Chapter 6 : The Master/Slave Operator

This operator, which was introduced in chapter 2, can be modelled in a similar (though preferable) manner over M. Because of the use made of hiding in its definition it will be necessary to restrict its definition to cases where the set T of communicated values is finite.

6.1 $(A \| a::B) = (A_\Sigma \|_{a.\Gamma} a.(swap!?(B))/(a.\Gamma)$

where $\Gamma = T \cup ?T \cup !T$ and swap is defined

$swapab(A) = \{(swapab(w), swapab(X)) \mid (w,X) \in A\}$

($swapab(X)$ for $X \subseteq \Sigma$ is defined to be the natural extension of the definition for Σ given in chapter 2).

6.2 Examples

a) If $A = A_1 \| A_2 \| \ldots \| A_k$ and some of the A_i might have to execute a critical section which needs to be executed in parallel with no other critical section then if we guard each c.s. with a request to a suitable (joint) slave this condition can be ensured.

e.g. $A'_i = B_i;(a!i \to \underline{skip});CS_i;(a?i \to \underline{skip});C_i$

(where $A_i = B_i;CS_i;C_i$)

$D = ?n:\{1,2,..,k\} \to !n \to D$

$A^* = ((A'_1 \| A'_2 \| \ldots \| A'_k) \| a::D)$

b) Five dining philosophers: ($\boxplus, \boxminus \equiv$ mod 5 arithmetic)

If $PHIL_i \Leftarrow a.sit.i \to i.pickup_i \to i.pickup_{i\boxplus 1} \to i.eat \to$

$i.putdown_{i\boxplus 1} \to i.putdown_i \to a.getup.i \to PHIL_i$

$FORK_i \Leftarrow i.pickup_i \to i.putdown_i \to FORK$

$\Box\, i\boxminus 1.pickup_i \to i\boxminus 1.putdown_i \to FORK$

$B = B_0$ where $B_0 \Leftarrow sit.j:\{0,..4\} \to B_1$

$i \in \{1,2,3\} \Rightarrow B_i \Leftarrow sit.j:\{0,..4\} \to B_{i+1}$

$\Box\ getup.j:\{0,..4\} \to B_{i-1}$

$B_4 \Leftarrow getup.j:\{0,..4\} \to B_3$

Then $(\|_{i=0}^{4} PHIL_i) \| a::B) \| (\|_{i=0}^{4} FORK_i)$ is free of deadlock.

This last example is reasonably easy to prove using the techniques that were developed in chapter 5 and those which we will develop in this chapter and the next.

6.3 Lemma

If T is finite then $(A \| a::B)$ is a well-defined continuous function of its parameters. Furthermore we have the following criterion for membership of $(A \| a::B)$:

$(w,X) \in (A \| a::B)$ if and only if

either there is some $w' \leqslant w$ such that

$$\{t \in dom(A) \mid (t\restriction a.\Gamma \in a.swap!?(dom(B))) \ \&\ (t/a.\Gamma = w')\}$$

is infinite

or there is some $(u,U) \in A$ and $(v,V) \in B$ such that

$$X \cup a.\Gamma = U \cup (a.swap!?(V \cap \Gamma))$$

The proof of this is very similar to that of 5.18, the corresponding result on the $(A \gg B)$ operator.

If the first case holds of any (w,X) then we say that $(A \| a::B)$ contains infinite internal chatter.

In the or clause we say that $((u,U),(v,V))$ is a derivation for (w,X) in $(A \| a::B)$ or equivalently that $((u,U),(v,V)) \Leftrightarrow (w,X)$ in $(A \| a::B)$.

6.4 Theorem

Suppose that each of $(A \| a::B)$ and $(A \| b::C)$ is free of infinite internal chatter and that $a.\Gamma \cap b.\Gamma = \emptyset$.

Then $((A \| a::B) \| b::C) = (A \| b::C) \| a::B)$.

The proof of this is an immediate corollary to lemma 5.35, the same result used in proving 5.19.

Theorem 2.38 also carries straight over to this model:

6.5 Theorem

Define C^a_j (for $a \in \Sigma$ and $j \in Z$) as follows:

$$C^a_j(A) \triangleq \forall w \in dom(A).\ (\#(w\restriction\Gamma) - \#(w\restriction a.\Gamma)) \geqslant \min(j, \#(w\restriction(\Gamma \cup a.\Gamma)))\ .$$

a) $C^a_j(A)$ is a (strongly) continuous predicate.

b) $C^a_j(A) \Rightarrow (A \| a::B)$ is free of infinite internal chatter.

c) $C^a_k(A) \Rightarrow \forall n.\forall B.\ (A \| a::B)\restriction n+k+1 = (A \| a::B\restriction n)\restriction n+k+1$.

d) $C^a_j(A) \Rightarrow C^a_j(A \| b::B)$.

The proof of (a) is the same as 2.38(a).
The proof of (b) is similar to (though easier than) that of 5.20.
The proofs of (c) and (d) are similar to 2.38(b)&(c).

We will shortly see that the master/slave operator is a very useful tool in defining processes by recursion. As in chapter 2 the above results help us to identify the cases where it is a constructive or non-destructive function of its second (slave) argument. The following is a list of technical lemmas of a computational nature which will subsequently be used freely and informally in proofs.

6.6 Lemma Rules for $(A \| a::B)$

(i) If $A^o \cap a.\Sigma = \emptyset$ then $(A \| a::B)(\langle\rangle) = A(\langle\rangle)$, $(A \| a::B)^o = A^o$
and $(A \| a::B)\ \underline{\text{after}} x = (A \underline{\text{after}} x \| a::B)$

(ii) $(A \ \underline{\text{or}}\ B \| a::C) = (A \| a::C)\ \underline{\text{or}}\ (B \| a::C)$
$(A \| a::(B \ \underline{\text{or}}\ C)) = (A \| a::B)\ \underline{\text{or}}\ (A \| a::C)$

(iii) $(a!x \to A \| a::(?x:T \to B_x)) = (A \| a::B_x)$
$= ((a!x \to A) \,\square\, (a?x:T \to C_x) \| a::(?x:T \to B_x))$
$= (a!x \to A \| a::((?x:T \to B_x) \,\square\, (!y \to C)))$

(iv) $(a?x:T \to A_x \| a::(!x \to B)) = (A_x \| a::B)$
$= ((a?x:T \to A_x) \,\square\, (!y \to C) \| a::(!x \to B))$
$= (a?x:T \to A_x \| a::((!x \to B) \,\square\, (?y:T \to C_y)))$

(v) Define a function $f:\Sigma \to \Sigma$ as follows:
$f(y) = a!x$ if $y = ?x$ for any x
$= a?x$ if $y = !x$ for any x
$= y$ otherwise.
If $B^o \subseteq a.\Sigma$ and $B^o \cap f(C^o) = \emptyset$ then $(A \,\square\, B \| a::C) = (A \| a::C)$

(vi) If $A^o \cap a.\Sigma = \emptyset$ & $B^o \subseteq a.\Sigma$ & $\neg\exists X \in B(\langle\rangle), Y \in C(\langle\rangle).X \cup f(Y) = a.\Sigma$
then $(A \,\square\, B \| a::C) = ((A \| a::C) \,\square\, (B \| a::C))\ \underline{\text{or}}\ (B \| a::C)$.
(The condition here ensures that B and C cannot deadlock, thus ensuring that some hidden transition will eventually take place if nothing else is offered.)
e.g. $(?x:T \to A_x \,\square\, a?x:T \to B_x \| a::(!y \to C)) =$
$((?x:T \to (A_x \| a::(!y \to C))) \,\square\, (B_y \| a::C))\ \underline{\text{or}}\ (B_y \| a::C)$

(vii) If $A^o \cap a.\Sigma = \emptyset$ and $B^o \subseteq a.\Sigma$ and $B^o \cap f(C^o) \neq \emptyset$
and $\exists X \in B(\langle\rangle), Y \in C(\langle\rangle)$ s.t. $X \cup f(Y) = a.\Sigma$
then $(A \,\square\, B \| a::C) = (A \| a::C)\ \underline{\text{or}}\ (B^* \| a::C^*)$
where $D^*(w) = \{X \mid X \cap D^o = \emptyset\}$ if $w = \langle\rangle$
$= D(w)$ otherwise.
(If B & C deadlock on the first step then A must be allowed to take precedence.)

Rules (i) - (vii) deal essentially with the first step behaviour of $(A \| a::B)$. It is possible however to derive a few more general ones. ((viii) is just a restatement of 6.4.)

(viii) If $a \neq b$ and each of $(A \| a::B)$ and $(A \| b::C)$ is free of infinite internal chatter then
$((A \| a::B) \| b::C) = ((A \| b::C) \| a::B)$.

(ix) If $(A \| a::B)$ is free of infinite internal chatter and $a.\Sigma \cap Y = \emptyset$ then $((A_{X \cup \{a\}} \|_Y B) \| a::C) = ((A \| a::C)_X \|_Y B)$.

Each of (i) - (vii) is quite easy to prove directly from the definitions of the operators involved. (ix), like (viii) is a consequence of 5.35.

We will concentrate in the next few pages on seeing how $(A \| a::B)$ can be used as a recursive tool, and on how processes defined recursively with this operator can be proved correct by our methods.

In the following examples the recursive scheme used is $R \Leftarrow (X \| a::R)$ (or some slight variant), where X is a known process. This says that R is the process which X is when it calls R as its slave. One can think of R as a process which is allowed to call itself as a subroutine running in parallel. In 6.7 and 6.8 exactly this scheme is used, modelling a stack and a buffer respectively; in 6.9 we allow the process to call two independent versions of itself and model C.A.R. Hoare's Quicksort.

6.7 Example: Stack

Define the process R recursively by

$$R \Leftarrow (X \| a::R)$$

where
$$X \Leftarrow ?x:T \rightarrow Y_x$$
$$Y_x \Leftarrow ?y:T \rightarrow a!x \rightarrow Y_y$$
$$\quad \Box\, !x \rightarrow Z$$
$$Z \Leftarrow ?x:T \rightarrow Y_x$$
$$\quad \Box\, a?x:T \rightarrow Y_x \quad .$$

R is intended to model an empty stack. Intuitively we can interpret its actions as follows (bearing in mind that it has as a slave a copy of itself labelled "a"):

Initially it is empty and so cannot output. It can however input any element of T and store it (without reference to its slave).
If subsequently it is storing an element of T it can output it to its environment. It can also input any element of T from its environment; if it does this it outputs the old element it was storing to its slave and stores the new one itself (the new one becoming the new top of the stack).
If (at a time other than the start) the control process finds itself not storing any element of T it does not know whether or not its slave is empty. It is therefore prepared either to input the new top of the stack from its slave or from its environment.

We can define a more obviously correct stack as follows by infinite mutual recursion over T*:

$$S_w \Leftarrow ?x{:}T \to S_{\langle x\rangle} \qquad (\text{if } w = \langle\rangle)$$
$$S_w \Leftarrow ?x{:}T \to S_{\langle x\rangle w} \qquad (\text{if } w = \langle y\rangle v,\ y \in T \ \&\ v \in T^*)$$
$$\quad \Box\ !y \to S_v$$

Here S_w represents the stack with contents w; the top of the stack being the leftmost component of w. An empty stack can only input; a stack with top element y can either output y (and lose y from the top of the stack) or input some element which it adds to the stack at the top.

We have not formally defined any correctness criterion for stacks. If this were done it is likely that $S_{\langle\rangle}$ would be much easier to prove correct than R. Indeed one very plausible correcness condition for infinite stacks is congruence with $S_{\langle\rangle}$, a process which it is very easy to prove (for example) free of deadlock. We will therefore content ourselves with a proof of the congruence $R = S_{\langle\rangle}$, leaving out any further work that needs to be done proving $S_{\langle\rangle}$ to be correct.

Define $R_{\langle\rangle} = R$ and $R_{\langle y\rangle w} = (Y_y \,||\, a{::}R_w)$.
Claim that (i) $\forall w.(Z \,||\, a{::}R_w) = R_w$
and (ii) $R_{\langle\rangle} = ?x{:}T \to R_{\langle x\rangle}$ and $\forall y,w.R_{\langle y\rangle w} = ?x{:}T \to R_{\langle xy\rangle w}$
$\quad \Box\ !y \to R_w$.

We will prove these two results not by recursion induction

but by ordinary mutual induction on the length of w.

a) $w = \langle\rangle$

i) $R_w = (X \,||\, a::R)$

$= ?x:T \to (Y_x \,||\, a::R)$ (by rule (i))

$= ?x:T \to R_{\langle x \rangle}$ (by defn. of $R_{\langle x \rangle}$)

ii) $(Z \,||\, a::R_w) = ((?x:T \to Y_x \,\Box\, a?x:T \to Y_x) \,||\, a::R)$

$= (?x:T \to Y_x \,||\, a::R)$ (by rule (v) as $f(R^o) = a!T$)

$= (X \,||\, a::R)$

$= R_w$

b) $w = \langle y \rangle v$ (assuming the results to hold of all shorter w')

(ii) $R_w = (Y_y \,||\, a::R_v)$

$= !y \to (Z \,||\, a::R_v)$

$\Box\, ?x:T \to (a!y \to Y_x \,||\, a::R_v)$ (by rule (i))

$= !y \to R_v$ (inductive hypothesis (i))

$\Box\, ?x:T \to (a!y \to Y_x \,||\, a::(?y:T \to R_{\langle y \rangle v}(\Box\, !z \to R_u)))$

(by inductive hypothesis (ii) where if $v = \langle\rangle$ the bracket is not present otherwise $v = \langle z \rangle u$, say)

$= !y \to R_v$

$\Box\, ?x:T \to (Y_x \,||\, a::R_{\langle y \rangle v})$ (by rule (ii) in either case)

$= !y \to R_v$

$\Box\, ?x:T \to R_{\langle xy \rangle v}$ as desired.

(i) $(Z \,||\, a::R_w) = ((?x:T \to Y_x \,\Box\, a?x:T \to Y_x) \,||\, a::(!y \to R_v \,\Box\, ?x:T \to R_{\langle xy \rangle v})$

(by (ii) above)

$= ((?x:T \to (Y_x \,||\, a::R_{\langle y \rangle v}) \,\Box\, P) \text{ or } P)$

where $P = (Y_y \,||\, a::R_v)$ (by rules (vi), (i) & (iv

now $P = R_{\langle y \rangle v}$

$= !y \to R_v \,\Box\, ?x:T \to R_{\langle xy \rangle v}$ (by (ii) above)

$= ((?x:T \to R_{\langle xy \rangle v}) \,\Box\, ((!y \to R_v) \,\Box\, ?x:T \to R_{\langle xy \rangle v}))$

$\text{or}\ ((!y \to R_v) \,\Box\, ?x:T \to R_{\langle xy \rangle v})$

$= (?x:T \to R_{\langle xy \rangle v}) \,\Box\, (!y \to R_v)$ (by laws of $\Box$ and or)

$= R_{\langle y \rangle v}$ (by (ii) above)

$= R_w$ as desired.

This completes the proof of our two inductive hypotheses, so the two results are proved for all w.

It is now a simple (recursion) induction on the definition of $\underset{\sim}{S}$ to show that $\forall w.\ S_w = R_w$ (the predicate $R(\underset{\sim}{X}) = \forall w.X_w = R_w$ is trivially continuous and satisfiable and the $\underset{\sim}{S}$-recursion is clearly constructive).

$$R(\underset{\sim}{X}) \Rightarrow \left\{ \begin{array}{l} ?x{:}T \to X_{\langle x \rangle} = ?x{:}T \to R_{\langle x \rangle} = R_{\langle\rangle} \\ \left(\begin{array}{l} ?x{:}T \to X_{\langle xy \rangle w} \\ \square\ !y \to X_w \end{array} \right) = \left(\begin{array}{l} ?x{:}T \to R_{\langle xy \rangle w} \\ \square\ !y \to R_w \end{array} \right) = R_{\langle y \rangle w} \end{array} \right\} \Rightarrow R(F(\underset{\sim}{X}))$$

This completes the proof that $R = S_{\langle\rangle}$, since by definition $R = R_{\langle\rangle}$.

The above illustrates one possible scheme of proof which is possible for processes defined in the way we are studying. We never needed the fact that the R-recursion is constructive, though this fact is quite easy to prove. To do this one shows that X, Y_x and Z respectively satisfy the conditions C_1^a, C_o^a and C_{-1}^a by mutual (recursion) induction. A tabular method for doing this is decribed in 6. .

Intuitively the master/slave operator is well adapted to the definition of stacks, just as "»" is well adapted to modelling buffers. It is easy to imagine how a process defined recursively by $R \Leftarrow (X \,||\, a::R)$ might act as a stack but harder to imagine how one might act as a buffer. This is because there must then be some leap-frogging of information. In the case of a stack information input from the environment is put first on the queue for re-output; this naturally ties in with the fact that the output of an "$R \Leftarrow (X \,||\, a::R)$" process is in the same place as its input. This is not the case with a buffer, where an element input from the environment must be put last on the queue for output. The next example shows how this might be done. In 6.8 the method adopted is to throw any input down to the bottom of the recursion, so that except when some input is being processed the process settles down to be like a queue of information waiting to get out: the further from the output/input port the longer its wait.

This method requires a slightly less constructive recursion: we can no longer induct on the amount contained in the buffer.

6.8 Buffer

This example is modelled closely on the previous one. The proof is rather more involved because of the rather less constructive recursion.

We use the same scheme of recursion:

$$R \Leftarrow (X \,||\, a::R)$$

where $X \Leftarrow ?x{:}T \to Y_x$

$Y_x \Leftarrow ?y{:}T \to a!y \to Y_x$

$\quad \Box\, !x \to Z$

$Z \Leftarrow ?x{:}T \to a!x \to Z$

$\quad \Box\, a?x{:}T \to Y_x$

It is easily shown by induction on their joint definition that X satisfies C^a_1, each Y_x satisfies C^a_0 and that Z satisfies C^a_{-1}. Thus the R-recursion above is constructive by 6.5(c).

Theorem $R = B^\infty$

Recall that $B^\infty = B^\infty_{\langle\rangle}$, where $B^\infty_{\langle\rangle} \Leftarrow ?x{:}T \to B^\infty_{\langle x\rangle}$

$$B^\infty_{w\langle y\rangle} \Leftarrow (?x{:}T \to B^\infty_{\langle x\rangle w\langle y\rangle}) \,\Box\, (!y \to B_w) \ .$$

The proof of this result depends on two inductions, one on the definition of B and one on the definition of R.

Firstly claim that (i) $\forall w.\ (Z \,||\, a::B^\infty_w) = B^\infty_w$

(ii) $B^\infty_{\langle\rangle} = (X \,||\, a::B^\infty_{\langle\rangle})$

$\forall w.\forall y.\ B^\infty_{w\langle y\rangle} = (Y_y \,||\, a::B^\infty_w)$.

To prove these results define $C \in M^{T^*}$ as follows:

$$C_{\langle\rangle} = (X \,||\, a::B^\infty_{\langle\rangle}) \ \underline{\text{or}}\ (Z \,||\, a::B^\infty_{\langle\rangle})$$

$$C_{w\langle y\rangle} = (Y_y \,||\, a::B^\infty_w) \ \underline{\text{or}}\ (Z \,||\, a::B^\infty_{w\langle y\rangle}) \ .$$

We then have (using many applications of rules 6.6)

$$\begin{aligned} C_{\langle\rangle} &= ?x{:}T \to (Y_x \,||\, a::B^\infty_{\langle\rangle}) \ \underline{\text{or}}\ ?x{:}T \to (a!x \to Z \,||\, a::(?x{:}T \to B^\infty_x)) \\ &= ?x{:}T \to ((Y_x \,||\, a::B^\infty_{\langle\rangle}) \ \underline{\text{or}}\ (Z \,||\, a::B^\infty_x)) \\ &= ?x{:}T \to C_x \end{aligned}$$

$$C_{w\langle y\rangle} = (!y \to (Z \,||\, a::B^\infty_w)) \,\Box\, (?x{:}T \to (a!x \to Y_y \,||\, a::(?x{:}T \to B^\infty_{\langle x\rangle w}(\Box\, *))))$$

$$\underline{\text{or}}\ ((?x{:}T \to ((a!x \to Z) \,||\, a::(?x{:}T \to B^\infty_{\langle x\rangle w\langle y\rangle}) \,\Box\, !y \to B^\infty_w)) \,\Box\, P) \ \underline{\text{or}}\ P)$$

where * (present if $w \neq \langle\rangle$) has the form $(!z \to B^\infty_v)$ and

$$\begin{aligned} P = (Y_y \,||\, a::B^\infty_w) &= ?x{:}T \to (a!x \to Y_y \,||\, a::(?x{:}T \to B^\infty_{\langle x\rangle w}(\,*)) \\ &\quad \Box\, !y \to (Z \,||\, a::B^\infty_w) \\ &= (?x{:}T \to (Y_y \,||\, a::B^\infty_{\langle x\rangle w})) \,\Box\, (!y \to (Z \,||\, a::B^\infty_w)) \end{aligned}$$

Hence

$$C_{w\langle y\rangle} = !y \to (Z \,||\, a::B^{\infty}_{w}) \,\Box\, ?x:T \to ((Y_y \,||\, a::B^{\infty}_{\langle x\rangle w}) \text{ or } (Z \,||\, a::B^{\infty}_{\langle x\rangle w\langle y\rangle}))$$
$$\sqsupseteq (!y \to C_w) \,\Box\, (?x:T \to C_{\langle x\rangle w\langle y\rangle})$$

Thus $\underset{\sim}{C} \sqsupseteq F(\underset{\sim}{C})$, where F is the function of the $\underset{\sim}{B}^{\infty}$-recursion. This implies that F has some fixed point below $\underset{\sim}{C}$ in the complete partial order M^{T^*}.
But fix(F) is maximal in M^{T^*}, since F is a constructive function with a unique fixed point which is a vector of deterministic processes (the fact that this is so is easily proved by induction on the definition of $\underset{\sim}{B}^{\infty}$).

Hence $\underset{\sim}{C} = \underset{\sim}{B}^{\infty}$, and this is easily seen to imply (i) & (ii), again because all the B^{∞}_{w} are maximal in M.

We thus have that $(X \,||\, a::B^{\infty}) = B^{\infty}$, and since the R-recursion is already known to be constructive it is an easy induction to show that $R = B^{\infty}$.

Notice that we have also shown that $B^{\infty}_{w} = (Z \,||\, a::B_{w})$, which shows that the recursion $Q \Leftarrow (Z \,||\, a::Q)$ cannot possibly be constructive, for we could then prove that $Q = B_w$ for each $w \in T^*$. This shows that 6.5(c) is as strong as possible, for Z satisfies C^{a}_{-1}.

6.9 Quicksort (After C.A.R.H.)
Suppose that T' is given some total order $<$, and $\mathbf{T} = T' \cup \{e,f\}$. It is possible to model a version of the "quicksort" algorithm using the same scheme of recursion used in the last two examples, except that here the process will call two independent versions of itself as slaves.

$$Q \Leftarrow ((X \,||\, u::Q) \,||\, d::Q)$$
$$\text{where} \quad X \Leftarrow (?x:T' \to Y_x) \,\Box\, (?e \to !f \to X)$$
$$Y_x \Leftarrow (?y:\{y \mid y>x\} \to u!y \to Y_x) \,\Box\, (?y:\{y \mid y\leq x\} \to d!y \to Y_x)$$
$$\Box\, (?e \to u!e \to d!e \to Z_x)$$
$$Z_x \Leftarrow (u?y:T' \to !y \to Z_x) \,\Box\, (u?f \to !x \to Z^*)$$
$$Z^* \Leftarrow (d?y:T' \to !y \to Z^*) \,\Box\, (d?f \to !f \to X)$$

The intention is that Q should receive a stream of input symbols terminated by the symbol "e" which is not in T' and then output them in descending order, ending the output stream by "f", another special symbol. Q goes about this

by using the first symbol in T' that it receives (assuming that the input stream is not empty) as a pivot. All the symbols subsequently are either sent to an "up" copy of Q if they are greater than the first, and to a"down"copy if not. These two copies of Q then sort the symbols by recursion. When the input stream of Q terminates, Q tells its slaves and they output their contents, sorted, to Q which relays this information to the environment. When the "up" slave (which is activated first) has finished its outputting, Q interrupts the two slaves by inserting the pivotal first element.

The way we will prove this implementation correct is to prove it congruent to a process which is defined in a simple way (like the S_w of 6.7) which it is easy to prove things about.

Define processes A_w, B_w ($w \in K$) as below, where K is the set of linearly ordered strings of T' (in descending order).

$$A_w = (?e \to B_w) \,\square\, (?x{:}T' \to A_{u(w,x)})$$
$$B_{\langle\rangle} = !f \to A_{\langle\rangle}$$
$$B_{\langle y\rangle w} = !y \to B_w$$

where $u{:}K\times T' \to K$ is defined

$$u(\langle\rangle, y) = \langle y\rangle$$
$$u(w\langle x\rangle, y) = w\langle xy\rangle \text{ if } x \geqslant y$$
$$= u(w,y)\langle x\rangle \text{ otherwise}$$

(That u is a well-defined function in $K\times T' \to K$ is easily proved by induction.)

The main result of this section will be to prove the theorem $Q = A_{\langle\rangle}$.

It is an easy induction on the definitions of X, Y_x, Z_x, Z^* to show that they satisfy (in that order)
$(C^u_o, C^u_{-1}, C^u_{-1}, C^u_1)$ and also $(C^d_1, C^d_o, C^d_o, C^d_{-1})$.

We thus know (by 6.5(b) and 6.4) that for all processes R&S $((X^* \,||\, u::R) \,||\, d::S) = ((X^* \,||\, u::S) \,||\, u::R)$ and that they are free of internal chatter, where X^* is any one of the four terms above.
It is also easily shown by 6.5(c) and (d) that the Q-recursion is constructive:

$$((X \,||\, u::R) \,||\, d::R)\restriction n+1 = ((X \,||\, u::R) \,||\, d::(R\restriction n))\restriction n+1$$

(by $C_1^d(X \,||\, u::R)$ which comes by 6.5(d))

$$= ((X \,||\, d::(R\restriction n)) \,||\, u::R)\restriction n+1 \quad \text{(by 6.4)}$$

$$= ((X \,||\, d::(R\restriction n)) \,||\, u::(R\restriction n))\restriction n+1$$

(by $C_0^u(X \,||\, d::(R\restriction n))$ which comes by 6.5(d))

$$= ((X \,||\, u::(R\restriction n)) \,||\, d::(R\restriction n))\restriction n+1$$

as required

Claim that $A_{\langle\rangle} = ((X \,||\, u::A_{\langle\rangle}) \,||\, d::A_{\langle\rangle})$

and $w\langle x\rangle v \in K \Rightarrow A_{w\langle x\rangle v} = ((Y_x \,||\, u::A_w) \,||\, d::A_v)$

and $w\langle x\rangle v \in K \Rightarrow B_{w\langle x\rangle v} = ((Z_x \,||\, u::B_w) \,||\, d::B_v)$

and $v \in K \Rightarrow B_v = ((Z^* \,||\, u::A_{\langle\rangle}) \,||\, d::B_v)$

This can be proved by several methods, including 5.15, but we will content ourselves by using the same device as that used in the last example:

let $C_{\langle\rangle} = ((X \,||\, u::A_{\langle\rangle}) \,||\, d::A_{\langle\rangle})$

$\langle\rangle \neq w \in K \Rightarrow C_w = \bigvee_{s\langle x\rangle t=w} ((Y_x \,||\, u::A_s) \,||\, d::A_t)$

$\langle\rangle \neq w \in K \Rightarrow D_w = \bigvee_{s\langle x\rangle t=w} ((Z_x \,||\, u::B_s) \,||\, d::B_t)$ or $((Z^* \,||\, u::A_{\langle\rangle}) \,||\, d::B_w)$

$D_{\langle\rangle} = ((Z^* \,||\, u::A_{\langle\rangle}) \,||\, d::B)$

Now $C_{\langle\rangle} = ?x:T' \to ((Y_x \,||\, u::A_{\langle\rangle}) \,||\, d::A_{\langle\rangle})$

$\Box ?e \to !f \to ((X \,||\, u::A_{\langle\rangle}) \,||\, d::A_{\langle\rangle})$

$= ?x:T' \to C_x$

$\Box ?e \to ((Z^* \,||\, u::A) \,||\, d::B_{\langle\rangle})$

$= (?x:T \to C_x) \,\Box\, (?e \to D_{\langle\rangle})$

and $C_w = ?y:T' \to (\bigvee_{\substack{s\langle x\rangle t=w \\ y>x}} ((u!y \to Y_x \,||\, u::A_s) \,||\, d::A_t)$ or

$\bigvee_{\substack{s\langle x\rangle t=w \\ y<x}} ((d!y \to Y_x \,||\, u::A_s) \,||\, d::A_t))$

$\Box ?e \to (\bigvee_{s\langle x\rangle t=w} ((u!e \to d!e \to Z_x \,||\, u::A_s) \,||\, d::A_t))$

$= ?y:T' \to (\bigvee_{\substack{s\langle x\rangle t=w \\ y>x}} ((Y_x \,||\, u::A_{u(s,y)}) \,||\, d::A_t)$ or

$\bigvee_{\substack{s\langle x\rangle t=w \\ y<x}} ((Y_x \,||\, u::A_s) \,||\, d::A_{u(t,y)}))$

$\Box ?e \to (\bigvee_{s\langle x\rangle t=w} ((Z_x \,||\, u::B_s) \,||\, d::B_t))$

Now $s\langle x\rangle t \in K$ and $y>x$ implies $u(s\langle x\rangle t, y) = u(s,y)\langle x\rangle t$ and also $y \leq x$ implies $u(s\langle x\rangle t, y) = s\langle x\rangle u(t,y)$, these facts being easy to show by definition of u.

Thus $C_w \sqsupseteq ?y:T' \to (\bigvee_{s\langle x\rangle t = u(w,y)} ((Y_x \,||\, u::A_s) \,||\, d::A_t))$

$\Box ?e \to (\bigvee_{s\langle x\rangle t=w} ((Z_x \,||\, u::B_s) \,||\, d::B_t))$

$\sqsupseteq (?y:T' \to C_{u(w,y)}) \,\Box\, (?e \to {}_w)$

Similarly (and more easily) it can be shown that

$$D_{\langle y\rangle^\frown w} \sqsupseteq\ !y \rightarrow D_w$$

and $D_{\langle\rangle} = !f \rightarrow C_{\langle\rangle}$.

The manipulations on the last page can be justified by repeated use of 6.4 and 6.6.

We thus have that $F(\underset{\sim}{C},\underset{\sim}{D}) \sqsubseteq (\underset{\sim}{C},\underset{\sim}{D})$, where F is the function of the combined $\underset{\sim}{A},\underset{\sim}{B}$-recursion. Hence, since it is an easy induction to show that all the A_w and B_w are deterministic, we must have that $\underset{\sim}{A} = \underset{\sim}{C}$ and $\underset{\sim}{B} = \underset{\sim}{D}$ (by the same argument as in the last example).

This gives us that $A_{\langle\rangle} = ((X \,||\, u::A_{\langle\rangle}) \,||\, d::A_{\langle\rangle})$, and since we have already seen that the Q-recursion is constructive this implies that $A_{\langle\rangle} = Q$, by induction.

By this result it is easy to prove that Q sorts its input correctly and is free of deadlock, amongst other things.

It is also possible, using exactly the same recursive scheme, to model the dual algorithm of quicksort, namely "shell sorting". Instead of pivoting on one of the elements of the input stream and recursively sorting the elements above and below the pivot, this works by splitting the input stream into two halves, sorting them and merging the two output streams. The process S below has this behaviour. It can be proved correct in very much the same way as Q (in fact $Q = S = A_{\langle\rangle}$).

$$S \Leftarrow ((X \,||\, a::S) \,||\, b::S)$$

where $X \Leftarrow (?x{:}T' \rightarrow (?e \rightarrow !x \rightarrow !f \rightarrow X) \,\square\, (?y{:}T' \rightarrow a!x \rightarrow b!y \rightarrow X_1)) \,\square\, (?e \rightarrow !f \rightarrow X)$

$X_1 \Leftarrow (?x{:}T' \rightarrow a!x \rightarrow X_2) \,\square\, (?e \rightarrow a!e \rightarrow b!e \rightarrow Y)$

$X_2 \Leftarrow (?x{:}T' \rightarrow b!x \rightarrow X_1) \,\square\, (?e \rightarrow a!e \rightarrow b!e \rightarrow Y)$

$Y \Leftarrow a?x{:}T' \rightarrow Y_x$

$Y_x \Leftarrow (b?y{:}\{y \mid y \geq x\} \rightarrow !y \rightarrow Y_x) \,\square\, (b?y{:}\{y \mid y < x\} \rightarrow !x \rightarrow Y'_y) \,\square\, (b?f \rightarrow !x \rightarrow Z_2)$

$Y'_x \Leftarrow (a?y{:}\{y \mid y \geq x\} \rightarrow !y \rightarrow Y'_x) \,\square\, (a?y{:}\{y \mid y < x\} \rightarrow !x \rightarrow Y_y) \,\square\, (a?f \rightarrow !x \rightarrow Z_1)$

$Z_1 \Leftarrow (b?f \rightarrow !f \rightarrow X) \,\square\, (b?x{:}T' \rightarrow !x \rightarrow Z_1)$

$Z_2 \Leftarrow (a?f \rightarrow !f \rightarrow X) \,\square\, (a?x{:}T' \rightarrow !x \rightarrow Z_2)$

Note that this process still makes a special case of the first symbol it inputs, even though it does not require it as a pivot. It is this special treatment of the first symbol, not sending it to a slave until a second symbol is input, which makes the recursion constructive. Intuitively this behaviour corresponds to not bothering to sort a list of one symbol, since any list of one symbol is already sorted. It is quite easy to show that X satisfies the conditions C_o^a and C_o^b.

Both these recursive algorithms require O(n) time and O(n) processors to sort a list of n symbols. It is obviously possible to refine the definitions of the processes to make them slightly more efficient, and if this were done the following observations would probably remain true. The Q algorithm has the advantage of requiring rather less processors and being more economical on data transmission (both because of the retention of the pivot). The S algorithm has the advantages that both the recursive structure generated in any run and the work load of any given processor are much more predictable (both these being because of the certain division of the input stream into two nearly equal halves, which does not always occur in Q because of the random nature of the pivot).

There is of course no reason why more complicated recursive structures should not be invoked when defining processes with the (A || a::B) operator. For a simple example of a mutual recursion let X be as it was in defining Q above and let Y be the "X" used in defining S above. Then each of the processes T & U defined below is equal to $S_{\langle\rangle}$.

T ⇐ ((X || u::T) || d::U)

U ⇐ ((Y || a::T) || b::U)

(This follows quite easily from what we already know, namely that the pair $(S_{\langle\rangle}, S_{\langle\rangle})$ is indeed a fixed point of the recursive function and that this function is constructive and so has a unique fixed point.)

There is a clear sense in which each of the recursions we have seen so far in this chapter using (A || a::B) can be thought of as defining a network of intercommunicating

processes. This network has the form of a directed tree in which the base node communicates with the environment and with its successors (slaves), and all other nodes communicate with their unique predecessors (masters) and their successors (slaves). This tree will normally be infinite and the real world is finite, so it would be unfortunate if in carrying out some finite computation an infinite portion of the tree were used. Consider the following example.

Let $X = ?x \rightarrow a!x \rightarrow \underline{abort}$ (x any element of T)
Trivially X satisfies C^a_o, and so the recursion

$A \Leftarrow (X \| a::A)$ is constructive (with value $?x \rightarrow \underline{abort}$).
However in the network which one expects to correspond with A (an infinite linear tree of "X"s) the simple act of communicating "?x" generates an infinite sequence of communications like an infinite row of toppling dominoes.

It is worrying that the condition C^a_o which is satisfied by the "X" above is exactly the same one which we used to prove some of our example processes correct. (We will in fact find that C^a_o is the only constructive C^a_i which allows this pathological behaviour.) All of our examples do in fact avoid this sort of behaviour, but there is no way of telling this from the final values of the processes. For example, consider the X of example 6.7; let $X^* = ?on \rightarrow a!on \rightarrow X$ and $X' = ?x{:}T \rightarrow a!on \rightarrow Y_x$ (where "on" is for the purposes of this example some symbol not in T but in the alphabet of the $(A \| a::B)$ operator). Now let $R^* \Leftarrow (X^* \| a::R^*)$ and $R' \Leftarrow (X' \| a:;R^*)$ be defined by recursion. Then each of X' and X* satisfies C^a_o (because X satisfies C^a_1) and $R = R'$ but the definition of R' gives rise to very much the same sort of pathological behaviour as the earlier example.

This sort of pathological behaviour clearly has a lot in common with "infinite internal chatter": both are caused by an infinite number of internal actions occurring in a process after it has only communicated finitely with its environment. The difference is that in the case we are now studying, let us call it <u>network chatter</u>, these actions are spread between an infinite number of distinct hiding operations and it is possible that no individual hiding operation (i.e. link between two processes) gives trouble.

Since we cannot detect the presence or otherwise of network chatter in a complete network of processes from its final value in our model we must try to find some criterion for deciding whether or not it is present from the values of the components of a network. For simplicity, and since it corresponds best with our experience, we will restrict our selves to the consideration of networks defined by recursions of the following type:

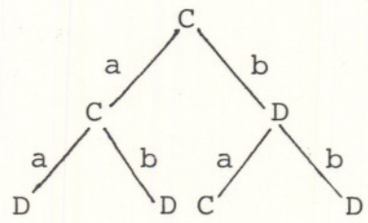
$$B_1 \Leftarrow ((\ldots((A_1 \,||\, a_1::B_1) \,||\, a_2::B_2)\ldots) \,||\, a_n::B_n)$$
$$B_2 \Leftarrow ((\ldots((A_2 \,||\, a_1::B_2) \,||\, a_2::B_2)\ldots) \,||\, a_n::B_n)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$B_n \Leftarrow ((\ldots((A_n \,||\, a_1::B_2) \,||\, a_2::B_2)\ldots) \,||\, a_n::B_n)$$

where each of the A_i contains no recursive calls of any of the B_j in its own definition and all of the names a_i are distinct. We will make two further simplifying assumptions, namely that it is impossible for any of the A_i to engage in infinite internal chatter with any slaves named $a_1,\ldots,a_n$ presented to it in any order, and that the above recursion has a unique fixed point. The justifications for these two assumptions are that firstly it is not worth considering the possibility of network chatter in a network if there is a possibility of the (worse?) condition infinite chatter between two components, and secondly that we need both of them to be able unambiguously to regard such a recursive definition as a network.

The following are a few remarks of a more general nature on the above recursive scheme. Fistly note that this scheme does not prevent one process calling two independent versions of itself or another process (as was done in the two sorting examples) since this can be achieved by inserting two identical A_is. Secondly note that the convention introduced above that the same slave is always addressed by the same name throughout the recursion is of little significance, since the names by which a process addresses its slaves in no way affects the final value of the combination (so long as a few simple rules are observed). Finally note that the fact that we might not want some of the B_i to call all of the other B_j as slaves does not matter either since $(A \,||\, a::C) = A$ if dom(A) contains no string with any symbols named by "a".

Remember that the n-tuple $\underline{B}$ is defined to be $\bigsqcup_{i=0}^{\infty} F^i(CHAOS^n)$, where $F(\underline{C})_i = ((\ldots(A_i \,|\!|\, a_1::C_1)\ldots) \,|\!|\, a_n::C_n)$ for $\underline{C} \in M^n$. Use the notation B_{ij} for the jth approximation $(F^j(CHAOS^n))_i$ to B_i. Because of our assumptions about freedom from infinite internal chatter we can unambiguously regard each B_{ij} as a finite tree of processes. This tree has a special form, which we must specify and annotate before we can continue.

Define (for A and X non-empty sets) an A-tree of depth n over X as follows. If n=o then it is a tree consisting solely of a base node, which is labelled by some element of X. If n=k+1 then it is a tree with base node labelled by some element of X, and from which there is a collection of edges, one labelled with each element of A, each edge leading to the base node of an A-tree of depth k over X. For example, the following is a {a,b}-tree of depth 2 over {C,D}.



base node
level 1 edges
level 1 nodes
level 2 edges
leaf nodes

In an A-tree of depth n one can specify a node by its co-ordinates, which are a string of elements of A with length n or less: if $\langle c_1..c_k \rangle$ is such a string then it specifies the node which is reached from the base by the path which consists of edges labelled $c_1,..,c_k$ (in that order). If T is such a tree and $\underline{a}$ such a string use the notation $T(\underline{a})$ for the node with co-ordinates $\underline{a}$ in T. If $n \geqslant 1$ and $a \in A$ denote by T[a] the tree of depth n-1 which is at the end of the level 1 edge labelled a.

With this notation we can think of B_{ij} as a $\{a_1,..,a_n\}$-tree of depth j over M. Define T_{ij}, the tree corresponding to B_{ij}, as follows: if j=o then $T_{ij}(\langle\rangle) = CHAOS$, otherwise $T_{ij}(\langle\rangle) = A_i$. If $\underline{a}$ has length j then $T_{ij}(\underline{a}) = CHAOS$; if $\underline{a}$ has non-zero length $<j$ with last component a_k then $T_{ij}(\underline{a}) = A_k$.

Because of the lack of infinite internal chatter it is easy to compound 6.2 to obtain the following result.

6.10 Lemma

For each $1 \leqslant i \leqslant n$ and $w \in \Sigma^*$ we have $w \in dom(B_{ij})$ if and only if there exists a $\{a_1,..,a_n\}$-tree of depth j over Σ^* with

the following properties:

(i) For each co-ordinate $\underline{a}$ we have $t(\underline{a}) \in dom(T_{ij}(\underline{a}))$

(ii) $t(\langle\rangle) \upharpoonright (\Sigma - (a_1.\Gamma \cup \ldots \cup a_n.\Gamma)) = w$

(iii) If $\underline{a}$ and $\underline{b}$ are two co-ordinates such that $\underline{a} = \underline{b}\langle a_k \rangle$ then $t(\underline{a}) \upharpoonright (\Sigma - (a_1.\Gamma \cup \ldots \cup a_n.\Gamma)) =$
$swap?!(strip(a_k)(t(\underline{b}) \upharpoonright a_k.\Gamma))$

(iv) If $\underline{a}$ is a co-ordinate of length j then <u>either</u> j=0 <u>or</u> $t(\underline{a}) \in \Gamma^*$

(In the above $\Gamma = T \cup ?T \cup !T$, as before.)

Say that such a tree is a tree for w in B_{ij}. Intuitively the tree t represents one possible way in which the processes which make up B_{ij} can co-operate to execute trace w. Say that t is a <u>perfect</u> tree if all its terminal nodes are labelled "$\langle\rangle$", and <u>imperfect</u> otherwise. A perfect t corresponds to all of the work being done by the "proper", fully defined components of the network; an imperfect t corresponds to the situation where some of the work is left to the improper "CHAOS" components of the tree.

Suppose t is a $\{a_1,..,a_n\}$-tree of depth r over Σ^* and $s \leq r$. Define $t \upharpoonright s$ to be the tree of depth s such that:
(i) if $\underline{a}$ has length $<s$ then $t \upharpoonright s(\underline{a}) = t(\underline{a})$, and
(ii) if $\underline{a}$ has length s then $t \upharpoonright s(\underline{a}) = t(\underline{a}) \upharpoonright (\Sigma - (a_1.\Gamma \cup \ldots \cup a_n.\Gamma))$.

The following result is a simple consequence of 6.10.

6.11 <u>Lemma</u>
a) If $k < j$ and t is a tree for w in B_{ij} then $t \upharpoonright k$ is a tree for w in B_{ik}.
b) If $k > j$ and t is a perfect tree for w in B_{ij} then $t \uparrow k$ is a tree for w in B_{ik}, where $t \uparrow k(\underline{a}) = t(\underline{a})$ if $|\underline{a}| \leq j$, $t \uparrow k(\underline{a}) = \langle\rangle$ otherwise.
c) If $j > 0$ and t is a tree for w in B_{ij} then $t[a_k]$ is a tree for some w' in B_{kj-1}. If t is perfect then so is $t[a_k]$ and at least one of the $t[a_k]$ is imperfect if t is.

Since $dom(B_i) = \bigcap_{j=0}^{\infty} dom(B_{ij})$ we clearly have by (b) above that the existence of a perfect tree for w at any level ensures that $w \in dom(B_i)$. If w has only imperfect trees in B_{ij} we cannot tell whether w $dom(B_i)$; but if $w \in dom(B_i)$ we can interpret the existence of imperfect trees for w in B_{ij} as signifying that B_i, while executing the string w,

might make use of its j+1st level. We can use this information to give a precise definition of network chatter in the B_is. Say that B_i admits network chatter on w if for each $m \in N$ there exists $r \geq m$ and tree t such that t is an imperfect tree for w in B_{ir}. Say that B_i contains network chatter if it admits it for any string w. There are several points worth noting about these definitions:

a) They apply only to the case of a recursion of the stated type with a unique fixed point and infinite chatter free components.

b) Because of the finite branching nature of the network of processes which makes up each B_i and the fact that T (the set of communicated symbols) is finite one can apply Konig's lemma to obtain the result that if B_i admits network chatter on w there exists a sequence $t_1, t_2, \ldots$ of trees with the properties that firstly each t_j is a tree for w in B_{ij} and $t_{ij+1} \upharpoonright j = t_{ij}$ and secondly that infinitely many of the t_j are imperfect. There is a clear sense in which one can think of such a nested sequence of trees as representing a single behaviour of B_i. In this sense a sequence in which infinitely many components are imperfect can be seen to represent a behaviour which includes network chatter (since it involves the use of infinitely many components of the network).

c) There is the possibility that there might exist an imperfect tree for some string w in B_{ij+1} when none exists in B_{ij}. This seems a little paradoxical as it implies that it is possible for B_i to call its j+2th level of recursively defined processes without seeming to call its j+1th level (the level which calls the j+2th level). The situation this represents is the spontaneous occurrence of communication between the j+1th and j+2th levels of processes without prompting by lower levels. This can certainly occur in the type of network we are considering, and while this type of behaviour cannot influence the external behaviour of the system it seems correct to include it in our consideration of network chatter.

The next stage of our work will be to seek sets of conditions which, if satisfied by the processes A_j, ensure that the B_i are free of network chatter. Our first aim will be to ensure that (under certain conditions) the "spontaneous communication" described above cannot occur. This will mean that all activity in the network is the result of chains of command originating at the base node. This will mean that we can to a large extent restrict ourselves to the study of these chains of command. By far the most convenient and practical condition which ensures our first end is the following EF (environment first):

$$EF(A) \Leftrightarrow \forall a \in \Sigma . \langle a \rangle \in dom(A) \Rightarrow a \in \Gamma$$

6.12 Lemma

If each of the A_i in our usual recursion satisfies the condition EF, then whenever t is an imperfect tree for some string w in B_{ij+1} $t\lceil j$ is an imperfect tree for w in B_{ij}. This means that "spontaneous communication", in the sense described above, is impossible.

The proof of this is an easy application of 6.11(a).

Intuitively the condition EF demands that the first communication of a process is with its master/environment and not with any of its slaves.

Note that if A is a process such that $dom(A) \subseteq (\Gamma \cup a_1.\Gamma \cup \ldots \cup a_n.\Gamma)^*$ and $C_o^a(A)$ holds for each j then EF(A) holds also.

Lemma 6.12 formalizes the idea that in studying any condition which is stronger than EF we need only worry about the activity which is in some sense the direct or indirect result of some communication with the environment. What we would like to find is some condition which ensures that all chains of command through the network are finite. The obvious interpretation of the word "command" here is the strings of symbols which pass between masters and their slaves. The obvious method for ensuring that all chains of command are finite is to verify that at all times and for each process in the network the commands given out to a process' slaves are strict reductions in some well-founded partial order of the command received from its own master. This idea is formalized in the next result.

6.13 Theorem

Suppose that $<'$ is some well-founded partial order on Σ^* with unique minimal element $\langle\rangle$. Suppose that the A_i in our usual recursive definition of the B_j all satisfy the following condition:

C) If $w \in dom(A)$ and $w \neq \langle\rangle$ then, for all $1 \leqslant j \leqslant n$

$w\lceil\Gamma >' swap?!(strip(a_j)(w\lceil a_j.\Gamma))$

then none of the B_j contain network chatter.

proof

Suppose for contradiction that the conditions of the theorem hold but that network chatter does exist in some of the B_i. We may suppose without loss of generality that w is minimal in the p.o. $<'$ with respect to giving rise to network chatter in any of the B_i and that B_r is one of the B_i which admits network chatter on w.

Observe first that condition C above implies EF since if $\langle a\rangle \in dom(A)$ we must have $a \in \Gamma$, for otherwise (as $<'$ has unique minimal element $\langle\rangle$) the inequality in C would not be satisfied for any j if this were not so.

By assumption there exists some imperfect tree t for w in B_{rj} for some $j \geqslant 1$. Applying 6.12 j-1 times we see that there must be an imperfect tree t' for w in B_{r1}. This t' consists of a base node (labelled $v \in dom(A_r)$, say) and n leaf nodes (labelled $v_1,..,v_n$, say). Since t' is imperfect there is some j s.t. $v_j \neq \langle\rangle$. This is easily seen to imply that $v \neq \langle\rangle$, so we can deduce that $w\ (=v\lceil\Gamma) \neq \langle\rangle$ (by $EF(A_r)$).

By definition of network chatter in B_r there must exist some infinite sequence of trees $t_1, t_2, ..$ with the properties that each t_i is an imperfect tree for w in some $B_{rj(i)}$, and that the resulting sequence j(i) is strictly increasing. By 6.12 and the fact that $EF(A_j)$ holds for each j we can assume that each t_i is an imperfect tree for w in B_{ri}. By 6.11(c) there must be, for each i, some j such that $t_i\lceil a_j\rceil$ is imperfect. One j at least must be repeated infinitely; we can therefore assume (applying 6.12 once again) that there is some j such that all the t_i have $t_i\lceil a_j\rceil$ imperfect. Each one of these is a tree for some w_i in B_{ji-1}. If there were infinitely many possible values w_i then

infinite internal chatter would be possible in the process $((..((A_r \| a_1::C)..a_{j-1}::C) \| a_{j+1}::C)..a_n::C) \| a_j::C)$ at the highest level, where C is an abbreviation for CHAOS. This would contradict our assumptions on the nature of the A_i. We may therefore conclude that there is some w* with the property that infinitely many of the $t_i[a_j]$ are imperfect trees for w* in B_{ji-1}. This tells us that B_j admits network chatter on w*.

However by construction there exists some $v \in dom(A_r)$ (for example any of the base nodes of the t_i such that $t_i[a_j]$ is a tree for w*) such that $w = v\restriction(\Sigma - (a_1.\Gamma \cup .. a_n.\Gamma))$ $w^* = swap?!(strip(a_j)(v\restriction a_j.\Gamma))$. Since condition C holds of A_r we can infer that $w^* <' w$, but this contradicts our assumption that w is minimal with respect to giving rise to network chatter in any of the A_is.

We may therefore conclude that network chatter is impossible in the processes B_j as claimed.

The above result can be strengthened slightly: in place of a partial order on Σ^* one can instead use a partial order on $\Sigma^* \times \{a_1,..,a_n\}$ with joint minimal elements $(\langle\rangle,a_1),..,(\langle\rangle,a_n)$ and require that if process A_i sends command w' to its a_j-slave when it has itself received command w then we must have $w=\langle\rangle$ or $(w,a_i) > (w',a_j)$. The proof of this strengthened result is a simple adaptation of the above.

Having established a set of conditions which ensure freedom from network chatter we should check to see that each of the examples seen earlier (6.7, 6.8, 6.9) is free from it. The one obvious difficulty here is that all our definitions and results apply only to the standard form of recursion which was set out earlier, and that the two sorting processes do not conform to that pattern since they make multiple calls of identical slaves. As was stated earlier it is possible to recast such recursions into our canonical form by defining two processes by mutual recursion with identical "master" processes. It is easy to see that the various approximating trees (T_{ij}) would be identical in the two versions of such a process. It is of course possible to extend all our definitions and proofs to the case of "multiple calls" with the penalty of requiring more complex notation. When one does

this is the natural way it is not hard to show that network chatter exists in a process with "multiple calls" if and only if it exists in the version of the process recast in our canonical form in the natural way.

As suggested earlier our task is simplest in the cases of 6.7 and 6.8 because the "X"s used satisfy C_1^a. That this implies the freedom from network chatter of the stack R and buffer R is a consequence of the following result, itself a corollary to 6.13.

6.14 Theorem

If L is some set of labels and Π some subset of Γ define the condition D_Π^L as follows:

$\forall a \in L.\forall w \in dom(A).\ w \neq \langle\rangle \Rightarrow |w\restriction\Pi| > |swap?!(stripa(w\restriction a.\Gamma))\restriction\Pi|$.

Suppose that in our usual recursion each of the processes $A_1,..,A_n$ satisfies D_Π^L, where $L = \{a_1,..,a_n\}$, then each of the processes B_j defined by the usual recursive scheme is free of network chatter.

The partial order used in applying 6.13 to prove this is the one defined $w > v$ if <u>either</u> $|w\restriction\Pi| > |v\restriction\Pi|$ <u>or</u> $w \neq \langle\rangle$ & $v = \langle\rangle$ (or both).

It it easy to see that if a process satisfies each of C_1^a, with "a" ranging over a set of labels L, and if further the process satisfies $dom(A) \subseteq (\cup\{\Gamma, a.\Gamma \mid a \in L\})^*$ then it satisfies D_Γ^L. It is this fact which tells us that the definitions of the stack and buffer are free of network chatter.

It remains to show that the two sorting examples are free from network chatter. One way in which one might seek to show a recursion to be "well-defined" is to check that no process <u>outputs</u> as much to its slaves as it has <u>input</u> from its environment. Indeed a close examination of each of the "X"s used in 6.9 will reveal that at no time have they performed as many "a!x"s (outputs to slaves) as they have performed "?x"s (inputs from environment). This suggests that ?T might be a good "Π" to use in applying 6.14. This is in fact so; if we introduce the condition E^L as below it is easy to show that $E^L = D_{?T}^L$ and that a suitable E^L is satisfied by each of the "X" processes of 6.9.

$E^L(A) \Leftrightarrow \forall a \in L.\forall w \in dom(A).\ w \neq \langle\rangle \Rightarrow |w\restriction ?T| > |w\restriction a!T|$ $(\Leftrightarrow \forall a \in L.E^a(A)$, say)

It is easy to see that all conditions of the form D_{Π}^{L} are strongly continuous. However unlike the C_i^a they do not in themselves imply freedom from infinite internal chatter with arbitrary processes (or even other D_{Π}^{L}-processes) as slaves. If this can be done by some other means there is though a class of restriction operators closely related to D_{Π}^{L} which has an interesting and useful theory in relation to master/slave recursion via D_{Π}^{L}-processes. This class of operators is the one where $A\lceil n$ behaves like A until it has communicated n elements of π and then dissolves into CHAOS. We will not follow up this subject here however.

Let us sum up this section. Having identified a certain unfortunate type of possible behaviour in recursively defined networks we discovered that its presence did not appear to influence the external behaviour of a network. We established a plausible formal definition of network chatter over the class of recursive definitions which we could most easily regard as networks, and in that case derived laws which could be shown to imply that it was absent (in the formally defined sense). One might choose to regard the insensitiveness of our model to this condition as a weakness in the model; this and other related problems will be discussed in chapter eight. The crucial fact about the type of recursive tree discussed in the latter part of this chapter was that it could be thought of as the limit of a sequence of finite networks. It should be reasonably easy to extend the ideas used to other infinite networks with this property, provided that one could prove that the notions of network chatter introduced were well-defined in that they were independent of the convergent sequence of networks used. (This is quite easy to do for the networks we have already met.) It is not quite so clear what one should do in cases (common in chapter five) where recursive calls of processes are made not only through parallel operators but also guarded (5.31 for example). Intuitively these recursions define networks which are of a less clear-cut type, but which somehow appear to be less prone to this type of behaviour. Again one could possibly define corresponding trees of processes (of a less simple nature) relative to which one could define and avoid infinite chains of recursive calls.

Postscript: Proving C_i^a and E^L of simple processes.

The conditions C_i^a and E^L admit a fairly simple and mechanical method of proof which can be applied to processes defined by tail recursion. Say that a process is defined by tail recursion if it is written in the form:

$$B_\lambda \text{, where } \quad \begin{array}{l} \zeta \in \Gamma_1 \Rightarrow B_\zeta \Leftarrow A_1 \\ \quad \cdots\cdots \\ \quad \cdots\cdots \\ \zeta \in \Gamma_s \Rightarrow B_\zeta \Leftarrow A_s \end{array}$$

and each of the A_i is formed from the syntax consisting of "skip", "abort", "a →", "x:U →", "a.x:U →" and "□" together with recursive calls of the B_λ, each of which has the property that it can be syntactically deduced exactly which of the Γ_i it must inevitably fall in. Note that each of the "X"s used in examples 6.7 - 6.9 is of this form.

If "a" is a label define the condition E_i^a as follows:

$$E_i^a(A) \Leftrightarrow \forall w \in \mathrm{dom}(A).\ |w\restriction ?T| \geq \min(|w\restriction a!T|+i, |w\restriction (?T \cup a!T)|)$$

This condition is plainly both strongly continuous and of a very similar form to C_i^a. It is easy to see that $E^L(A) \Leftrightarrow (\forall a \in L.E_1^a(A))\ \&\ A^o \subseteq ?T$. This means that any method we develop to prove the E_i^a can be used to prove the E^L.

In order to develop our method we will need a list of technical lemmas, which we will later combine to produce inductive proofs.

6.15 Lemma

a) Suppose that A & B are processes satisfying C_i^a and C_j^a respectively, then

(i) $C_k(c \to A)$ holds, where $k = i+1$ if $c \in \Gamma$
$= \min(-1, i-1)$ if $c \in a.\Gamma$
$= i$ otherwise;

(ii) $C_k(A \,\square\, B)$ holds, where $k = \min(i,j)$.

b) Suppose that A & B are processes satisfying E_i^a and E_j^a respectively, then

(i) $C_k(c \to A)$ holds, where $k = i+1$ if $c \in ?T$
$= \min(-1, i-1)$ if $c \in a!T$
$= i$ otherwise;

(ii) $C_k(A \,\square\, B)$ holds, where $k = \min(i,j)$.

c) C_i^a and E_i^a always hold of skip and abort (because "✓" can never be in T).

6.16 Lemma

a) Suppose that $U \subseteq T$ and that for each $x \in U$ we have $C^a_i(A_x)$, then

(i) $C^a_{i+1}(x{:}U \to A_x)$, $C^a_{i+1}(!x{:}U \to A_x)$, $C^a_{i+1}(?x{:}U \to A_x)$;

(ii) $C^a_i(b.x{:}U \to A_x)$ if $b \notin \{a, a?, a!, ?, !\}$;

(iii) $C^a_j(a.x{:}U \to A_x)$, $C^a_j(a!x{:}U \to A_x)$, $C^a_j(a?x{:}U \to A_x)$, where $j = \min(-1, i-1)$.

b) Suppose that $U \subseteq T$ and that for each $x \in U$ we have $E^a_i(A_x)$, then

(i) $E^a_{i+1}(?x{:}U \to A_x)$;

(ii) $E^a_j(a!x{:}U \to A_x)$, where $j = \min(-1, i-1)$;

(iii) $E^a_i(b.x{:}U \to A_x)$, $E^a_i(x{:}U \to A_x)$ if $b \notin \{?, a!\}$

Now suppose that we have a process which is defined by tail recursion, and that it is written

$$B \text{ , where } \begin{array}{l} \zeta \in \Gamma_1 \Rightarrow B_\zeta \Leftarrow A_1 \\ \quad . \; . \; . \; . \; . \; . \\ \quad . \; . \; . \; . \; . \; . \\ \zeta \in \Gamma_s \Rightarrow B_\zeta \Leftarrow A_s \end{array} .$$

One will often be able to find upper bounds for the strengths of conditions satisfied by some of the above clauses from their lowest recursive level (by application of the "min" clauses). For example, recalling the definition of Y_x in the buffer example $Y_x \Leftarrow (?y{:}T \to a!x \to Y_x) \,\square\, (!x \to Z)$, it is clear that Y_x cannot satisfy either of C^a_1 or E^a_1. Using knowledge of this type and one or two iterations of the recursion one can extend these bounds throughout the recursion. Thus in the buffer example we cannot expect "X" to satisfy C^a_2 because it is defined $X \Leftarrow ?x{:}T \to Y_x$ and Y_x never satisfied C^a_1.

Suppose now that one has developed a hypothesis of the form <u>either</u> $(\zeta \in \Gamma_i \Rightarrow C^a_{k(i)}(B_\zeta))$ <u>or</u> $(\zeta \in \Gamma_i \Rightarrow E^a_{k(i)}(B_\zeta))$. A <u>test</u> of such a hypothesis will be the action of assuming that it is true of the vector $\underset{\sim}{B}$ and seeing if it remains true of $F(\underset{\sim}{B})$, where F is the function associated with the $\underset{\sim}{B}$-recursion. Such a test can be carried out very easily because of our assumption that each recursive call must fall within exactly one of the Γ_i, and by Lemmas 6.15 and 6.16.

It is also extremely easy to check that recursions of our given form are constructive (for example such recursions are constructive if all recursive calls are guarded). Since all predicates of the form C^a_i and E^a_i are trivially satisfiable the following is a trivial application of our inductive principle.

6.17 Theorem

If in a constructive tail recursion of the form given above we have a hypothesis either of the form $(\zeta \in \Gamma_i \Rightarrow C^a_{k(i)}(A_\zeta))$ or $(\zeta \in \Gamma_i \Rightarrow E^a_{k(i)}(A_\zeta))$ which is testable then we can infer that it holds of the vector $\underset{\sim}{B}$.

6.18 Example

To prove E^u_1 and E^d_1 of the "X" used in 6.9 (Quicksort).

step 1

Recall our recursive definition:

$$X \Leftarrow (?x{:}T' \to Y_x) \,[]\, (?e \to !f \to X)$$

$$Y_x \Leftarrow (?y{:}\{y \mid y \geqslant x\} \to u!y \to Y_x) \,[]\, (?y{:}\{y \mid y < x\} \to d!y \to Y_x) \,[]\, (?e \to u!e \to d!e \to Z_x)$$

$$Z_x \Leftarrow (u?y{:}T' \to !y \to Z_x) \,[]\, (u?f \to !x \to Z^*)$$

$$Z^* \Leftarrow (d?y{:}T' \to !y \to Z^*) \,[]\, (d?f \to !f \to X)$$

It is clear that we cannot expect Y_x to satisfy any stronger E^u condition than E^u_0 or any stronger E^d condition than E^d_0. By substituting this into X we see that E^u_1 and E^d_1 are the maximum conditions satisfied by X, and also by Z* and Z_x. Let us therefore take as our hypothesis that X satisfies E^u_1 and E^d_1, each Y_x satisfies E^u_0 and E^d_0, each Z_x satisfies E^u_1 and E^d_1, and that Z* satisfies E^u_1 and E^d_1. We will just check the E^u-hypothesis since the other is very similar.

step 2

Assume that the vector $(X', \underset{\sim}{Y}', \underset{\sim}{Z}', Z^{*\prime})$ satisfies our E^u hypothesis. Then we get

$$\begin{array}{l|l} E^u_0(Y'_x) \quad (x \in T') & E^u_1(X') \\ \Rightarrow \; E^u_1(?x{:}T' \to Y'_x) & \Rightarrow E^u_1(!f \to X') \\ & \Rightarrow E^u_2(?e \to !f \to X') \end{array}$$

$$\Rightarrow E^u_1((?x{:}T' \to Y'_x) \,[]\, (?e \to !f \to X')) \quad \text{as required.}$$

. . . . . .

$$\begin{array}{l|ll} E^u_0(Y'_x) & E^u_0(Y'_x) & (x \in T') \\ \Rightarrow E^u_{-1}(u!y \to Y'_x) & = E^u_0(d!y \to Y'_x) & (x, y \in T') \end{array}$$

$\Rightarrow E_o^u(?y:\{y|y\geq x\} \rightarrow u!y \rightarrow Y'_x) \;\Big|\; \Rightarrow E_1^u(?y:\{y|y<x\} \rightarrow d!y \rightarrow Y'_x)$

$\Rightarrow E_o^u((?y:\{y|y\geq x \rightarrow u!y \rightarrow Y'_x) \,\square\, (?y:\{y|y<x\} \rightarrow d!y \rightarrow Y'_x))$

and $E_1^u(Z'_x)$

$\Rightarrow E_1^u(d!e \rightarrow Z'_x)$

$\Rightarrow E_o^u(u!e \rightarrow d!e \rightarrow Z'_x)$

$\Rightarrow E_1^u(?e \rightarrow u!e \rightarrow d!e \rightarrow Z'_x)$

$= E_o^u(((?y:\{y|y\geq x\} \rightarrow u!y \rightarrow Y'_x) \,\square\, (?y:\{y|y<x\} \rightarrow d!y \rightarrow Y'_x))$
$\square (?e \rightarrow u!e \rightarrow d!e \rightarrow Z'_x))$

as required.

. . . . . .

$E_1^u(Z'_x)$ | $E_1^u(Z*')$ $(x \in T')$

$\Rightarrow E_1^u(!y \rightarrow Z'_x)$ | $\Rightarrow E_1^u(!x \rightarrow Z*')$

$\Rightarrow E_1^u(u?y:T' \rightarrow !y \rightarrow Z'_x)$ | $\Rightarrow E_1^u(u?f \rightarrow !x \rightarrow Z*')$

$= E_1^u((u?y:T' \rightarrow !y \rightarrow Z'_x) \,\square\, (u?f \rightarrow !x \rightarrow Z*'))$ as required

. . . . . .

$E_1^u(Z*')$ | $E_1^u(X')$

$\Rightarrow E_1^u(!y \rightarrow Z*')$ | $\Rightarrow E_1^u(!f \rightarrow X')$

$\Rightarrow E_1^u(d?y:T' \rightarrow !y \rightarrow Z*')$ | $\Rightarrow E_1^u(d?f \rightarrow !f \rightarrow X')$

$\Rightarrow E_1^u((d?y:T' \rightarrow !y \rightarrow Z*) \,\square\, (d?f \rightarrow !f \rightarrow X'))$ as required

The above, together with the fact that the recursion is constructive, tells that $E_1^u(X)$ holds as desired.

These proofs can be recast in a tabular form, assigning a number to each point in the syntax of a process. For example the second clause of the above proof could be re-written

$((?y:\{y|y\geq x\} \rightarrow u!y \rightarrow Y'_x) \,\square\, (?y:\{y|y<x\} \rightarrow d!y \rightarrow Y'_x))$

o -1 o o 1 o o

$\square (?e \rightarrow u!e \rightarrow d!e \rightarrow Z'_x))$

$\underline{o}$ 1 o 1 1

(The underlined "o" represents the highest syntactic level.)

Chapter 7 :- Alternative Parallel Combinators

In the first half of this chapter we will briefly study the theories of a few more parallel/hiding combinators. In the second half we will examine the important problem of how one can prove networks of processes free from deadlock.

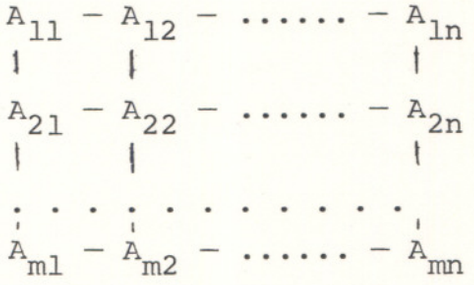
The following is a list of a few possible parallel/hiding combinators which we might wish to use.

a) A bidirectional "pipe" operator "$\aleph$" in which processes are connected very much in the same way as in the old pipe "$\gg$". A process will now though be expected to be able to input and/or output down two named channels "l" and "r"; the "left" process will have its right hand ("r") outputs and inputs connected to the left hand ("l") inputs and outputs of the other "right" process.

$$(A \aleph B) = (\text{swap?!}(\text{stripr}(A))_X ||_Y \text{stripl}(B)))/(?T \cup !T),$$
$$\text{where } X = l!T \cup l?T \cup ?T \cup !T$$
$$Y = r!T \cup r?T \cup ?T \cup !T$$

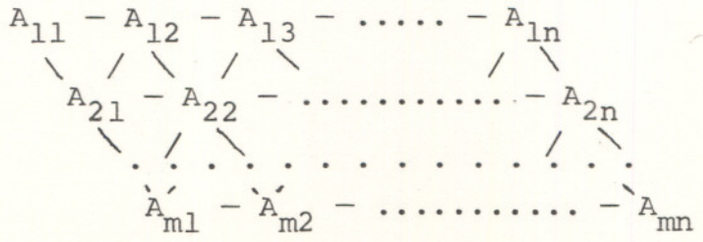
b) $(A_X \leftrightarrow_Y B) = (A_X ||_Y B)/(X \cap Y)$, the operator in which two processes running ordinarily in parallel have all their intercommunication hidden.

c) $[A_{ij}]_{m,n}$, in which an mxn matrix of processes operate in parallel, each communicating with its four immediate neighbours.



For a precise definition of this operator see later.

d) $\{A_{ij}\}_{m,n}$, in which an mxn matrix of processes is arranged in a hexagonally connected array instead of the square above.



To some extent the operator "↔" defined in (b) above is the most basic operator of its type, all the others being derived from it and alphabet transformers. Observe how both of our old operators may be defined:

$$A \gg B = (\text{strip}!A_{T\cup?T} \leftrightarrow_{T\cup!T} \text{strip}?B)$$

$$(A \,||\, a::B) = (A_{\Sigma} \leftrightarrow_{a,r} a.\text{swap}!?(B)) \ .$$

The conditional associativity of "≫" and the conditional commutativity of $(A \,||\, a::B)$ can both be deduced from the above definitions and the following lemma, which is as one might expect a consequence of 5.35.

7.1 Lemma

If $(A_X \leftrightarrow_Y B)$ and $(B_Y \leftrightarrow_Z C)$ are both free of infinite internal chatter (in the obvious sense) and $X \cap Y \cap Z = \emptyset$ then

$$((A_X \leftrightarrow_Y B)_{X\cup Y} \leftrightarrow_Z C) = (A_X \leftrightarrow_{Y\cup Z} (B_Y \leftrightarrow_Z C)) \ .$$

proof

Suppose that the hypotheses of the lemma hold, then

$$\begin{aligned}
((A_X \leftrightarrow_Y B)_{X\cup Y} \leftrightarrow_Z C) &= ((A_X \,||_Y B)/(X\cap Y)_{X\cup Y} \,||_Z C)/((X\cup Y)\cap Z) \\
&= ((A_X \,||_Y B)_{X\cup Y} \,||_Z B)/((X\cap Y)\cup(X\cap Z)\cup(Y\cap Z)) \quad (5.35) \\
&= (A_X \,||_{Y\cup Z} (B_Y \,||_Z C))/((X\cap Y)\cup(X\cap Z)\cup(Y\cap Z)) \\
&= (A_X \,||_{Y\cup Z} (B_Y \,||_Z C)/(Y\cap Z))/(X\cap(Y\cup Z)) \quad (5.35) \\
&= (A_X \leftrightarrow_{Y\cup Z} (B_Y \leftrightarrow_Z C)) \quad \text{as desired.}
\end{aligned}$$

The "meaning" of this lemma is that in a network so long as there can be no confusion about the destination of any message ($X\cap Y\cap Z = \emptyset$) and there is no infinite chatter, it does not matter how the network was constructed.

Let us study the structure of the combinator $[A_{ij}]_{mn}$ introduced in (c) above. It is fundamentally different from our other ones in that the structures it creates are not normally trees, and therefore it is likely to introduce loops. So far we have not specified the exact nature of the operator; we will expect it to achieve the following:

a) Each A_{ij} will have four channels u,d,l,r. The "u" channel of $A_{i+1,j}$ will be connected to the "d" channel of A_{ij}, and the "r" channel of A_{ij} will be connected to the "l" channel of $A_{i,j+1}$ (all internal communication being hidden).

b) The row of n accessible "u" channels will be addressed by the names u.i ($i \in \{1,..,n\}$), the m accessible "l" channels by the names l.i ($i \in \{1,..,m\}$), etc.

c) The communications down each channel will be in T, the usual finite set of unnamed symbols. (If desired the operators we define can easily be adapted to assuming communication in ?T∪!T with inputs being connected to outputs and vice-versa.)

From our experience with other operators it appears that there is some ambiguity left in this definition, this resulting from the many possible orders of putting such a network together and hiding the internal communication. We might also expect this ambiguity to disappear when the network is free of infinite internal chatter, as we should then be in a position to apply 7.1.

It is clear that it is possible to construct arbitrarily large matrices with the following combinators.

7.2 Definitions

(i) Define $X(a,b,c,d,r,s,t,u) = \bigcup_{i=r}^{s}(a.i.T \cup b.i.T) \cup \bigcup_{i=t}^{u}(c.i.T \cup d.i.T)$ for labels a,b,c,d and integers r,s,t,u.

(ii) $[A] = swapu(u.1)(swapd(d.1)(swapl(l.1)(swapr(r.1)(A))))$
This is the combinator which produces a 1x1 matrix from a single process with u,d,l,r channels.

(iii) ${}_{n}[A|B]_{s}^{m} = (swaprb(A)_{Y \leftrightarrow Z}swaplb(inc\{u,d\}(r,n)(B)))$ where
$Y = X(l,b,u,d,1,m,1,n)$, $Z = X(b,r,u,d,1,m,n+1,n+s)$,
"b" is a label distinct from u,d,l and r, and $inc\{e,f\}(t,s)(A)$
$= swap(e.1)(e.s+1)(swap(f.1)(f.s+1)(..swap(f.t)(f.t+s)(C))..)$.

(iv) $\left[\frac{A}{B}\right]_{s}^{m}{}_{n} = (swapda(A)_{Y \leftrightarrow Z}swapua(inc\{l,r\}(s,m)(B)))$, where
$Y = X(l,r,u,a,1,m,1,n)$, $Z = X(l,r,a,d,m+1,m+s,1,n)$,
and "a" is a label distinct from b,u,d,l and r.

These combinators join blocks together, joining mxn and mxs blocks to make a mx(n+s) matrix, and mxn and sxn blocks to make a (m+s)xn matrix respectively.

In future we will habitually suppress the dimension parameters (n,m,s above) when they are obvious from their context. One might think that it is possible to drop the "central" parameter (m in (iii) and n in (iv)), but if we were to do this it would be necessary to hide an infinite alphabet (there being no bound on the possible integer part of the labels of communications).

There are clearly many ways in which one could produce a

definition of $[A_{ij}]_{nm}$ from these three combinators. For example, the number of different ways of producing a 1xn or a nx1 matrix is $\frac{1}{n}\binom{2n-2}{n-1}$, and the first few terms in the table of the number of ways of producing a nxm matrix are shown below.

m \ n	1	2	3	4	5	6
1	1	1	2	5	14	42
2	1	2	8	45	318	2644
3	2	8	64	770	13008	290544
4	5	45	770	19450	729148	41031312
5	14	318	13008	729148	57378464	7222570064
6	42	2644	290544	41031312	7222570064	****

where **** = 1816170558336

The generating relations of this table are

$$t_{11} = 1$$

$$i \neq 1 \text{ or } j \neq 1 \Rightarrow t_{ij} = \sum_{k=1}^{j-1} t_{ik} \cdot t_{ij-k} + \sum_{k=1}^{j-1} t_{kj} \cdot t_{i-kj} \quad .$$

In proving the (conditional) independence of the final value from the method of construction the following three lemmas are vital.

7.3 Lemma

If A,B,C represent mxn, rxn and sxn matrices respectively (i.e. if their alphabets are consistent with this) and each of $\left[\frac{A}{B}\right]$ and $\left[\frac{B}{C}\right]$ is free of infinite internal chatter then

$$\left[\frac{\left[\frac{A}{B}\right]}{C}\right] = \left[\frac{A}{\left[\frac{B}{C}\right]}\right] .$$

7.4 Lemma

If A, B, C represent nxm,nxr and nxs matrices respectively and each of [A|B] and [B|C] is free of infinite internal chatter then [[A|B]|C] = [A|[B|C]].

7.5 Lemma

If A,B,C,D represent nxm, nxs, txm and txs matrices respectively and each of [A|B], [C|D], $\left[\frac{A}{C}\right]$, $\left[\frac{B}{D}\right]$, $\left[\frac{[A|B]}{[C|D]}\right]$ and $\left[\left[\frac{A}{C}\right]\left[\frac{B}{D}\right]\right]$ is free of infinite internal chatter, then

$$\left[\frac{[A|B]}{[C|D]}\right] = \left[\left[\frac{A}{C}\right]\left[\frac{B}{D}\right]\right].$$

proof

The proofs of 7.3 and 7.4 are very similar to the proof of 5.19 (the associativity of "$\gg$") for obvious reasons, We will therefore content ourselves with a sketch proof of 7.5 (a result which is another elaborate corollary to lemma 7.1).

$$\left[\frac{[A|B]}{[C|D]}\right] = (\text{swapda}[A|B]_{Y\leftrightarrow Z}\text{swapua}(\text{inc}\{1,r\}(t,n)[C|D]))$$

where $Y = X(1,r,u,a,1,n,1,m+s)$, $Z = X(1,r,a,d,n+1,n+t,1,m+s)$

$$= ((A^*_{U\leftrightarrow V}B^*)_{Y\leftrightarrow Z}(C^*_{W\leftrightarrow R}D^*)), \qquad (*)$$

where $A^* = \text{swapda}(\text{swaprb}(A))$

$B^* = \text{swapda}(\text{swaplb}(\text{inc}\{u,d\}(s,m)(B)))$

$C^* = \text{swapua}(\text{inc}\{1,b\}(t,n)(\text{swaprb}(C)))$

$D^* = \text{swapua}(\text{inc}\{b,r\}(t,n)(\text{swapbl}(\text{inc}\{u,d\}(s,m)(D))))$

$U = X(1,b,u,a,1,n,1,m)$, $V = X(b,r,u,a,1,n,m+1,m+s)$

$W = X(1,b,a,d,n+1,n+t,1,m)$, $R = X(b,r,a,d,n+1,n+t,m+1,m+s)$.

Now $U\cup V \supseteq Y$ and $W\cup R \supseteq Z$, and so it is easy to see that $(U\cup V)$ and $(W\cup R)$ can be substituted for Y and Z in (*) above without changing the value.

Also $(U\cup V)\cap W\cap R = \emptyset$, and since infinite internal chatter is absent by assumption we get that (*) is equal to

$$((A^*_{U\leftrightarrow V}B^*)_{U\cup V\leftrightarrow W}C^*)_{U\cup V\cup W\leftrightarrow R}D^*) \qquad \text{by 7.1}$$

$$= (((A^*_{U\leftrightarrow W}C^*)_{U\cup W\leftrightarrow V}B^*)_{U\cup V\cup W\leftrightarrow R}D^*) \qquad " \quad "$$

$$= ((A^*_{U\leftrightarrow W}C^*)_{U\cup W\leftrightarrow V\cup R}(C^*_{V\leftrightarrow R}D^*)) \qquad " \quad "$$

This is readily shown to be equal to $\left[\left[\frac{A}{C}\right]\left[\frac{B}{D}\right]\right]$, as desired.

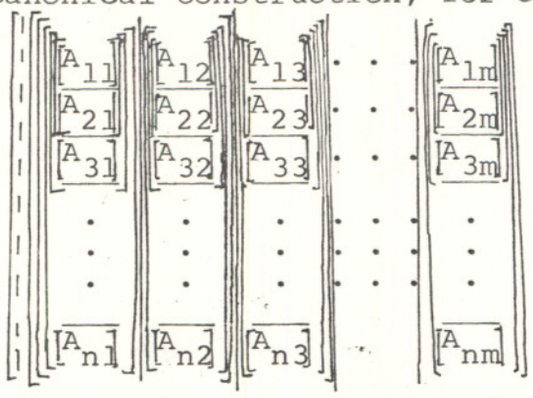
(The various manipulations required to fill out this outline proof are easy but tedious.)

Having established these results we can now prove that under certain conditions we can disregard the order of construction of matrices.

7.6 Lemma

Suppose that $\{A_{ij} \mid 1\leq i\leq n,\ 1\leq j\leq m\}$ is a set of processes such that every construction of a (not necessarily proper) sub-matrix of A_{ij} using the constructs of 7.2 is free of infinite internal chatter. Then all possible constructions of the matrix $[A_{ij}]_{nm}$ give rise to the same value.

This result is not difficult to prove by induction on the dimensions of the matrix. One proves that all possible ways of constructing the matrix are equivalent to some canonical construction, for example



(This is the construction where the columns are put together first, associating to the top; then the complete columns are put together, associating from the left.)

Let us conventionally adopt the definition that $[A_{ij}]_{nm}$ is the above form (in every case, whether or not it is free of infinite internal chatter). The force of lemma 7.6 is that the particular canonical form chosen is irrelevant in all cases where infinite internal chatter is impossible.

The table which we saw earlier is a demonstration of the fact that we cannot expect to prove that the conditions of 7.6 are satisfied by examination of cases. We therefore need to find some general method for proving the absence of infinite internal chatter from networks. An example of such a method is provided by the next result, which is similar in statement, effect and proof to 5.20.

7.7 Lemma

If "a" is any label define the predicate P^a on M as follows:

$$P^a(A) \equiv \neg\exists w_0 < w_1 < \ldots < w_i < \ldots \in \mathrm{dom}(A) .\ \forall i .\ (w_i \restriction a.\Sigma) = (w_0 \restriction a.\Sigma) .$$

(i) If A and B are two processes satisfying P^a $(a \in \{u,d,l,r\})$ then so do $[A]$, $[A|B]$ and $\left[\frac{A}{B}\right]$.

(ii) If each of A and B satisfies P^a $(a \in \{u,d,l,r\})$ then $[A|B]$ and $\left[\frac{A}{B}\right]$ are both free of infinite internal chatter.

The above result tells us that if each A_{ij} of some matrix satisfies the same P^a $(a \in \{u,d,l,r\})$ then the conditions of 7.6 are satisfied. The critical feature about any predicate with this property is that it satisfies (i) above, as

well as implying freedom from infinite internal chatter, for then it implies that all the partial constructions of $[A_{ij}]_{nm}$ also satisfy it. The four conditions P^u, P^d, P^l and P^r correspond intuitively to ensuring that there is always some flow of information towards one of the four faces of a matrix. This motivates the following four conditions, which correspond to the corners of the matrix in much the same way as the P^a correspond to the faces.

7.8 Lemma

If a and b are two distinct labels define the predicate Q^a_b as follows:

$$Q^a_b(A) = \neg\exists\, w_o < w_1 < \ldots < w_i \ldots \in \mathrm{dom}(A).\forall i.\ w_i\restriction(a.\Sigma \cup b.\Sigma) = w_o\restriction(a.\Sigma \cup b.\Sigma).$$

(i) If $a \in \{u,d\}$ and $b \in \{l,r\}$ and A,B are two processes satisfying Q^a_b then $[A]$, $[A|B]$ and $\left[\frac{A}{B}\right]$ all satisfy Q^a_b.

(ii) If $a \in \{u,d\}$, $b \in \{l,r\}$ and each of A and B satisfies Q^a_b then $[A|B]$ and $\left[\frac{A}{B}\right]$ are both free of infinite internal chatter.

Note that $P^a \Rightarrow Q^a_b$ and $P^b \Rightarrow Q^a_b$, so this second set of predicates is more general than the first.

Note also that the conditions Q^u_d and Q^l_r satisfy neither (i) nor (ii) above. For an example of this consider the two processes $A \Leftarrow d.a \rightarrow A$ and $B \Leftarrow u.a \rightarrow B$, which both satisfy Q^u_d trivially but for which $\left[\frac{[A]}{[B]}\right]$ does nothing but infinite internal chatter.

In common with the conditions used in a similar way in 5.20 the predicates P^a and Q^a_b are unfortunately not continuous, but again there are large classes of continuous predicates which imply them. (For example any predicate which expresses a bound on the number of "wrong" symbols which can appear before every "correct" one.)

It is clear that the "hexagonally connected array" can be defined in a very similar manner to the rectangular matrix of the above discussion. There are several excellent algorithms making use of arrays of these forms for such things as matrix multiplication and inversion as well as the more obvious uses such as the numerical solution of partial differential equations. For a description of a number of these see Mead and Conway ( ).

There are clearly many other possible configurations for parallel processes which we have not defined or studied so far. These include three (and higher) dimensional arrays; arrays with more complex interconnection; arrays in which all processes can be individually addressed by the environment; rings and even spheres of processes. It is not hard to adapt the techniques we have used so far to produce a reasonable definition for any of these. It is clear from the work we have done to date that we can expect each to have its own characteristic set of theorems, but that many of these theorems will follow set patterns.

Deadlock in networks

Deadlock is an important subject in the study of networks of parallel processes. So far when we have studied particular processes which have been defined as networks (in chapters 5 and 6) the desirable feature "freedom from deadlock" has almost always been proved as a corollary to a more powerful result. We have either proved that our networks were equivalent to other processes which were known to be free of deadlock (as in 6.8 and 6.9) or proved that the value of a network satisfied some predicate which implied freedom from deadlock (as in many of the buffer examples of chapter 5). These two techniques both have a worthwile place in our repertoire, but there are certainly going to be times when neither is applicable. Since freedom from deadlock is of such fundamental importance it is worthwile to try to find other methods for establishing it. The following is not an extensive treatment of such methods, merely a summary of a few ways in which the techniques we have developed might be applied to the problem.

The author believes that theorem 5.14 could be applied in many cases, in much the same way as it was applied in the proof of 5.27 (the "fundamental buffer theorem"). If one could identify a finite or infinite set of "states" of a network, and could prove some simple constructive relation between these states (very much as in 5.27) it would be possible to deduce freedom from deadlock (so long as the constructive relation preserves it).

So long as we can use 7.1 (and other similar results) to

bring all the hiding of a definition to the outside (so that it has the form A/X, where $\surd \notin \Sigma$ and the definition of A is free from hiding), and if we can show that the definition is free from infinite chatter, then we restrict ourselves to proving freedom from deadlock in the process before any hiding is carried out ("A" above). This is because, in the absence of infinite internal chatter, the only way deadlock (the refusal of "Σ" after some string) can occur in A/X is when A itself can refuse "Σ" after some possibly different string.

For example, in the process $A = ((B_X \|_Y C)/Z_{X \cup Y} \|_{U \cup V} (D_U \|_V E)/W)/S$, where $Z \cap (U \cap V) = W \cap (X \cap Y) = \emptyset$, if each of the hiding operators individually is free of infinite chatter, then to prove A free from deadlock it is sufficient to prove it in $((B_X \|_Y C)_{X \cup Y} \|_{U \cup V} (D_U \|_V E))$.

This fact has several uses, not the least of which is the fact that by effectively eliminating hiding from our consideration we can generally expect it to be much easier to find the constructive relations between states required for the previous method suggested. It can also be used to reduce some apparently complex problems to forms to which the next class of methods is applicable.

It is well known that networks which have the form of trees are generally easier to prove free from deadlock than those which possess loops. This fact is brought out by the next few results.

7.9 Theorem

Suppose that $A_1,..,A_n$ is a finite collection of processes $(n \geq 2)$ with associated alphabets $X_1,..,X_n$, and that A* is the result $((..((A_1 {}_{Y_1}\|_{X_2} A_2)_{Y_2} \|_{X_3} A_3)_{Y_3} \|...)_{Y_{n-1}} \|_{X_n} A_n)$ of combining them in parallel (where $Y_i = X_1 \cup ... \cup X_i$). Suppose further that the A_i and X_i satisfy the following conditions:

(i) each pair $(A_i {}_{X_i}\|_{X_j} A_j)$ $(i \neq j)$ is free of deadlock;

(ii) if i,j,k are all different then $X_i \cap X_j \cap X_k = \emptyset$;

(iii) for each i and string w $(A_i \underline{\text{after}}\ w)^o$ has non-empty intersection with at most one of the X_i s.t. $i \neq j$,

then A* can deadlock on string w only if there is a sequence $n_1,..,n_k$ of distinct elements of $\{1,..,n\}$ with the properties set out below.

(i) k (the length of the sequence) is at least three.

(ii) Letting $w_i = w \restriction X_{n_i}$, $Z_i = X_{n_i}$ and $B_i = A_{n_i}$, we have $w_i \in dom(B_i)$ for each $i \in \{1,..,k\}$.

(iii) There is a sequence $W_1,...,W_k$ of subsets of Σ such that (for each i) W_i is a maximal element of $B_i(w_i)$ and $Z_i - W_i$ is a non-empty subset of Z_{i+1} (or Z_1 when $i=k$).

proof

If A* can deadlock after w then $\Sigma \in A^*(w)$ (by definition of deadlock). It is an easy consequence of the definition of the parallel combinator that for any string v and set V we have $(v,V) \in A^*$ if and only if $v \in (X_1 \cup ... \cup X_n)^*$ and there exist sets $V_1,...,V_n$ such that

a) $V_i \in A_i(v \restriction X_i)$ for each i, and
b) $V \cap (X_1 \cup ... \cup X_n) = (V_1 \cap X_1) \cup ... \cup (V_n \cap X_n)$.

In the case when $V = \Sigma$ we may clearly assume that each of the sets V_i is maximal in $A_i(v \restriction X_i)$. Thus if A* can deadlock after string w we can deduce that there exist sets $V_1,..,V_n$ such that

a) V_i is a maximal element of $A(w \restriction X_i)$ for each i, and
b) $(X_1 \cup ... \cup X_n) = (V_1 \cap X_1) \cup ... \cup (V_n \cap X_n)$.

Let us suppose that the conditions of the theorem hold, that A* can deadlock after w, and that $V_1,...,V_n$ are as above. At most one of the $V_i \cap X_i$ can equal X_i, for otherwise there would be some $i \neq j$ such that the pair $(A_i {}_{X_i}\|_{X_j} A_j)$ would be able to deadlock after $w \restriction (X_i \cup X_j)$. From relation (b) above it can be seen that for each i $(X_i - \bigcup_{j \neq i} X_j) \subseteq V_i$; hence $X_i - V_i \subseteq \bigcup_{j \neq i} X_j$ for each i. Since by assumption each V_i is maximal in $A_i(w \restriction X_i)$ we can infer that $X_i - V_i \subseteq (A_i \underline{\text{after}} (w \restriction X_i))^{\circ}$. Putting these facts together, and using the fact that $(A_i \underline{\text{after}}\ v)^{\circ}$ has non-empty intersection with at most one X_j s.t. $i \neq j$ for all v, we see that for all i, with possibly one exception, there is some $j(i) \neq i$ such that $X_i - V_i$ is a non-empty subset of $X_{j(i)}$.

Any k such that j(k) is not defined must have $X_k \subseteq V_k$. Suppose there were some i with $j(i) = k$ where $X_k \subseteq V_k$; then it is easy to see that $(V_i \cap X_i) \cup (X_k \cap V_k) = (X_i \cup X_k)$. This tells us that it is possible that $(A_i {}_X\|_X A_k)$ deadlock after

string $w\restriction(X_i \cup X_k)$, which contradicts our assumptions. We can thus deduce that any k such that j(k) is not defined is not the image under j of any i. "j" is thus a function from I into I, where I is the finite set of indices on which it is defined. It is easy to see that, for any element r of I, in the sequence $r, j(r), j^2(r), j^3(r), \ldots$ there must be some repetition. In other words there is some finite sequence $n_1, .., n_k$ of distinct elements of I such that $j(n_i) = n_{i+1}$ $(i<k)$ and $j(n_k) = n_1$. Since $j(i) \neq i$ for all i we can deduce that $k \neq 1$. Suppose that k=2, then there are some i,j such that $i \neq j$, $(X_i - V_i) \subseteq X_j$ and $(X_j - V_j) \subseteq X_i$. By relation (b) in the construction of the V_k, and since $X_i \cap X_j \cap X_k = \emptyset$ for all k distinct from i and j, we get the relation $X_i \cap X_j = (V_i \cup V_j) \cap (X_i \cap X_j)$, which in turn implies that $(X_i - V_i) \subseteq V_j$ and $(X_j - V_i) \subseteq V_i$. Hence $(X_i \cap V_i) \cup (X_j \cap V_j) = X_i \cup X_j$, contradicting the fact that $(A_i \,{}_{X_i}\|_{X_j}\, A_j)$ cannot deadlock after $w\restriction(X_i \| X_j)$. We can thus infer that $k \geq 3$, and it is easy to see that by construction the sequence $n_1, .., n_k$ satisfies all that is required of it.

The above theorem, interpreted informally, means that in a network of processes satisfying conditions (ii) and (iii) (which we will interpret shortly), if all pairs of processes are free of deadlock then whenever deadlock occurs it must contain a ring of at least three distinct processes each demanding to communicate with one of its neighbours and refusing to communicate with its other neighbour (which wants to communicate with it). Condition (ii) of the theorem says that every communication is participated in by at most two processes. Condition (iii) says that each process can never be willing to communicate with two of its neighbours (i.e. it can never have the option to communicate with either one neighbour or the other).

7.10 Corollary

In any network which both satisfies the conditions of 7.9 and for which the graph formed with nodes A_i and edges between A_i & A_j when $X_i \cap X_j \neq \emptyset$ is a tree, there can be no deadlock.

proof

If $n_1,..,n_k$ is the sequence which is produced by 7.9 when there is any deadlock then $A_{n_1},...,A_{n_k}$ is a circuit in the graph.

Since any tree of processes automatically satisfies condtion (ii) of 7.9 this result tells us that in any tree whose elements satisfy condition (iii) it is possible to eliminate global deadlock by showing that it is impossible between any pair of its components.

In any graph with only a few circuits (or a lot of circuits of only a few types) it is often possible to reduce the proof of freedom from deadlock to the checking of a few cases. (If there are n circuits in a graph there are 2n possible sequences $n_1,..,n_k$ arising from 7.9). Thus in a ring of processes there are only two possible circuits.

The following example (after E.W. Dijkstra) represents a ring of processes, any of which might be requested to obtain some "token" (which is passed round the ring) so that it can carry out some action, and then release the token for use by other processes. Suppose $n \geqslant 3$, we can define processes X_i, Y_i for $i \in \{0,1,..,n-1\}$ thus

$$X_i \Leftarrow i.get \rightarrow i+1.find \rightarrow i.pri \rightarrow i.cri \rightarrow i.rel \rightarrow Y_i$$
$$\quad \square\ i.find \rightarrow i+1.find \rightarrow i.pri \rightarrow i-1.pri \rightarrow X_i$$

$$Y_i \Leftarrow i.get \rightarrow i.cri \rightarrow i.rel \rightarrow Y_i$$
$$\quad \square\ i.find \rightarrow i-1.pri \rightarrow X_i$$

(all arithmetic is modulo n)

(X_i represents a process without the token "pri", which before it allows the environment to perform its critical action "cri" must put in a request to its neighbour to find it and pass it back. Y_i represents a process with the token, which will allow the environment to perform "cri" or will pass it to its neighbour if requested.)

If we set up a ring with one token, which initially is in the 0-process (each process being given as its alphabet the set of symbols which it can potentially use), then it would be useful to be able to prove it free of deadlock. It is easy to see that if R is the process which results from combining $Y_0, X_1, ..., X_{n-1}$ in parallel, the graph produced in the manner of

7.10 is a ring, there being edges between the i-process and i+1-process for each i (addition modulo n).

It is not hard to prove that the processes and alphabets which make up the networks satisfy conditions (ii) and (iii) of 7.9. Also the proof that they satisfy condition (i) can be reduced to a fairly easy analysis of cases, proving by mutual induction that $(X_{i+1}\ {}_{Z_{i+1}}\|_{Z_i}\ X_i)$, $(Y_{i+1}\ {}_{Z_{i+1}}\|_{Z_i}\ Y_i)$ and $(Y_{i+1}\ {}_{Z_{i+1}}\|_{Z_i}\ X_i)$ are free of deadlock, all other cases (non-adjacent processes) being trivial. (Z_i is used to denote the alphabet of the i-process.)

We can thus infer that the network R can only deadlock if each process is waiting for its i+1 neighbour or if each process is waiting for its i-1 neighbour. It is quite easy to prove that at each point in the ring's history there is exactly one process with the token, in the sense that either it has never left Y_o and no other process has been passed it (communicated i.pri) or there is exactly one process i which has not passed the token on (communicated i-1.pri) since it last received it (communicated i.pri). However a process can only be waiting for its i+1 neighbour if it does not have the token, and can only be waiting for its i-1 neighbour if it does have the token. We can thus infer that the network is free of deadlock.

The above argument, while it is not an absolutely rigorous proof, can easily be extended to one.

It is not hard to extend the above to the cases when any non-zero number of tokens are initially in the network. If all the processes are without tokens initially (i.e. are all equal to X_i) then deadlock occurs.

In the above example, and others where it is applicable, 7.9 seems to formalize the intuitive reasons why one expects a network to be free of deadlock. This makes it a useful tool in eliminating deadlock. Condition (iii) of 7.9 is fairly restricive in its nature (it means that the result is not applicable to networks such as the "five dining philosophers" of 6.2). It is possible if we drop condition (iii) to prove a weaker version of 7.9, which is stated below and has a similar proof.

7.11 Theorem

Suppose that $A_1,..,A_n$ is a finite collection of processes ($n \geqslant 2$) with associated alphabets $X_1,..,X_n$, and that A* is the result $((..(A_1 \underset{Y_1}{\|_{X_2}} A_2) \underset{Y_2}{..}) \underset{Y_{n-1}}{\|_{X_n}} A_n)$ of combining them in parallel ($Y_r = X_1 \cup .. \cup X_r$). Suppose also that whenever i,j,k are distinct $X_i \cap X_j \cap X_k = \emptyset$. Define a ternary relation on $\{1,..,n\}$ as follows:

$$R(i,j,k) \Leftrightarrow i \neq j \;\&\; j \neq k \;\&\; k \neq i \;\&\; \exists w.(((A_i \underline{\text{after}}\ w)^{\circ} \cap X_j) \neq \emptyset) \;\&\; (((A_i \underline{\text{after}}\ w)^{\circ} \cap X_k) \neq \emptyset)$$

If i and j are distinct elements of $\{1,..,n\}$ define clique(i,j) to be the smallest subset of $\{1,..,n\}$ which both contains i and j and is closed under R in the sense $s,t \in X \;\&\; R(s,t,u) \Rightarrow u \in X$. (clique(i,j) places an upper bound on the sets of processes which might interfere directly with any communication between A_i and A_j.)

If X is any non-empty subset of $\{1,..,n\}$ define A_X to be the result of combining in parallel all the processes indexed by elements of X:

$$A_X = A_i \quad \text{if } X = \{i\}$$

$$A_X = ((..(A_i \underset{X_i}{\|_{X_j}} A_j) \underset{X_i}{\|_{X_l}} ..) \underset{X_i \cup .. \cup X_s}{\|_{X_k}} A_k) \quad \text{if } X = \{i,j,..,s,k\} \text{ has at least two elements.}$$

If the network A* is free from local deadlock in the sense that all subsets of cliques are deadlock-free, then deadlock can only occur in the complete network A* on string w when there exist $Z_1,..,Z_n \subseteq \Sigma$ which satisfy the following:

(i) Z_i is maximal in $A_i(w \upharpoonright X_i)$;

(ii) $X_1 \cup .. \cup X_n = (Z_1 \cap X_1) \cup .. \cup (Z_n \cap X_n)$;

(iii) for each pair $i \neq j$ there is some element k of clique(i,j) with the property that $(X_k - Z_k) \subseteq \bigcup \{X_r \mid r \notin \text{clique}(i,j)\}$;

(iv) there exists a sequence $n_1,..,n_k$, not contained within any one clique, such that $(X_{n_i} - Z_{n_i}) \cap X_{n_{i+1}} \neq \emptyset$ for each i (where k+1 is interpreted as 1).

* * * * * * * * * *

The power of this result depends critically on the structure of the space of cliques in a given network. In general one can easily show that clique(i,j) = clique(j,i) for each pair (i,j), and that $k \in \text{clique}(i,j) \Rightarrow (\text{clique}(i,k) \subseteq \text{clique}(i,j))$. When all processes in a network are different (as will usually be the case in a network without hiding) one can unambiguously

think of cliques as being sets of processes and being defined by pairs of processes.

If $X_i \cap X_j = \emptyset$ we must have clique(i,j) $= \{i,j\}$, and it is easy to see that so long as there is at least one pair r&s such that $X_r \cap X_s \neq \emptyset$ we can disregard all such cliques. Bearing this in mind the only cliques we need consider in the "five dining philosophers" example are $\{PHIL_i, FORK_i, PHIL_{i\oplus 1}\}$ (which is both clique($PHIL_i$,$FORK_i$) and clique($FORK_i$,$PHIL_{i\oplus 1}$)) and $\{PHIL_1,\ldots,PHIL_5,B\}$ (which is clique(B,$PHIL_i$) for each i).

Note that under the assumptions of 7.9 we have clique(i,j) $= \{i,j\}$ for all i and j, so that the conclusions of 7.11 more or less imply those of 7.9. We can prove an analogous result to 7.10, though the proof is harder:

7.12 Theorem

Suppose that A* is the network described in 7.11, and that in addition to satisfying the hypotheses of 7.11 it is a tree in the sense of 7.10; then it is free of deadlock.

proof

If there were deadlock possible in A* on string w then there would exist sets $Z_1,\ldots,Z_n$ and a sequence $n_1,\ldots,n_k$ satisfying the conditions of 7.11(i-iv).

Claim that every sequence $m_1,..,m_r$ of elements of $\{1,..,n\}$ with the properties that (i) $(X_{m_i} - Z_{m_i}) \cap X_{m_{i+1}} \neq \emptyset$ $(i < r)$ & $(X_{m_r} - Z_{m_r}) \cap X_{m_1} \neq \emptyset$, (ii) $r \geq 2$ and (iii) $m_i \neq m_{i+1}$ $(i < r)$ & $m_r \neq m_1$ has the property that $\{m_1,..,m_r\} \subseteq$ clique(m_1,m_2).

We will prove this by induction on r. The result is trivially true when r=2, so let us suppose that it holds for all such sequences with length less than r and that $m_1,..,m_r$ is such a sequence. There are two cases to consider: either m_1 is repeated later in the sequence or it is not.

In either case it is easy to see that the processes indexed by the m_i form a connected subtree of the network, and that A_{m_i} is joined to $A_{m_{i+1}}$ by an edge, as is A_{m_r} to A_{m_1}. In the second (or) case we can thus deduce that $m_2 = m_r$, which means that by our inductive hypothesis $\{m_2..m_{r-1}\} \subseteq$ clique(m_2,m_3) (we must have $r > 3$ for otherwise $m_2 = m_{2+1}$). However by construction $(((A_{m_2} \text{ after } w \restriction X_{m_2}) \circ \cap X_{m_3}) \neq \emptyset)$ & $(((A_{m_2} \text{ after } w \restriction X_{m_2}) \circ \cap X_{m_1}) \neq \emptyset)$ (since $m_2 = m_r$ and $(X_i - Z_i) \subseteq (A_i \text{ after } w \restriction X_i) \circ$). This

tells us that $m_3 \in clique(m_1,m_2)$, which as we observed earlier implies that $clique(m_2,m_3) \subseteq clique(m_1,m_2)$. This completes the proof in this case.

In the "either" case we must have $m_1 = m_s$ for some $s \neq r$, $s \neq 1$. Clearly each of $m_1,..,m_{s-1}$ and $m_s,..,m_r$ is a sequence which satisfies the hypotheses of this result, so we can deduce that $\{m_1,..,m_{s-1}\} \subseteq clique(m_1,m_2)$ and that $\{m_s..m_r\} \subseteq clique(m_1,m_{s+1})$ from our inductive hypothesis. However by construction $(((A_{m_1} \text{ after } w \upharpoonright X_{m_1}) \overset{\circ}{\cap} X_{m_2}) \neq \emptyset)$ & $(((A_{m_1} \text{ after } w \upharpoonright X_{m_1}) \overset{\circ}{\cap} X_{m_{s+1}}) \neq \emptyset)$ (as before), which implies that $m_{s+1} \in clique(m_1,m_2)$. Thus $clique(m_1,m_{s+1}) \subseteq clique(m_1,m_2)$, which completes the proof of the "either" case.

We may therefore conclude that this holds of all such sequences. This contradicts our assumption of deadlock, for the sequence $n_1,...,n_k$ satisfies the hypotheses but not the conclusion of the above result (by 7.11(iv) it is not contained in any clique).

The cliques of a tree have a very regular form, as shown by the next result (the proof of which is similar to parts of the above).

7.13 Lemma

If the network A* of 7.11 has the form of a tree, then if A_i and A_j are not adjacent we have $clique(i,j) = \{i,j\}$. If A_i and A_j are adjacent then the nodes indexed by $clique(i,j)$ form a connected subtree with the property that whenever A_r and A_s are two other adjacent nodes either $clique(i,j) = clique(r,s)$ or $clique(i,j) \cap clique(r,s)$ has at most one element.

7.12 provides us with a very general technique for proving freedom from deadlock in finite trees. As an example of this consider the system formed by combining n philosophers and n+1 forks in a line:

$FORK_0 \,||\, PHIL_0 \,||\, FORK_1 \,||\, PHIL_1 \,||\, \ldots \,||\, FORK_{n-1} \,||\, PHIL_{n-1} \,||\, FORK_n$

(processes defined as in 6.2 but with ordinary rather than mod 5 arithmetic).

The cliques of this system are just $\{FORK_0, PHIL_0\}$, $\{FORK_n, PHIL_{n-1}\}$ and $\{FORK_i, PHIL_i, PHIL_{i-1}\}$ $(1 \leq i \leq n-1)$ whose subsets are easy

to prove free from deadlock. Having done this we can infer that the whole system is deadlock-free.

Many of the arguments and results in this section have been (implicitly or explicitly) graph-theoretic. The basic type of graph we have been interested in is that of "requests" on deadlock (the directed graph which results from drawing in an edge leading from each element of a deadlocked system to the elements of the system from which it would accept communication). Theorems 7.9 and 7.11 tell us things about the gross structure of these graphs (essentially that they contain loops of certain types). It is often possible to limit the possibilities for these graphs if we have some local knowledge of structure. One might for example be able to show that if in such a deadlock graph some component of the network has an edge leading to it from one of its neighbours then the edges leading from it have some particular structure. In particular one might be able either to completely eliminate the possibility of the loops implied by 7.9 or 7.11 or cut down the number of possible loops to manag le proportions.

It is not very hard to apply this type of technique to prove freedom from deadlock in the "five dining philosophers" example of 6.2. The outline of such a proof is set out below.

(i) Show that in any deadlock graph an edge from $PHIL_i$ to $FORK_i$ implies that $FORK_i$ has a unique edge leading out of it which leads to $PHIL_{i\boxminus 1}$, and that when $PHIL_{i\boxminus 1}$ has an edge leading to $FORK_i$ there is a unique edge leading from $FORK_i$ (which leads to $PHIL_i$).

(ii) Show by consideration of $PHIL_i \| FORK_i$ that whenever $FORK_i$ has a unique edge from it leading to $PHIL_i$ then $PHIL_i$ has a unique edge leading from it, to $FORK_{i\boxplus 1}$. Correspondingly whenever $FORK_{i\boxplus 1}$ has a unique edge from it, leading to $PHIL_i$, then $PHIL_i$ has a unique edge from it, to $FORK_i$.

(iii) By 7.11 there must be some edge leading out of the clique $\{B, PHIL_1, PHIL_2, .., PHIL_5\}$, which means by (i) and (ii) that there is a ring of edges leading round the "outside" of the graph, and that these are the

only edges leading out of the $FORK_i$s and $PHIL_i$s.

(iv) Show that this is impossible by the structure of B (by counting "getup"s and "sitdown"s).

It ought to be possible to develop a better notation for this type of argument, and to develop a simple theory to make its application easier.

This concludes our brief discussion of deadlock. We have seen that it is often possible to reduce the problem of proving freedom from deadlock in a general network to the corresponding problem in a network free from hiding. We conjectured that it might be reasonably easy to attack some such networks by means of constructive relations between states, and developed various graph-theoretic methods for application to such networks. There are other, similar, predicates which one might wish to prove in lieu of simple freedom from deadlock. These might include proving that every component of a network remains alive (in some sense), or proving that a process is "live" in the sense that at all times there is some way of getting it back to its initial state. It should be possible to adapt several of our methods to such problems (of the two examples quoted here the first is likely to be easier than the second).

footnote

Note that the identification of deadlock with the appearance of "$\surd$" in the image of strings is critical in this section. This would not have been the case if we had chosen the finite refusal sets model in chapter 4 instead of the directed-closed model. If we had done this it would have been necessary to prove a result equivalent to the difficult consistency theorem 4.15 in order to prove 4.9, 4.11 and their corollaries.