Calculi for Synchrony and Asynchrony

2. TUTORIAL PAPERS

Using algebra for concurrency : some approaches

Robin Milner

Edinburgh University, September 1983

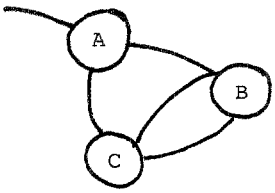
Introduction

A prominent feature of any algebra is that its expressions, by their form, either exhibit the structure of the objects which they represent, or exhibit the way in which those objects were built, or could be built, or may be viewed. Often indeed an object does not possess structure, but we impose structure upon it by our view of it - and thereby understand it better. A rectangular array of numbers, for example, is not of itself a row of columns, nor is it a column of rows; these are views which we impose upon it, and any linear expression of such an array will impose some such biassed view.

So it is no accident that algebra is useful in understanding complex distributed systems; for such systems must have many parts (else they would not be complex), and a structured view is essential in understanding something with many parts.

In designing an algebra for distributed systems, we are first faced with an inherent difficulty; the connectivity of the components is not in general tree-like, whereas the structure of an algebraic expression is always tree-like. It follows that the connectivity of a system is not expressible merely by the form of an expression. However, the analysis of an expression into subexpressions will express the analysis of the system into subsystems - and the expression will often be chosen in such a way that the subsystems which are thus identified are physically meaningful, and possess properties from which properties of the complete system follow naturally.

A more detailed problem for the algebra is: what is the nature of the connecting links between subsystems of a distributed system? In a system such as the following



do the arcs represent directed channels carrying data from one node to another, in which case do they have any memory capacity? Or do they represent simply the contiguity of the objects represented by the connected nodes - an interface across which they exchange an immediate interaction? And in either case does the forked arc from B to A and C carry a communication between B and both A and C, or does it signify that a single communication occurs between either B and A or B and C but not both?

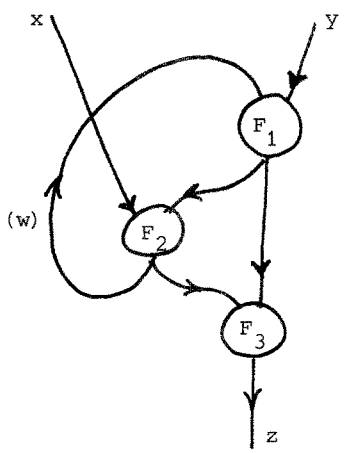
One modest purpose of this paper is to show that precise answers to these questions can indeed be given by choosing one algebra or another, and that the different choices differ markedly. In section 2 we look at an algebra in which the arcs represent unbounded queues of data elements. In sections 3 - 6 we look at more primitive (but more general) models in which the arcs are immediate interfaces; in this case the queues of section 2 would themselves be represented by nodes of a particular nature. Another - not so modest - purpose is to illustrate in each case that algebraic proofs of system properties can indeed be carried out. We have no space either to treat complex examples or to show the full richness of the algebraic theories concerned. Instead, we hope that readers will find interest in the significance and importance of the fundamental choices in building an algebraic model - namely, fixing the nature of the objects, and fixing the basic operators by which a rich enough class of objects can be built.

In the final section 7, we comment very briefly upon the relation between algebra and other theoretical tools for analysing concurrent systems.

2. Pipelining : Kahn networks

A particularly simple and attractive form of concurrency is provided by the Data flow idea which arose first from the work of Jack Dennis at MIT and his group, but was first put on an algebraic footing by Gilles Kahn first at Stanford and then at IRIA (now INRIA) near Paris.

Simple networks are considered in which each node receives a (possibly infinite) sequence of values along each of zero or more input lines, and delivers such a sequence along zero or more output lines. If an output line serves more than one succeeding node, then its values go to all of them. There may be loops in the network, and typically some lines are designated as inputs and outputs of the entire network. An example is shown below, in which the nodes are uninterpreted



Now in this network, the node F_2 may be interpreted as a function of two input sequences, yielding one output sequence; the other nodes similarly.

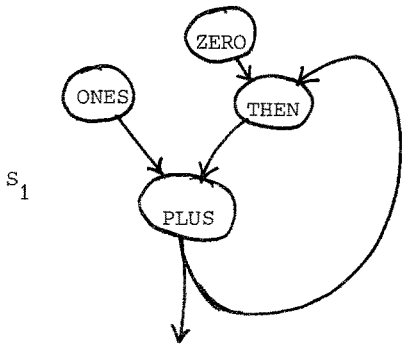
The question is: given the functions F_1,F_2 and F_3, how may we express the function represented by the entire network, which takes input sequences x and y and yields output sequence z ? The answer is gained simply by introducing an unknown w standing for the sequence of values which travel along the single arc which loops back form F_2 to F_1. For then the output of F_1 is $F_1(w,y)$ - a sequence - and this is fed into F_2, so that w satisfies the equation

$$w = F_2(x, F_1(w,y))$$

and it can be shown that under simple conditions there is a unique solution to this equation - though depending on F_1 and F_2 it may be an infinite, finite or even empty sequence. Finally, since F_3 receives as inputs w and $F_1(w,y)$, the output z is given by

$$z = F_3(w, F_1(w,y))$$

As a more concrete example, consider the following net S_1 (with no input lines and one output line). We can calculate that it generates the sequence $S_1 = 1.2.3. \cdots$ of all positive integers.



S_1

To do this, we must first interpret the four nodes:

ZERO = 0.ε (a zero, followed by the empty sequence ε)

ONES = 1.ONES (the infinite sequence of ones)

THEN(x,y) = first(x).y (the sequence y preceded by the first member of the sequence x)

PLUS(x,y) = (first(x) + first(y)).PLUS(rest(x),rest(y))
(adds the pairs of inputs, one by one)

Note that any sequence x can be split into its leading member first(x) and its remaining sequence rest(x). Now the sequence S_1 generated by the whole net clearly satisfies

$$S_1 = \text{PLUS}(\text{ONES}, \text{THEN}(\text{ZERO}, S_1)) \qquad (1)$$

Now we can begin computing S_1 as follows:

$$S_1 = \text{PLUS}(\text{ONES}, \text{THEN}(0.\varepsilon, S_1))$$

$$= \text{PLUS}(1.\text{ONES}, 0.S_1)$$

$$= 1.\ \text{PLUS}(\text{ONES}, S_1) \qquad (2)$$

To go further, let's define inductively

$$S_{k+1} = \text{PLUS}(\text{ONES}, S_k) \quad (k = 1,2,\ldots) \qquad (3)$$

If we can show that for all $k \geq 1$

$$S_k = k.S_{k+1} \qquad (4)$$

then we have what we want, for it will follow that

$$S_1 = 1.S_2 = 1.2.S_3 = 1.2.3.S_4 = \ldots$$

$$= 1.2.3.\ \ldots$$

So let us prove (4) by induction on k. It certainly holds for $k = 1$, since $S_1 = 1.S_2$ follows from (2) and (3); so now assume that (4) holds at k, and prove it at $k + 1$:

$$S_{k+1} = \text{PLUS}(\text{ONES}, S_k) \qquad \text{by definition of } S_{k+1}$$

$$= \text{PLUS}(1.\text{ONES}, k.S_{k+1}) \qquad \text{by assumption}$$

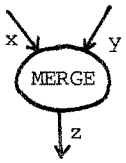
$$= (k+1).\text{PLUS}(\text{ONES}, S_{k+1}) \qquad \text{by PLUS}$$

$$= (k+1).S_{k+2} \qquad \text{by definition of } S_{k+2}$$

which is what we wanted.

Nets of this kind can, in a very succinct manner, compute interesting and nontrivial functions. Wadge (in his work on LUCID) and others have given many examples, and the proofs can always be carried out in the above algebraic style - which is definitely a mathematical style rather than a specialised program-proof methodology.

Certainly the nets exhibit a form of concurrency and communication, namely "pipelining" ; what are their limitations? First, the model and the proof method become considerably more complex as soon as the nodes are not assumed to be determinate - or at least not fully described as functions; an example of a non-determinate node is the MERGE



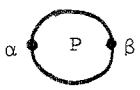
in which it is known that z contains all members of x and of y in the right order, but interleaved in an unspecified manner (e.g. according to order of arrival, which is not specified in the model). Such non-determinism can be very useful. Second, the model attains its simplicity partly by omitting one feature of behaviour which we may sometimes wish to take into account, namely the relative order in which the input elements are received and the output elements delivered in a network. For - considering our first illustrated net with nodes F_1,F_2 and F_3 - the solution which determines z as a function of x and y does not indicate how many elements from x and y are absorbed before the first, second,... element of z is generated.

A third limitation is that any realization or implementation of the model will require unbounded memory capacity to represent the queues of values which build up on internal arcs of a network. It is important to be able to ignore this detail at a high level of modelling, but if memory capacity is to be modelled then the Kahn networks are not the appropriate tool.

To achieve a general model of communicating agents which removes these limitations involves, apparently, a totally different approach. We illustrate one such approach - but emphasize that the purity of the Kahn model should tempt us to use the latter whenever we can accept its limitations.

3. Interacting agents

We look now at an algebraic way of presenting agents which interact with other agents linked to them. A convenient simplification, to begin with, is to treat interaction as neither input nor output of values, but as a symmetric handshake between two (or perhaps more) agents; its occurrence carries no value from one agent to another, but merely means that something (e.g. a high voltage pulse) rather than nothing has occurred. Each agent - which may be realised by one or many processors - carries sites or ports on its periphery at which such events may occur; a Greek letter may be conveniently used both to name a port and to stand for an event occurring at that port. Here is an agent with two ports:



If we wish P to be an agent which alternates between α and β events, then it may be defined by the equation

$$P = \alpha.\beta.P$$

Of course, by expanding this, we can obtain

$$P = \alpha.\beta.\alpha.\beta.\alpha. \cdots$$

showing that the order of events (here, a strict alternation) at different ports is indeed recorded. A slightly more complex agent

which alternately performs either α_1 or α_2, then β, may be defined by the equation

$$Q = \alpha_1.\beta.Q + \alpha_2.\beta.Q$$

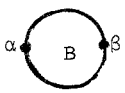
(which may be abbreviated by $Q = (\alpha_1 + \alpha_2).\beta.Q$) ; here the binary operator "+" between agent expressions indicates that either arm may be entered, but not both, during a computation. Thus we already have two operations on agent expressions; summation - meaning disjunction - and the prefixing $(\alpha.)$ of an atomic action at a particular port.

Typically, an agent P will have the form

$$P = \Sigma(\alpha_i.P_i)$$

where i ranges over some set, indicating the possible next actions of P.

We will not yet deal with how to stick agents together to form bigger agents; even with the slender resources introduced so far we can represent the handling of data values. For suppose we wish an agent



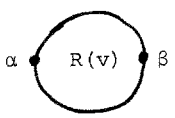
to represent a buffer with capacity one, alternately receiving values in $\mathbb{N}$ (non-negative integers) at port α and delivering them at β. We may do this by taking α to stand not for a single port, but for a family $\{\alpha_i \mid i \in \mathbb{N}\}$ of ports, one for each value; likewise β. Then our buffer can be defined

$$B = \sum_{i \in \mathbb{N}} (\alpha_i.\beta_i.B)$$

A convenient notation for this (avoiding writing Σ too often) is gained by introducing variables $x,y,\ldots$ over $\mathbb{N}$ - or whatever data domain is appropriate - and taking the first occurrence of such a variable to imply summation over $\mathbb{N}$:

$$B = \alpha x.\beta x.B$$

A rather different - but equally simple - agent with two ports is a storage register which can be assigned a value at α and can deliver its current value at β :



The parameter v in $R(v)$ indicates the current value stored in the register, and - using a variable as indicated above - we can define $R(v)$ thus:

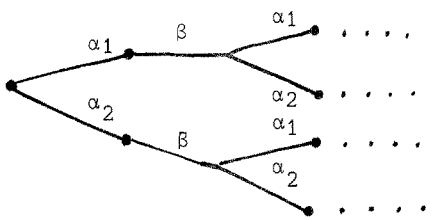
$$R(v) = \alpha x.R(x) + \beta v.R(v)$$

The importance of this example is that the formalism can treat both passive agents — e.g. memory — and active agents on exactly the same footing. This is valuable in many applications; if we consider the systolic arrays discussed by Mead and Conway, for example, then we find agents where memory capacity and processing power are united in the same element, and it would be irksome to have these roles treated by different notations.

It is often helpful to represent the possible "courses of action" of an agent graphically. For this purpose we can use a derivation tree. If we expand the agent Q , given above, a little way, then we get

$$Q = \alpha_1.\beta.(\alpha_1.\beta.Q + \alpha_2.\beta.Q) + \alpha_2.\beta.(\alpha_1.\beta.Q + \alpha_2.\beta.Q)$$

and we can conceive the indefinite expansion by the tree



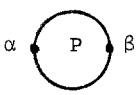
Such a tree represents both the action sequences which are possible (these are the paths of the tree) and the possible alternatives at each point in an execution (these are the branches from a node).

One final point before considering the composition of agents: the treatment so far is ambiguous in the sense that it has not been determined whether our agents are synchronous (forced to do something at every tick of a universal clock) or asynchronous (able to wait indefinately until an interaction is expected or demanded by the environment). Operators which compose agents cannot remain uncommitted in this sense; from now on we shall adopt the second (asynchronous) alternative, but here remark that a synchronous calculus is equally possible.

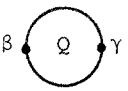
4. Product of agents

The focal point of an algebra of concurrent communicating agents, such as we are discussing, is undoubtedly the choice of an operator (a kind of product) which puts together two agents to make a single agent, whose behaviour reflects both the independent actions of each component and also their mutual interaction.

Let us consider two agents P and Q, which are buffer-like (as our very first example):



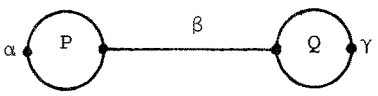
$P = \alpha.\beta.P$



$Q = \beta.\gamma.Q$

We revert to the simple form in which values are not carried by handshakes, but the addition of values poses no real difficulties. Notice that we have arranged P and Q to share a port name β ; this arrangement can be made by using "renaming" operators which we do not consider in this paper.

Now following the method of Hoare and his group, and also of George Milne, we wish to "multiply" P and Q together to form an agent which may be pictured



in which the actions α and γ may occur independently, but the action β may only occur (as "interaction") when both P and Q are capable of it. Let us denote this product operator by $\&_\beta$ — we may call it <u>β-synchronization</u>. There will be such an operator $\&_\alpha$ for any action α, and in general we may wish to use $\&_A$, <u>A-synchronization</u>, for any set A of actions. Sticking to $\&_\beta$, and recalling that we wish to consider agents expressed in the form $\Sigma\alpha_i.P_i$, what equation should be satisfied by

$$(\Sigma\, \alpha_i.P_i) \;\&_\beta\; (\Sigma\, \gamma_j.Q_j) \qquad ?$$

The product agent should be able to do any α_i which is $\neq \beta$, or any γ_j which is $\neq \beta$, or β itself provided $\alpha_i = \beta = \gamma_j$ for some i and some j. So we propose :

If $P \equiv \Sigma(\alpha_i.P_i)$ and $Q \equiv \Sigma(\gamma_j.Q_j)$,

then $P \&_\beta Q$ =

$$\sum_{\alpha_i \neq \beta} \alpha_i.(P_i \&_\beta Q) + \sum_{\gamma_j \neq \beta} \gamma_j.(P \&_\beta Q_j)$$

$$+ \sum_{\alpha_i = \gamma_j = \beta} \beta.(P_i \&_\beta Q_j)$$

The first and second sums represent the independent actions of P and Q respectively, while the third represents their interactions for all pairs i,j such that $\alpha_i = \beta = \gamma_j$. Such a general equation may be less easy to understand than a particular case, so let us calculate $P \&_\beta Q$ for our particular case in which $P = \alpha.\beta.P$ and $Q = \beta.\gamma.Q$. We proceed as follows:

$$\begin{aligned} P \&_\beta Q &= \alpha.(\beta.P) \&_\beta \beta.(\gamma.Q) \\ &= \alpha.(\beta.P \&_\beta \beta.(\gamma.Q)) \end{aligned} \qquad (1)$$

Here we have used the product rule once, noting that the only possible first action is α performed by P, since P cannot yet allow Q to perform β. Now we shall be able to find some equations which determine the behaviour $P \&_\beta Q$, for we have

$$\begin{aligned} \beta.P \&_\beta \beta.(\gamma.Q) &= \beta.(P \&_\beta \gamma.Q) \\ &= \beta.(\alpha.\beta.P \&_\beta \gamma.Q) \\ &= \beta.(\alpha.(\beta.P \&_\beta \gamma.Q) + \gamma.(\alpha.\beta.P \&_\beta Q)) \end{aligned} \qquad (2)$$

(this step reflects independent action by either component).

Also,

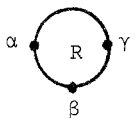
$$\begin{aligned} \beta.P \&_\beta \gamma.Q &= \gamma.(\beta.P \&_\beta Q) \\ &= \gamma.(\beta.P \&_\beta \beta.\gamma.Q) \end{aligned} \qquad (3)$$

while $\alpha.\beta.P \&_\beta Q$ is just the original $P \&_\beta Q$.

If we put (1), (2) and (3) together, and write R for $(P \&_\beta Q)$ and S for $(\beta.P \&_\beta \beta.\gamma.Q)$, we get the simple equations

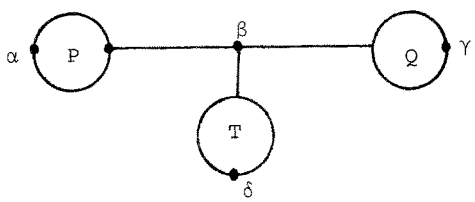
$$\begin{aligned} R &= \alpha.S \\ S &= \beta.(\alpha.\gamma.S + \gamma.\alpha.S) \end{aligned} \qquad (4)$$

Apparently, then, our composite agent R first performs α, then repeatedly performs β followed by α and γ in either order. In this simple case at least, we have been able to deduce a product-free description of the product of two agents; the equations (4) might have been written down to describe the behaviour of a single agent R with three ports:



Such transformations of description are the essence of the algebraic approach. It may be compared with the algebra of regular expressions, which describe the behaviour of finite automata in classical automata theory. But automata theory failed to provide a notion of product which was adequate to express how two concurrent automata can interact.

At this point, we should ask whether our product $P \&_\beta Q$ has given us what we want. On the one hand, we note that it could again be "β-synchronized" with yet another agent, T say, which is also capable of performing β from time to time. The resulting agent $P \&_\beta Q \&_\beta T$ could be pictured as follows



which reflects that the action β will only be performed when all three agents are capable of it; thus β-synchronization permits us to model multi-way (not just two-way) handshakes. In passing, we note that it is easy to show that $\&_\beta$ is both commutative and associative, that is:

$$P \&_\beta (Q \&_\beta T) = (P \&_\beta Q) \&_\beta T$$

$$P \&_\beta Q = Q \&_\beta P$$

and such algebraic laws are essential in a smooth calculus.

On the other hand, we may have wished something different for the product of P and Q. For we may argue that the intermediate port β sould serve only for interaction between P and Q, and that it should not

be visible or accessible outside the product. In other words, we look for a form of product in which the only remaining visible actions are α and γ.

Following Hoare and Milne, we choose to achieve this not by modifying the product, but by introducing an operation called hiding which may be applied to any agent to conceal some of its actions. Specifically, if R is some agent possibly capable of performing β from time to time, then

$$R/\beta$$

will represent R's behaviour with all β actions omitted. (Of course we have operators "$/\alpha$" for all actions α, and operators "$/A$" for all sets A of actions.) Thus, instead of forming the product $R = P \,\&\, Q$ of our two agents, we shall often prefer to form the hidden product $R' = (P \,\&_\beta\, Q)/\beta$; looking back at equations (4) above, we shall expect R' to satisfy instead the equations

$$\begin{aligned} R' &= \alpha.S' \\ S' &= \alpha.\gamma.S' + \gamma.\alpha.S' \end{aligned} \qquad (4')$$

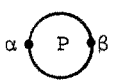
— i.e. the hidden product first performs α, and thereafter repeatedly performs α and γ in either order. We shll not give the exact definition of the hiding operators here; it requires refinements which would take up too much space.

There are variants of the product operators $\&_\alpha$ and $\&_A$. Instead of pursuing them further, we shall now look briefly at an alternative originally introduced by the author; it has an advantage over the above in that just one product operator is required, in place of a family of operators indexed by actions α or by sets A of actions, but a disadvantage (in the form given here) that it models only two-way (not multi-way) handshakes. Part of the purpose of describing two approaches in this paper is to dispel the tempting impression that there is one clearly best algebra of concurrent processes.

5. An alternative agent product

To define an alternative product, we make a new assumption, namely that for every action α there exists an inverse action $\bar{\alpha}$, and that an interaction may occur between two agents whenever they may perform inverse actions. Moreover, this interaction constitutes for the product agent a distinguished action — denoted by the symbol τ — which we may call the silent action. By this means we can get away with just a single operator, called composition and denoted by "$|$", in place of the family $\&_\beta$ of operators — though (as here presented) we thereby sacrifice multi-way handshakes and retain only two-way handshakes.

Let us treat the same example as before:



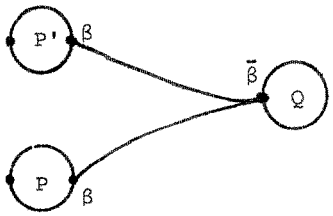
$P = \alpha.\beta.P$ $\qquad$ $Q = \bar{\beta}.\gamma.Q$

(Note that we have named one of Q's ports inversely to one of P's ports, to make the product work). Rather than writing down a general equation for the product $(\Sigma\alpha_i.P_i) \mid (\Sigma\gamma_j.Q_j)$, we shall state the rule informally: the next action of $P|Q$ can be either an action which is possible for P or Q independently, or a τ action if P and Q can perform inverse actions.

We now begin to compute $P|Q$:

$$P|Q = \alpha.\beta.P|\bar{\beta}.\gamma.Q$$
$$= \alpha.(\beta.P|\bar{\beta}.\gamma.Q) + \bar{\beta}.(\alpha.\beta.P|\gamma.Q)$$

No inverse actions were possible (hence no τ action results) on this first step. But the second term, which was absent when we worked out $P \&_\beta Q$, represents the possibility that Q's $\bar{\beta}$ action may be complemented by a β-action performed not by P but by some further agent P' to be added later. In other words, systems like



can be formed by this product operation, representing how Q may interact with either P or P' (but not both) through the same port. There is a disjunctive quality in "|" which contrasts with the conjunctive quality of "$\&_\beta$".

If we were to proceed further in computing $P|Q$ we would get a rapid expansion; for example, for one of the terms we would get

$$\beta.P|\bar{\beta}.\gamma.Q \;=\; \beta.(P|\bar{\beta}.\gamma.Q) \;+\; \tau.(P|\gamma.Q) \;+\; \bar{\beta}.(\beta.P|\gamma.Q)$$

since the three possibilities of independent action by either component, and interaction, are all present.

But we can avoid so much expansion by using an analogue to the hiding operator. This time, we require something a little different; we use an operator $\setminus\beta$ called restriction. The effect of $R\setminus\beta$ is to discard from R all alternatives (appearing as summands of R) which begin with either β or $\bar{\beta}$. This means that the only use of these actions within R is to permit interaction between different components of R (yielding τ actions for R itself).

Let us now compute, not $P|Q$, but $R'' = (P|Q)\setminus\beta$:

$$\begin{aligned} R'' &= (\alpha.\beta.P|\bar{\beta}.\gamma.Q)\setminus\beta \\ &= \alpha.(\beta.P|\bar{\beta}.\gamma.Q)\setminus\beta \\ &= \alpha.\tau.(P|\gamma.Q)\setminus\beta \end{aligned}$$

At each step, alternatives involving uncomplemented actions β or $\bar{\beta}$ have been discarded. We now compute $S'' = (P|\gamma.Q)\setminus\beta$:

$$\begin{aligned} S'' &= (\alpha.\beta.P|\gamma.Q)\setminus\beta \\ &= \alpha.(\beta.P|\gamma.Q)\setminus\beta \;+\; \gamma.R'' \\ &= \alpha.\gamma.(\beta.P\;Q)\setminus\beta \;+\; \gamma.R'' \\ &= \alpha.\gamma.\tau.S'' \;+\; \gamma.\alpha.\tau.S'' \end{aligned}$$

Putting these together, we have obtained the following product-free description of our composite agent R" :

$$\begin{aligned} R'' &= \alpha.\tau.S'' \\ S'' &= \alpha.\gamma.\tau.S'' + \gamma.\alpha.\tau.S'' \end{aligned} \qquad (4'')$$

If we compare this with the equations (4') in the previous section, we see that the only difference is in the presence of some τ actions, which are so to speak traces of internal communications. In fact, there is mathematical justification for the algebraic law

$$\alpha.\tau.P = \alpha.P$$

(for arbitrary α and P), and this law removes all difference between (4') and (4")!

There is a pleasant duality between the pair of operators $(\&_\beta, /\beta)$ on the one hand, and the pair $(|, \backslash\beta)$ on the other:

$\&_\beta$ (β synchronization) <u>demands</u> certain interactions;

$/\beta$ (β hiding) <u>releases</u> β from further synchronization demands;

while

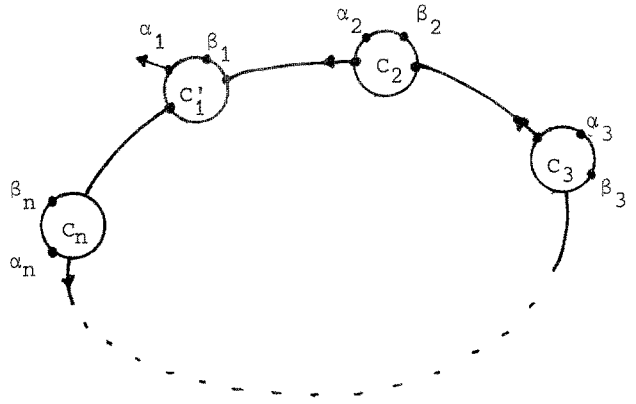
$|$ (composition) <u>permits</u> both independent action and interaction;

$\backslash\beta$ (β restriction) <u>inhibits</u> certain uncomplemented actions.

In both cases, the lesson learned is that a pleasant algebraic treatment is obtained by separating the synthesis of concurrent agents into two phases: a product operation which takes account of their interaction, and an encapsulation operation which prevents external access to internal interfaces. The importance of the separation is that a binary product operation can be applied repeatedly - to link an arbitrary number of agents together — before applying an encapsulation operation to "enclose" the composed system.

6. A bigger example

Consider the following system:



It consists of a ring of n identical agents, each waiting for a communication from its predecessor in the circular order (as indicated by the little arrows). except for C_1' which is waiting for a communication on its a_1 port. It is intended to act as a distributed scheduler for n independent agents $P_1,\ldots,P_n$ (not shown). P_i will be connected to C_i at both ports α_i and β_i ; P_i requests (at α_i) to initiate a certain activity, and indicates (at β_i) when it has completed the activity. The scheduling discipline is as follows:

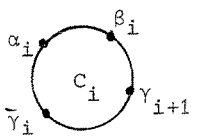
(1) Requests are treated in cyclic order, starting with P_1;

(2) Each P_i must alternate between α_i and β_i — i.e. it cannot be running more than one instance of the activity at any time.

It is quite easy to define the agents C_i, and then to put them all together, using either product operator ; moreover, the algebraic proof that the resulting system has the two desired properties is not hard. If we are going to use the second form of agent product, then we will define C_i as follows:

$$C_i = \bar{\gamma}_i.C_i'$$

$$C_i' = \alpha_i.(\gamma_{i+1}.\beta_i.C_i + \beta_i.\gamma_{i+1}.C_i)$$



(where subscript addition is modulo n)

Intuitively, C_i first learns (at $\bar{\gamma}_i$) from his predecessor that he may now grant a request (at α_i) ; after that request he then transmits request permission (at γ_{i+1}) and receives termination signal (at β_i) in either order; then he repeats.

It is not hard to see that this system works. In fact, the scheduler is expressed as

$$S = (C_1 | C_2 | \cdots | C_n) \setminus \gamma_1 \setminus \gamma_2 \cdots \setminus \gamma_n$$

and the formulation and proof that S satisfies properties like (1) and (2) above is not difficult. It has been given as an example in the author's book "A Calculus of Communicating Systems", and can equally well be treated using the operators $(\&_\beta, /\beta)$ instead of $(|, \setminus\beta)$.

Conclusion

This short introduction to an algebraic approach to concurrency has necessarily omitted some intricate details, as well as paying no attention to other algebraic approaches (for example, Vaughan Pratt has suggested an approach which generalises the Kahn networks in a different manner). What we hope to have shown is that four kinds of operator - namely atomic action ($\alpha.$), summation ($+$), product ($\&_\beta$ or $|$) and encapsulation ($/\beta$ or $\setminus\beta$) - together give great expressive power, and moreover satisfy interesting algebraic identities.

In a methodology for proof about particular systems, we almost certainly need more than "just" algebra. With algebra, we can typically prove equations between agent expressions; we often wish also to prove that an agent possesses some property which is not expressible by an equation. It is therefore important to look at the relation between such algebras and logics - Temporal or Modal logics - designed to express interesting properties of processes.

Another important relationship to study is between the algebraic approach and Net Theory. The emphases of these models are different; communication is the cornerstone of the algebra (in the present approach), while Net Theory emphasizes causal independence, provides a totally different graphical aid to intuition, and provides different tools for abstraction.

Finally, synchronous systems demand some form of treatment. The author has found one way of integrating the above asynchronous algebra with an algebra of synchronous (clocked) systems; this method has some mathematical simplicity - for example, the algebra becomes more conventional being at least a semi-ring (with agent sum and product as the semi-ring operations) - but is by no means obviously the best integration possible.