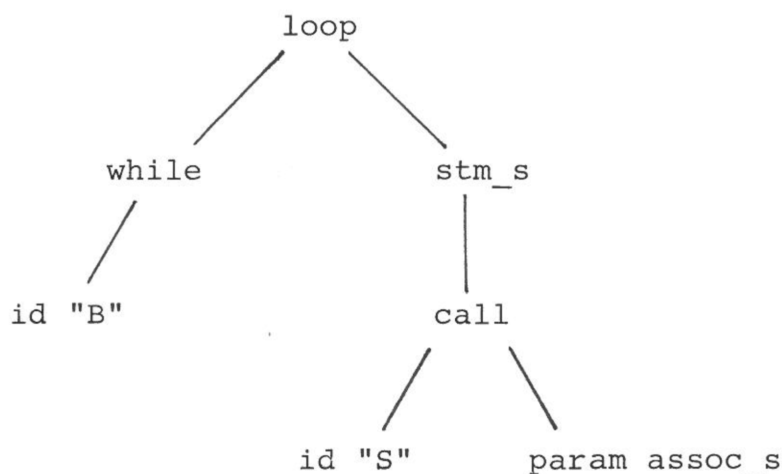
(1) the abstract syntax of ADA, so that each syntactically correct program has a tree representation;

(2) the static semantics, that typechecks a tree and rewrites it in a form that can be used by

(3) the dynamic semantics, which define the behaviour of the program.

Example

The program fragment "while B loop S; end loop" is a syntactically correct ADA loop statement with the tree representation



The static semantics does not change this tree and the dynamic semantics uses it as the actual parameter for the formal parameter "loop" in the function specification

```
function EXEC_WHILE_LOOP (loop:TREE; env: D_ENV;
                          store: STORE; cont: EXEC_CONT)
                  return IO_MAP;
```

We can consider the body of this function as a way of giving the meaning to the moves whose labels are syntactically correct ADA while loop statements - in this example, edges with the label while B loop S; endloop.

The passage through the abstract syntax and the static semantics may affect the correspondence between program points and the configurations in our diagram model. However, the formal definers of ADA write (page 1.14)

> The functions used in Dynamic Semantics are partitional into three groups those defining the elaboration of declarations ... those defining the evaluation of expressions ... those defining the execution of statements.

So we need only take declarations, expressions and statements as the labels of moves in our diagram model. Because of nesting there are many feasible ways of choosing configurations in our diagrams, but one possibility is

> Pick the largest expressions, the longest declarations and the smallest statements.

Example

Instead of choosing $0 \xrightarrow{\text{while B loop S; endloop}} 1$ we should have chosen the equivalent

Configurations (0,1,7)

Moves $0 \xrightarrow{?B} 1 + 7$

$7 \xrightarrow{S} 0$

where the move label S corresponds to the call of a procedure. This procedure has a declaration which was elaborated on some

earlier move with the label: procedure S ... end S. Moves with this label give a meaning to moves with labels S, but we can define this meaning by using the diagram model of the procedure body.

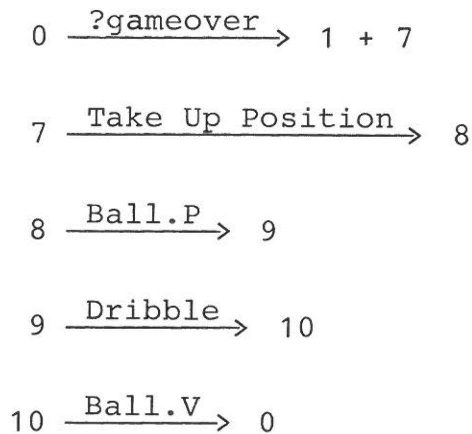
Now we can turn to parallelism in ADA, and start by describing the diagrams for task bodies. There are three kinds of statements that can only occur in a task body: delay statements, accept statements and selective wait statements. Because ADA makes no assumptions about the speed of processors, a delay statement can be represented by a move with label skip (just like a null statement). An accept statement can be represented by a move with a label like - accept E do ... end. Such a move gives a meaning to a move with label E in the same way that the body of a subprogram S gives meaning to moves with label S. Before discussing selective wait statements, let us look at the

Example

We shall later give the diagram model for an ADA program that simulates a soccer match. In this program there will be 22 tasks with the body:

```
task body PLAYER is
begin
    loop
       exit when game over;
       Take Up Position;
       Ball.P;
       Dribble;
       Ball.V;
    end loop
end PLAYER;
```

The diagram for this has configurations (0,1,7,8,9,10) and moves



□

There is an important difference between selective wait statements with an else part and those without, because of the word "immediately" in the ADA requirement (p. 9.10)

- If no alternative can be immediately selected and there is an else part, the else part is executed. If there is no else part, the task waits until an open alternative can be selected.

Like the distinction between kind and unkind choice in CSP, the distinction between selective waits with and without else parts is not completely apparent in the diagram, but only reappears when the diagram moves are given a meaning.

Suppose $B_1 \ldots B_n$ are the conditions in a selective wait statement. Selective wait statements without an else part are allowed to have a delay alternative or a terminate alternative but not both. A delay alternative gives an edge labelled skip, while a terminate alternative gives an edge labelled terminate to the normal exit point.

Example

The program fragment

select when B ⇒ accept E do ... end; end select

has the diagram:

Configurations (0,1,4,7)

Moves $0 \xrightarrow{\text{B?}} 4 + 7$

$7 \xrightarrow{\text{accept E do end}} 1$

The insertion of "else S;" after the semicolon would give the diagram

Configurations (0,1,7,8)

Moves $0 \xrightarrow{\text{?B}} 8 + 7$

$7 \xrightarrow{\text{accept E do end}} 1$

$8 \xrightarrow{\text{S}} 1$

the insertion of "or terminate;" would have given the diagram:

Configurations (0,1,3,7)

Moves $0 \xrightarrow{\text{?B}} 3 + 7$

$7 \xrightarrow{\text{accept E do end}} 1$

We should also give an example of the diagrams for the two kinds of non-deterministic statements which can occur outside a task body. The timed entry call "select urgent; or delay + 5.0; end select" has the diagram

Configurations (0,1,2)

Moves $0 \xrightarrow{\text{urgent}} 1$

$0 \xrightarrow{\text{skip}} 1$

whereas the conditional entry call "select urgent; else S; end select" has the diagram

Configurations (0,1,2)

Moves $0 \xrightarrow{\text{urgent}} 1$

$0 \xrightarrow{S} 1$

<u>Example</u>

In the ADA program that simulates a soccer match we will have one task with the body

```
task body BALL is
begin
    loop
       select
          accept P do end;
       or terminate;
       end select;
       accept V do end;
    end loop;
end BALL;
```

The diagram for this has configurations (0,1,2,3,7,8) and moves

$0 \xrightarrow{\text{?true}} 3 + 7$

$7 \xrightarrow{\text{accept P do}} \xrightarrow{\text{end}} 8$

$3 \xrightarrow{\text{terminate}} 1$

$8 \xrightarrow{\text{accept V do}} \xrightarrow{\text{end}} 0$

□

The diagram for a task body must be modified to allow for its abortion by another task. One must add the move

$j \xrightarrow{\text{aborted}} 2$

for each configuration j in the diagram of a task body that can be aborted.

Once we have the diagrams for the task bodies in an ADA program they can be combined using the diagram combinator ⓛ.

Example

In the ADA program that simulates a soccer match the diagrams for the one BALL task and the 22 player tasks can be combined using the ⓛ combinator with the diagram for the body

```
procedure Soccer is
   task type PLAYER;
   task BALL is entry P;
                entry V;
   end BALL;
   gameover: BOOLEAN:=FALSE;
   us,them: array(1..11) of PLAYER;
begin
   delay (90 + 60.0);
   gameover:=TRUE;
end Soccer;
```

The diagram for this body has the configurations (0,1,7) and the moves

$$0 \xrightarrow{\text{skip}} 7$$

$$7 \xrightarrow{\text{gameover:=TRUE}} 1$$

□

The interesting part of the ADA semantics is the use of the diagram combinator ⓛ to capture the synchronisation rules. ADA tasks interact in one of three ways: rendezvous, termination or abortion. For each of these ways of interacting we must define a class of allowable action products $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$. If nested interactions were forbidden in ADA, these classes could be defined by

class	requirement on $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$
REND	$(\alpha_j = \underline{\text{accept}}\ E\ \underline{\text{do}}\ \ldots\ \underline{\text{end}}) .\equiv. (\exists! i)(\alpha_i = E)$
TERM	$(\forall i)\ (\alpha_i = \underline{\text{terminate}})$
ABORT	$(\alpha_j = \text{aborted}) .\equiv.\ (\exists i)(\alpha_i = \text{task } j \text{ aborts})$

Example

Consider the ADA program Soccer when the BALL task and the 22 PLAYER tasks are active. The requirement for the action
$\alpha_S \times \underline{\text{accept}}\ P\ \underline{\text{do}}\ \underline{\text{end}} \times \alpha_{V1} \times \ldots \times \alpha_{V11} \times \alpha_{T1} \times \ldots \times \alpha_{T11}$
to be in REND is

"precisely one of $\alpha_{V1} \ldots \alpha_{T11}$ is BALL.P";

the requirement for the action
$\alpha_S \times \underline{\text{accept}}\ V\ \underline{\text{do}}\ \underline{\text{end}} \times \alpha_{V1} \times \ldots \times \alpha_{V11} \times \alpha_{T1} \times \ldots \times \alpha_{T11}$
to be REND is

"precisely one of $\alpha_{V1} \ldots \alpha_{T11}$ is BALL.V".

As always the class TERM has only one action. The action class ABORT is empty because abortion is not used in the program Soccer.

If we define IND as the class of $\alpha_S \times \alpha_B \times \alpha_{V1} \times \ldots \times \alpha_{V11} \times \alpha_{T1} \times \ldots \times \alpha_{T11}$ satisfying

$$\alpha_B \neq \underline{\text{accept}}\ P\ \underline{\text{do}}\ \underline{\text{end}}\ \&\ \alpha_B \neq \underline{\text{accept}}\ V\ \underline{\text{do}}\ \underline{\text{end}}\ \&\ \alpha_B \neq \underline{\text{terminate}}$$
$$\&\ \alpha_S \neq \underline{\text{terminate}}$$
$$\&\ (\forall j)\ (\alpha_{Vj} \neq \text{BALL.P}\ \&\ \alpha_{Vj} \neq \text{BALL.V}\ \&\ \alpha_{Vj} \neq \underline{\text{terminate}})$$
$$\&\ (\forall j)\ (\alpha_{Tj} \neq \text{BALL.P}\ \&\ \alpha_{Tj} \neq \text{BALL.V}\ \&\ \alpha_{Tj} \neq \underline{\text{terminate}})$$

then we can define the diagram for the parallel part of the Soccer program as

$$D_1 = D_0 \mid (\text{REND} \cup \text{TERM} \cup \text{IND})$$

where D_0 is the result of | on the diagrams for the main program, the ball and the 22 players. One of the possible "computation histories" in D_1 is

$0_S \times 0_B \times 0_{V1} \times 0_{V2} \times \dots \times 0_{T11}$

↓ skip × ?true × ?gameover × skip × ... × skip

$7_S \times 7_B \times 7_{V1} \times 0_{V2} \times \dots \times 0_{T11}$

-- player $U_S(1)$ takes up position while all other players sleep

↓

$7_S \times 7_B \times 8_{V1} \times 0_{V2} \times \dots \times 0_{T11}$

↓ gameover:=TRUE × <u>accept</u> P <u>do</u> <u>end</u> × BALL.P × 0_{V2} × ... × 0_{T11}

-- player $U_S(1)$ gets ball while whistle blows

$1_S \times 8_B \times 9_{V1} \times 0_{V2} \times \dots \times 0_{T1}$

-- player $U_S(1)$ dribbles

↓

$1_S \times 8_B \times 10_{V1} \times 0_{V1} \times \dots \times 0_{T11}$

↓ skip × <u>accept</u> V <u>do</u> <u>end</u> × BALL.V × 0_{V2} × ...×0_{T11}

-- player $U_S(1)$ kicks the ball

$1_S \times 0_B \times 0_{V1} \times 0_{V2} \times \dots \times 0_{T11}$

↓ skip × ?true × skip × ... × skip

$1_S \times 3 \times 0_{V1} \times 0_{V2} \times \dots \times 0_{T1}$

↓ terminate × ... × terminate

$1_S \times 1_B \times 1_{V1} \times 1_{V_2} \times \dots \times 1_{T11}$

□

In our definition of the class ABORT we have assumed that the diagram for every task body has the move

$$c \xrightarrow{\text{aborted}} 2$$

for every configuration c. Because abortion can happen while a task is actually making a move, this assumption is not enough and we must also assume the move

$$c \xrightarrow{\text{aborted } \alpha} 2$$

for every diagram move $c \xrightarrow{\alpha} c'$.

The informal definition of ADA describes the effect of abortion on a task in a rendezvous or one that has made an entry call, but it is not so clear about the effect of abortion on a task that is terminating or one that is aborting some other task. Another complication is that ADA rendezvous can be nested - entry calls may occur between the do and end in an accept statement. In a later paper we will describe how the classes REND, TERM and ABORT can be modified to cover such nested interactions.

Now let us describe the creation and destruction of ADA tasks. For the sake of simplicity let us ignore the possibility of abortion or the raising of an exception while declarations are being elaborated. In this case tasks are created by a move like

$$j \longrightarrow k \times 0_1 \times \ldots \times 0_m$$

where j is the configuration after all declarations have been elaborated. Analogously the destruction of tasks is given by a move like

$$1_0 \times 1_1 \times \ldots \times 1_m \xrightarrow{\text{end}} 1.$$

Example

The complete diagram for the ADA program that simulates a soccer match has the moves of D_1, the extra moves

$$0 \xrightarrow{\text{task type PLAYER}} 7$$

$$7 \xrightarrow{\text{task BALL is entry P; entry V; end BALL}} 8$$

$$8 \xrightarrow{\text{gameover: BOOLEAN := TRUE}} 9$$

$$9 \xrightarrow{\text{us, them: array(1..11) of PLAYER}} 10$$

$$10 \xrightarrow{\text{begin}} 0_S \times 0_B \times 0_{V1} \times \ldots \times 0_{T11}$$

$$1_S \times 1_B \times 1_{V1} \times \ldots \times 1_{T11} \xrightarrow{\text{end}} 1$$

the configurations of D_1 and the extra configurations (0,1,7,8,9,10).
□

So far we have described how any ADA program can be modelled by a diagram, but we have not given any meaning to moves in a diagram. The formal semantics of sequential ADA [Ad2] gives a meaning to moves whose labels are within the sequential part of ADA. There is no reason why a move with a label like "accept P do end" should have a meaning in isolation; the synchronization rules only allow such moves in combination with an appropriate entry call, so only combined rendezvous moves must have a meaning.

Example

Both rendezvous moves in our soccer program

$$7_B \times 8_P \xrightarrow{\text{accept P do end} \times \text{BALL.P}} 8_B \times 9_P$$

$$8_B \times 10_P \xrightarrow{\text{accept V do end} \times \text{BALL.V}} 0_B \times 0_P$$

have the identity state transformation as their meaning.
□

Part of the semantics of ADA places restrictions on the sequences of moves that can occur in a diagram model of an ADA program. Because calls on the same entry are queued, move sequences in our ADA soccer program are fair - the ball will honour all rendezvous requests impartially, it will not favour a particular player or team.

Are our diagram models of ADA programs built on sand, are they unsatisfactory because they replace the true parallelism of ADA by synchronized move sequences? The answer is NO for two reasons - the Petri net argument in [Ma2] and the fact that ADA can be implemented correctly on a single processor. If one has the luxury of a multiprocessor implementation of ADA, then the global state and time of a diagram model may be very different from the reality of concurrent computation but the real observable behaviour of an ADA program must correspond to a possible behaviour of the diagram model of the program.

SUMMARY

It has been shown that diagrams are a suitable computational model for programs in ADA and other languages with parallelism. It seems clear that the model can also handle exception propagation and other sequential ADA features, that we not discussed here.

REFERENCES

[Ad1] Reference Manual for ADA, SIGPLAN 1982.

[Ad2] Formal definition of ADA, C11-Honeywell Bull, 1981, Paris.

[Di] E.W. Dijkstra: A discipline of programming. Prentice Hall 1976.

[Fr] N. Francez, C.A.R. Hoare, D.J. Lehmann, W.P. de Roever: Semantics of nondeterminism, concurrency and communication. J. Comp. Sys. Sci. 19 (1979) 290-308.

[Ho] C.A.R. Hoare: Communicating Sequential Processes. CACM 21 (1978) 666-677.

[Ma1] B.H. Mayoh: Program models: meaning and proof. DAIMI PB-157 (1983).

[Ma2] B.H. Mayoh: Parallelism in ADA: program design and meaning. Springer LNCS 83 (1980) 256-268.

[Mi] R. Milner: Calculi for synchrony and asynchrony. Edinburgh report CSR-104-82.