THE REPRESENTATION OF COMMUNICATION AND CONCURRENCY

by

George J. Milne

September 1980

Computer Science Department

California Institute of Technology

Pasadena, California 91125

This research has been supported by the
Science Research Council of the United Kingdom,
the National Science Foundation under grant MCS-8011925
and the Air Force Office of Scientific Research
under grant AFOSR-80-0274

ABSTRACT

A formal system is described within which we may represent the communication and concurrency features found in systems of interacting computing agents. This formal system may be used both as a model in which to represent the behaviour of existing systems of computing agents or as a language in which to program desired systems. The notion of acceptance semantics is introduced and it is in terms of this that we give meaning to programs constructed in our framework.

ACKNOWLEDGEMENTS

A number of people have contributed to the ideas which are presented here.

This research is an offshoot of that performed at Edinburgh University with Robin Milner; his influence has contributed much to the ideas expressed here. Suggestions have also been made by Tony Hoare and his group in Oxford and by my colleagues here at Caltech.

Financial support was provided by the Science Research Council of the United Kingdom, the National Science Foundation and the Air Force Office of Scientific Research.

TABLE OF CONTENTS

THE REPRESENTATION OF COMMUNICATION AND CONCURRENCY

1. INTRODUCTION

1.1 Description of Systems

In the study of computation, or computer science if you like, we not only have to learn how to build systems on which to perform computations; we also must know how to describe such systems.

Formalism is required to enable us to describe and discuss a computation system in a precise and unambiguous manner. Computation is a precise science and there is little use in only being able to describe systems informally.

When systems are computing, they are performing actions; evaluating functions or communicating with other systems, for instance. The sequence of actions performed by a system is its behaviour. It is the behaviour of a system which we wish to be able to describe formally.

We shall use mathematical and logical concepts in constructing a framework in which to describe behaviour. This allows us to specify and describe systems in precise terms and to reason formally about their behaviour using mathematical techniques. The specifications of a system using a formalism allows one to inform others about a system unambiguously. This, together with the ability to perform proofs about the behaviour of a system, are two of the main reasons we wish formal descriptions.

In systems where there is a single locus of control, executing a program written in a serial language for instance, we are able to describe the behaviour using functions. Either we take the function to be the behaviour itself and so the program denotes the function, as in denotational semantics, or the behaviour is given in terms of an abstract machine which evaluates the program. Here, in this operational semantics, we have functions with the state of the abstract machine being the domain and range.

We wish to produce a formalism in which to represent the behaviour present in systems which are composed from a number of computing agents, either hardware or software. These agents will operate concurrently and are linked together forming a complex of interacting components. Networks, multiprocessor machines and concurrent programs fit into the above category. Indeed, most systems we meet involve some degree of concurrency and so could be described in the formalism, or specification language, which we construct in this paper.

1.2 Our Approach

The formalism presented aims to allow us to describe complexes of computing agents in which communication and concurrency is inherent. The formalism aims to be both a framework in which to represent the behaviour of existing systems and a language in which to program the desired behaviour of projected systems. It is believed that most existing systems can be represented in the formalism in an intuitive fashion and that we should find it more natural to program certain phenomena in our language rather than in existing frameworks.

We shall use the words framework, formalism and language interchangeably.

There are certain features which we require our language to clearly represent. These occur frequently in systems and include the communication between agents, the inherent nondeterminism within some agents, and the possibility of deadlock among the complex of agents.

That the language achieves these goals will be illustrated by giving a collection of examples which examine the representation of such features in detail. These examples should also indicate to the reader the underlying philosophy adopted in this work. It is hoped to be able to fully justify the choices taken in arriving at the formalism by use of a set of primitive examples.

The language itself consists of a set of operations allowing us to construct programs from smaller programs. A number of primitive operators gives us the lowest level programs. Each operator should not be considered just by itself but should be thought of in its relation to others, though some have more significance and are indeed more powerful than others.

Together with these operators we have a set of axioms which the programs will satisfy. These axioms permit us to manipulate the syntax of programs while preserving their semantics or meaning. Various properties of programs may be proved by the use of the axioms.

To give the semantics of the language we introduce the notion of acceptance. The behaviour of a program is given by its ability to accept, or reject, stimuli which are imposed on them. In terms of this semantics, we introduce the notions of equivalence and congruence between programs and show that our axioms are consistent. This experimental semantics is operational in nature; we experiment on programs by giving them all possible stimuli belonging to a set known as a sort and see how the programs react. They can react by evolving into a new program or by rejecting the stimulus and, in effect, destructing. Due to programs being able to react to one of a number of different stimuli at a time, we wish to observe how they react to all possible stimuli. Due to programs having the ability to represent nondeterminism, a number of different programs may result from some given stimulus.

In the language, we have features which allow us to distinguish between a program which can at some instance react to a number of different stimuli and produce (usually) different programs and a program which may produce different programs on receipt of a single stimulus. The former utilizes a choice construct in the language whilst the latter a nondeterminism language construct.

When programs communicate with others, then they themselves resolve the choice which can be made with only one interaction taking place at a time, but nondeterminism is an internal feature of a program and no other program can influence the outcome of a communication; the outcome is nondeterministic. Nondeterministic programs can arrive in two ways; they represent the behaviour of possibly physical computing agents which for some reason or other are intrinsically nondeterministic in behaviour; or they represent complexes of agents where we have abstracted away from the programs (or parts of programs) which cooperate to resolve a choice, so introducing nondeterminism. This situation arises where a choice could previously have been made but can now no longer be effected since the part of the program which participates in the choice has been hidden so preventing a choice being made externally to the program.

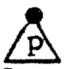
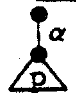
Communication, and so also our stimuli, take place via ports. If we imagine our program in reality as a machine running that, and only that program, the ports are the physical places on such machines where the wires between machines plug into. Ports have distinct identities and it is this which allows us to program distinct communications.

Meaning is given to our language using acceptors and this semantics should accurately describe the intended behaviour though sometimes in a rather complex manner. It is believed that a similar semantics which may, in a clearer way, give different meanings to the choice and nondeterminism operators, could be formulated by the introduction of the $\Box$ and $\Diamond$ modal operators, capturing the notion of "always" and "sometimes" respectively. This then gives us the ability in our semantics to talk about experiments which "always can" happen and about ones which "possibly may" happen. It would be hoped that these two semantics would be equivalent. A formulation of the modal semantics and an equivalence proof between the semantics remains to be performed.

2. THE LANGUAGE

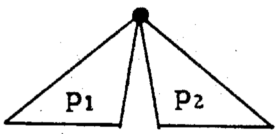
2.1 Primitive Language Constructs

To help illustrate the concepts which we capture using the language operators, we introduce synchronisation trees as a descriptive tool. The meaning of our syntax can then be represented by trees and the syntax taken to denote this tree semantics. We do not intend to formalise this denotational semantics; only use it to explain meaning in terms of the well-understood notion of trees. An operational semantics which gives meaning to programs by experimenting upon them, is given later.

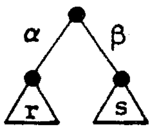
We have three primitive language operators, the first being guarding. This takes a program and appends in to something called a synchronisation label to produce a new program. For label α and program p then αp is this constructed program. Labels for the moment can be thought of as events, with programs being constructed out of them using our operators. Interaction between programs takes place using these labels. Semantically, programs denote trees while labels denote named arcs on the tree. If tree  is denoted by program p then program αp denotes the tree . Guarding gives us sequentiality. In the above program p follows the α event.

We wish programs to be able to cooperate with others, and depending on the other programs, to be able to perform different actions. A choice operation + allows this; it takes two programs and produces another program. For programs p_1 and p_2 then $p_1 + p_2$ is another such program.

In terms of synchronisation trees, the • node represents this externally resolvable choice. The program $p_1 + p_2$ denotes the tree

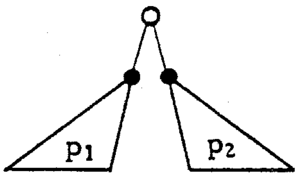


As an example, the two programs αr and βs are composed to give program $\alpha r_1 + \beta r_2$ which denotes tree

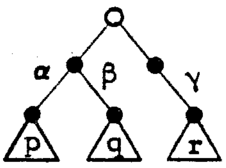


Now a program may, for some reason or other (to be made clear later), nondeterministically wish to perform certain events, or to perform some other events, but not their union. A program interacting with, or communicating with, this one has no control as to which of the sets it will be able to interact with; the nondeterministic choice will somehow be made internally. For programs p_1 and p_2, the program $p_1 \oplus p_2$ is their nondeterministic composition. The O node is used to indicate a nondeterministic branch in a tree.

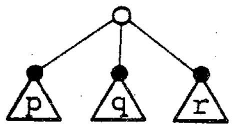
$p_1 \oplus p_2$ denotes the tree



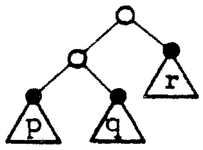
As an example, the program $(\alpha p + \beta q) \oplus \gamma r$ denotes the tree



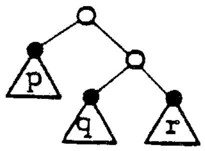
Our trees are thus bipartite. Some arcs are not labelled; these join ⊕ roots to ● nodes. Arcs joining ● roots to O nodes will always be labelled. O nodes will always have their O offspring separated from them by at least one level of ● nodes. This is due to us not labelling arcs appearing from O nodes and the associativity of the ⊕ operation. We have tree



rather than the trees



or



.

The ● nodes, corresponding to +, may have ● as their direct descendants since the arcs joining these ● nodes will always carry a label.

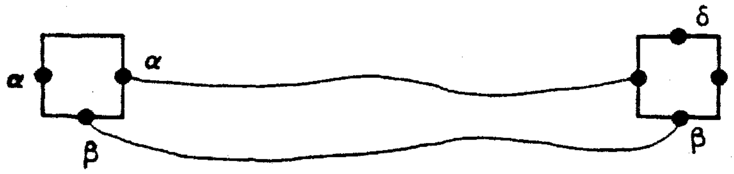
The final primitive operator is a nullary one Δ. That is, Δ takes no arguments, and is itself a program; the null program. Δ represents termination and deadlock. Termination and deadlock are very similar with termination being specified directly as a property of a single program and deadlock being a property of a number of interacting programs. An agent which wishes to perform event x or event β (to be decided on by the environment, i.e., other programs) and in either case to then terminate, is represented by program $\alpha\Delta + \beta\Delta$. The appearance of Δ representing deadlock will be described in Chapter 5.

2.2 Sorts

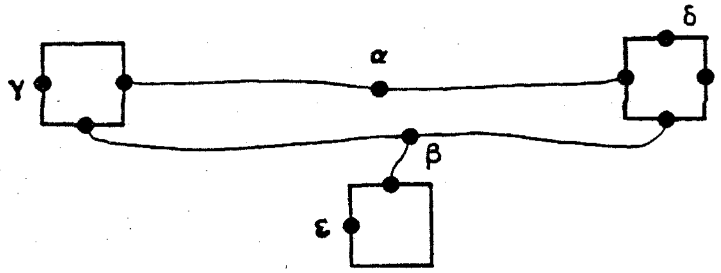
Imagine a program as representing a special purpose machine executing that, and only that code. As programs will communicate with others so machines will communicate with other machines. To carry the analogy further, we join machines using wires over which communications pass. Each machine has a number of ports which can be considered as the sockets into which the wires are fixed. Ports are used to both send and receive signals and to enable us to specify how send and receive ports are interlinked, we introduce naming on ports via labels.

To illustrate the labelling and linkages between machines, we may picture machines as boxes. These boxes have ports on the periphery, some of which are labelled. The convention between machines is that similarly labelled ports are linked.

Hence we may get



We shall permit two or more ports with the same label to be joined together and to facilitate this we move the label to a connector between the joined ports, and join on further ports via this connector. A connector has no further significance. Three machines, which can be thought of as being concurrently active, can be linked together as follows:



A set of labels is known as a sort. Each of the machines, or boxes above, has a sort and the program which describes the behaviour of each machine will also have a sort.

The labels which form sorts lie in the name-set Λ. Every program we have will have a sort though not all sorts will be made explicit. The labels used by a program must lie in its sort but the sort may well contain others.

Thus, program $\alpha p + \beta q$ may have the following sorts: $\{\alpha, \beta\}$, $\{\alpha, \beta, \delta\}$, $\{\alpha, \beta, \varepsilon\}$ and many others. The rules for defining programs and for constructing programs from programs will tell us what the sort of a program is. The sort is therefore implicit and is given by context.

As Δ is a program then this null program will also have a sort. We therefore may have $\Delta \neq \Delta$ where the two occurrences of Δ may have different sorts. Subscripting of Δ with its sort will sometimes be used to avoid such problems, but again the context usually helps us.

2.3 Machines, Behaviours and Programs

What is the difference between machines, programs and behaviours? As we wish to be able to represent both hardware and software computing agents without distinction, then to make it easier to talk about the topology of these concurrent systems, we use the physical analogy; machines, ports and wires. A machine, of course, has a behaviour given in our formalism, and the machine may be realised physically or by using software; it does not concern us which.

We represent the behaviour not the implementation which produces that behaviour.

Conceptually we are producing a formalism in which to both represent concurrent systems and to program concurrent systems. What a representation and a program have in common is behaviour, or to use another word, meaning. The representation or model aims at capturing the underlying behaviour of the system; the program is a representation or a denotation of an intended behaviour. In fact, we will model a concurrent system using a program and the behaviour of the system is then given in terms of a formal semantics for the language in which the program is written. Formally, a model is designed with respect to the properties we wish to represent. In our case it is qualitative concepts such as termination, deadlock and equivalence but other properties may be modelled. For instance, we have performance models of operating systems using queuing techniques and simulation. We can also have such quantitative models as in the realm of complexity theory. We do not concern ourselves with these two latter types of properties.

We use the word model in the sense of quantitative representation. To model a complex of interlinked computing agents requires us to represent the complex by some syntax; namely, a program. To specify the behaviour of this program requires that we have a semantics for the language in which the program is constructed. This semantics gives meaning to the program. Thus the representation of some particular computing agent, or complex of computing agents, consists of a program and the semantics of the language. We can then reason about properties of the real system by reasoning about their representations in our formalism; our model.

The model we describe in this paper consists of a language and a semantics for the language. We model the behaviour of a complex of agents via a program in our language. Its behaviour, and so that of the complex, is given by the formal semantics of the language.

Our language for complexes of a single computing agent has been described so far. This sequential language is extended to deal with a world of concurrently active computing agents; complexes with more than one agent. This is dealt with later but first we will give an operational semantics for the language defined so far.

2.4 Acceptance Semantics

We have informally described the properties of single computing agents which our formalism may represent. We will now formally give the semantics of our syntactic constructs using the notion of acceptors. Our operational semantics is then what we shall call an acceptance semantics.

Definition. For every subset L of a name-set Λ then L is known as a sort. The acceptance semantics is given by an acceptance relation of type

$$(\mathrm{PROG} \times \Lambda) \quad \times \quad (\mathrm{PROG} \cup \{*\})$$

where PROG is the set of programs to be the words algebra W_Σ formed from the signature Σ where

$$\Sigma = \Lambda \cup \{\Delta , + , \oplus , \bullet , -\}$$

Δ is a nullary operator and Λ is a set of unary operators known as labels. $+$, $\oplus$ and $\bullet$ are all binary operators while for λ ranging over Λ then $-\lambda$ is a unary operator. The set PROG may be partitioned according to sort such that $\text{PROG} = \bigcup_{L} \text{PROG}_L$

PROG is then the union of all phyla PROG_L for all sorts L; that is, for all subsets L of the name-set Λ.

Our acceptance relation between $(\text{PROG} \times \Lambda)$ and $(\text{PROG} \cup \{*\})$ will be restricted to taking $\langle$program, label$\rangle$ pairs where the labels lie in the sort of the program. The relation is undefined for $\langle$program label$\rangle$ pairs where the label lies outside the sort of the program.

Technically, we could have effected this by having a family of relations, one relation for each sort. Then for each sort L we have a relation of type

$$(\text{PROG}_L \times L) \times (\text{PROG}_L \cup \{*\}).$$

It will generally be understood what the sort of a given program is and thus we need not usually explicitly specify it.

The symbol $*$ is not in the syntax of the language but is a meta-symbol used in the semantics.

Meaning is given operationally to programs in our language using the family of acceptance relations. A program and a label from its sort will produce either a new program or the symbol $*$ under the relation. An experiment is performed here in that a label is given to a program and the resulting program (or *) indicates how the original program reacts to the stimulus of the label.

For programs $p, p' \in PROG_L$ and label $\varepsilon \in L$ then the relation $\langle (p, \varepsilon), p' \rangle$ for sort L is written as

$$(p, \varepsilon) \xrightarrow{L} p'$$

and indicates that after an ε stimulus the program p evolves into program p'. Our relation can be thought of as defining an acceptor; here program p accepts ε and evolves to program p'.

If $(p, \varepsilon) \xrightarrow{L} *$ then under our ε stimulus p produces * ; it does not accept the ε and so does not produce a new program. The label ε has not been rejected though; program p has evolved into a degenerate state on receipt of the ε stimulus.

The sort of a program will generally be understood and it will usually not be necessary to put a sort superfix on the symbol $\rightarrow$, as mentioned previously.

For any program/label pair a number of outcomes may result; the inherent nondeterminism of the language may cause different programs to result when the same original program is provided with the same stimulus label. But whether a program is nondeterministic or not, to fully specify its meaning we need to see how it reacts to all possible stimuli contained in its sort.

The semantics of programs constructed from the signature set $L \cup \{\Delta, +, \oplus\}$ for some, $L \subseteq \Lambda$ is given by the smallest relation satisfying for α and β ranging over L

1. $(\alpha p, \alpha) \rightarrow p$

2. $(\alpha p, \beta) \rightarrow *$

3. $\dfrac{(p, \alpha) \rightarrow p'}{(p+q, \alpha) \rightarrow p'}$

4. $\dfrac{(q, \alpha) \rightarrow q'}{(p+q, \alpha) \rightarrow q'}$

5. $\dfrac{(p, \alpha) \rightarrow * \ \wedge \ (q, \alpha) \rightarrow *}{(p+q, \alpha) \rightarrow *}$

6.1 $\dfrac{(p, \alpha) \rightarrow p'}{(p \oplus q, \alpha) \rightarrow p'}$

6.2 $\dfrac{(p, \alpha) \rightarrow *}{(p \oplus q, \alpha) \rightarrow *}$

7.1 $\dfrac{(q, \alpha) \rightarrow q'}{(p \oplus q, \alpha) \rightarrow q'}$

7.2 $\dfrac{(q, \alpha) \rightarrow *}{(p \oplus q, \alpha) \rightarrow *}$

8. $(\Delta, \alpha) \rightarrow *$

All relation symbols $\rightarrow$ could have had L as superfix. We therefore see that the following programs all have the same sort L where $\alpha \in L$;

$$p, \ q, \ \alpha p, \ p + q, \ p \oplus q, \ \Delta \ .$$

It should be pointed out that β is a member of L since $(\alpha p, \beta) \rightarrow *$. If $\beta \notin L$ then the relation for $(\alpha p, \beta)$ would be undefined.

Δ here is of sort L but we will have a Δ for each possible sort. To be more precise, we should have a collection of them each subscripted by its sort, but once again, we trust the context to keep us right and so avoid the need for these subscripts.

We have defined the semantics for a primitive language not having the ability to represent concurrency and communication. This comes later. First we shall explain the intuition behind our definition of the relations used in our acceptance semantics.

1.	$(\alpha, p, \alpha) \rightarrow p$	Program αp accepts an α label and evolves to program p. That is, an α stimulus is given to αp and program p results.
2.	$(\alpha p, \beta) \rightarrow *$	Program αp when given a β stimulus does not accept it.
3.	$\dfrac{(p, \alpha) \rightarrow p'}{(p+q, \alpha) \rightarrow p'}$	If program p can accept an α label and evolve to p' then so can program p+q. Clause 4 is similar.
5.	$\dfrac{(p, \alpha) \rightarrow * \wedge (q, \alpha) \rightarrow *}{(p+q, \alpha) \rightarrow *}$	If both programs p and q can fail when given an α stimulus, then p + q can also fail to accept an α.
6.1	$\dfrac{(p, \alpha) \rightarrow p'}{(p \oplus q, \alpha) \rightarrow p'}$	If program p receives an α stimulus and evolves either to program p' or to * (where $p' \neq *$) then so also does $p \oplus q$. Clauses 7.1 and 7.2 are similar.
6.2	$\dfrac{(p, \alpha) \rightarrow *}{(p \oplus q, \alpha) \rightarrow *}$	
8.	$(\Delta, \alpha) \rightarrow *$	The Δ program cannot accept any label. It will produce * on all stimuli.

In our language we shall wish to say which programs are equivalent, that is, which programs behave similarly. For two programs p and q, we say they are equivalent if they produce equivalent programs given the same stimulus, for all stimuli in their sorts. Also, for a given stimulus, if * results, then * must result when an equivalent program gets given the same stimulus.

<u>Definition.</u> Programs p and q of sort L are equivalent written $p \approx q$, iff $\forall \alpha \in L$. We have

a) $(p, \alpha) \rightarrow * \Rightarrow (q, \alpha) \rightarrow *$

b) $(q, \alpha) \rightarrow * \Rightarrow (p, \alpha) \rightarrow *$

c) $(p, \alpha) \rightarrow p' \Rightarrow \exists q'$ such that $(q, \alpha) \rightarrow q'$ and $(p' \approx q')$

d) $(q, \alpha) \rightarrow q' \Rightarrow \exists p'$ such that $(p, \alpha) \rightarrow p'$ and $(p' \approx q')$

This definition of equivalence is recursive but is adequate for finite programs. Finite programs terminate using Δ.

To handle recursively defined programs, we introduce a new equivalence $\sim$ where $\sim$ is taken to be the intersection of ascending indexed relations $\sim_n$. Thus $\sim$ is defined to be $\bigcap_n (\sim_n)$ where $p \sim_0 q$ always holds and for programs p and q of sort L

$p \sim_{n+1} q$ iff $\forall \alpha \in L$.

a) $(p, \alpha) \rightarrow * \Rightarrow (q, \alpha) \rightarrow *$

b) $(q, \alpha) \rightarrow * \Rightarrow (p, \alpha) \rightarrow *$

c) $(p, \alpha) \rightarrow p' \Rightarrow \exists q$ such that $(q, \alpha) \rightarrow q'$ and $p' \sim_n q'$

d) $(q, \alpha) \rightarrow q' \Rightarrow \exists p$ such that $(p, \alpha) \rightarrow p'$ and $p' \sim_n q'$

We will perform induction on the "depth" of the equivalence ~ when proving two programs p and q equivalent.

Later we shall prove that ~ is actually a congruence. That is, for all contexts C which can be constructed in our language, then

$$p \sim q \Rightarrow C[p] \sim C[q] .$$

A context is a program with a "hole" in it. C [p] has this hole replaced by program p.

3. INTERACTION BETWEEN PROGRAMS

3.1 Interaction Between Machines

Until now we have only concerned ourselves with the behaviour of a single machine. We now introduce language features which allow us to represent systems of machines. These machines will compute concurrently and will interact with each other by synchronised communication from time to time.

The behaviour which we capture in this framework is that of programs, or other computing agents, being given an external stimulus and evolving into some new program. The formal semantics of the "single component" language presented so far relies on this notion of stimulus. A stimulus is accepted and either produces a new program, or causes failure and so prevents any other stimuli from being accepted. Note that a program can always accept a stimuli but that it may well cause failure. Suppose the machine has behaviour denoted by program $\alpha p + \beta q$. Then it can receive a stimulus at port α and evolve into a new behaviour denoted by program p or it can receive stimulus at port β and evolve to program q. The environment only sends one stimulus at a time but regardless of whether it is an α or β the program can respond and evolve to sub-programs p and q respectively.

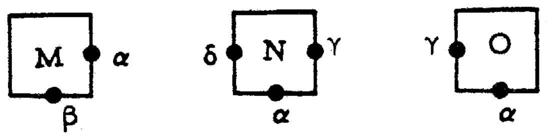
We have assumed that these stimuli are produced by the environment and our semantics says what happens to a program when they are accepted. The environment may well be another program, and so programs not only have the ability to receive stimuli but have the ability to generate stimuli.

Now the environment can also be thought of as a machine E generating stimuli on the various "lines" connecting it with machine M. When a synchronisation takes place the two machines M and E which are concurrently running programs, may exchange stimuli on one, and only one, of the lines and their behaviours, represented by programs m and e evolve into new programs m' and e'.

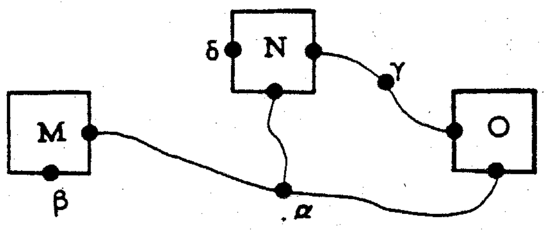
In our language, we shall not distinguish between generated and received stimuli. The reason for this is that we are interested in how the stimuli synchronise at instances in the life of our system. The generation and receipt of a stimulus is considered as an instantaneous act between programs. This act requires synchronisation between the participating programs. Synchronisation can therefore be thought of as an instantaneous exchange of stimuli with only one such synchronisation taking place at a time.

Earlier we showed how programs may be pictured as machines having ports. The ports are used to link machines together to form complexes, or systems, of machines. The convention adopted is that similarly labelled ports get joined through a connector node.

For machines M, N and O as follows:



they link together to give the following complex:



We see that M, N and O have made a three-way linkage using α ports and a two-way linkage via the γ ports. The β and δ ports still have not been connected but they, as well as the α and γ connectors, can be used to attach further machines.

We now have a convention for representing the communication structure of a system. But this is purely static as we have not specified the behaviour of the complex of machines; we have not said how they use the communication lines to exchange stimuli among themselves.

If m, n and o are the programs in our language which specify M, N and O—the programs running on M, N and O if you like—then the composite system is $m \bullet n \bullet o$. How does this composition operation work? That is, what are its semantics?

For programs p and q of sorts L_p and L_q respectively:

9. $$\frac{(p, \alpha) \rightarrow p' \wedge (\alpha \notin L_q)}{(p \bullet q, \alpha) \rightarrow p' \bullet q}$$

10. $$\frac{(q, \alpha) \rightarrow q' \wedge (\alpha \notin L_p)}{(p \bullet q, \alpha) \rightarrow p \bullet q'}$$

11. $$\frac{(p, \alpha) \rightarrow p' \wedge (q, \alpha) \rightarrow q'}{(p \bullet q, \alpha) \rightarrow p' \bullet q'}$$

12. $$\frac{(p, \alpha) \rightarrow *}{(p \bullet q, \alpha) \rightarrow *}$$

13. $$\frac{(q, \alpha) \rightarrow *}{(p \bullet q, \alpha) \rightarrow *}$$

The relation below the line in each of these clauses should have $L_p \cup L_q$ superscripted on the $\bullet$ operator, but we know implicitly that the sort of $p \bullet q$ is the union of the sorts of its components.

To explain how $p \bullet q$ behaves in terms of acceptor semantics, first note that we are considering p and q as two concurrently active programs which either communicate with each other by the exchange of stimuli or attempt to interact with the environment, i.e., receive a stimulus from the environment.

The first clause, clause 9, states that if p accepts an α stimulus which does not result in *, say program p, and if α is not a member of the sort of q, then $p \bullet q$ accepts an α stimulus and evolves into $p \bullet q$. Here, program p evolves to p' after accepting an α stimulus but q does not progress due to the α stimulus. Hence $p \bullet q$ evolves to $p' \bullet q$ on receipt of an α stimulus. Those labels in the sort of p but not in the sort of q are said to be <u>external</u>, thus α is an external label. The external stimulus, an α in this case, appears from the environment and not from the other program q; it is therefore external to the composite program $p \bullet q$.

Clause 10 gives meaning to external stimuli via labels lying in the sort of q but not in the sort of p.

Clause 11 states that if p accepts an α stimulus and evolves to p' and q accepts an α stimulus and evolves to q' with p' and q' not being * then $p \bullet q$ will accept an α stimulus and evolves to program $p' \bullet q'$. Here the α is an <u>internal</u> label; it lies in the intersection of the sorts of p and q. The two programs p and q synchronise on label α and exchange their α stimuli allowing both to evolve into the new programs p' and q'. These two programs are similarly composed using $\bullet$ to give what p$\bullet$q evolves to. The clause $\frac{(p, \alpha) \rightarrow *}{(p\bullet q, \alpha) \rightarrow *}$ states that if p on receipt of an α stimulus evolves to * then p·q on receipt of α stimulus also evolves to *.

If α is external to p$\bullet$q then α will not be in the sort of q. Thus if p on receipt of α cannot produce a new program then p$\bullet$q cannot produce a new program given the same stimulus. We therefore get * as a result. Of course, program p may involve the nondeterministic operator so p may give a new program as well on receipt of an α. Clause 9 deals with this case.

If α is internal to p$\bullet$q then α lies in the sort of both p and q. If p produces * on receipt of an α then so also will p$\bullet$q since regardless of whether q gives a new program or * on receipt of an α, p will give *. Of course, p may also produce a new program p' on receipt of α, which would be due to the presence of our nondeterminism construct. In this case either clause 11 would give a new program if q produces a program when given an α stimulus, or the dual of this clause would give an * as the result if q gives * on receipt of an α.

.2 A Derived Operator

Using our set of primitive operators over sorts, we may define other operators to simplify programs which are constructed in some common fashion. These derived operators may also be used to help illustrate how certain phenomena which are met in a concurrent world are represented in our language.

As an example of this, we define a polyadic composition operation $\prod$ using our binary concurrent composition operator $\bullet$. $\prod$ will be a sorted operator of sort $L_1 \times L_2 \times \cdots \times L_n \rightarrow L_1 \; L_2 \; \cdots \; L_n$, for n arguments. $\prod$ is defined by:

$$\prod(p_1, p_2, \cdots, p_n) = p_1 \bullet p_2 \bullet \cdots \bullet p_n$$

Why do we wish such an operator?

We introduce it because it helps to indicate the multiway synchronisation performed when we compose two or more programs using $\bullet$. An n-way synchronisation is represented by repeatedly applying the binary operator $\bullet$ to n or more programs, but we can imagine that with the $\prod$ operator this n-way synchronisation is performed as a single act. In fact, we can directly define a polyadic operator which does just this (rather than $\prod$ which is a derived operator) and which has $\bullet$ as its 2-argument instance but for technical reasons our binary $\bullet$ is preferred as a primitive in our language.

3.3 Properties of the Language

Using the language features introduced so far, we can construct programs and give their meaning using our acceptance semantics. Our understanding of how these language features were derived, i.e., in terms of stimuli, leads us to require that these language constructs possess certain features; that they satisfy certain laws. First let us introduce the notion of normal form programs.

Definition: A program is in CNF, (Conjunctive Normal Form), if either it is constructed from CNF subprograms using only the choice operator or it is constructed from a subprogram and a label using the guarding operator

Examples: $(\alpha p \oplus \beta q) + \gamma r$ is not in CNF

since $\alpha p \oplus \beta q$ is not in CNF

$(\alpha p + \beta q) + \gamma r$ is in CNF

$\varepsilon(\alpha p \oplus \beta q) + \gamma r$ is in CNF

Definition: A program is in DNF, (Disjunctive Normal Form), if it is constructed from subprograms using the nondeter mination operator.

Examples: $(\alpha p \oplus \beta q) + \gamma r$ is not in DNF

even though $\alpha p \oplus \beta q$ is in DNF

$(\alpha p + \beta q) \oplus \gamma r$ is in DNF

$\varepsilon(\alpha p + \beta q) \oplus \gamma r$ is in DNF

Definition: A program is in normal form if it is in CNF or DNF and its component subprograms are in normal form.

Note that the program $(\alpha p \oplus \beta q) + \gamma r$ is neither in CNF or DNF. It is thus not in formal form.

A CNF program can be written as $\sum_i \alpha_i p_i$ where $\sum_{i=1,n} \alpha_i p_i = \alpha_1 p_1 + \cdots + \alpha_n p_n$ and a DNF program can be written as $\bigoplus_{i=1,n} \alpha_i p_i = \alpha_1 p_1 \oplus \cdots \oplus \alpha_n p_n$. The empty sum $\sum_i = \Delta$; so Δ is the identity for $+$.

3.4 Program Axioms

Here we list certain laws which our language constructs satisfy and explain why we require them. Thinking of our language as a word algebra W_{Σ_1} where $\Sigma_1 = \Lambda \cup \{+, \oplus, \Delta, \bullet\}$ we may take these laws to be axioms.

$[+_1]$	$x + x = x$	idempotency
$[+_2]$	$x + y = x + y$	commutativity
$[++]$	$x + (y + z) = (x + y) + z$	associativity
$[+\Delta]$	$x + \Delta = x$	identity

In the above laws x, y, z are all CNF programs; they do not have operation $\oplus$ outermost.

We have the idempotency of our choice operation $+$ because to any program interacting with $x + x$ then the two copies behave just as if one were present; the commutativity of $+$ since the order of possible choices should be immaterial; the associativity of $+$ since we wish to allow for more than two choices to be made at certain times; the identity since nothing may interact with Δ.

[$\oplus_1$]	$x \oplus x = x$	idempotent
[$\oplus_2$]	$x \oplus y = y \oplus x$	commutative
[$\oplus\oplus$]	$x \oplus (y \oplus z) = (x \oplus y) \oplus x$	associative
[$+\oplus$]	$x + (y \oplus z) = (x+y) \oplus (x+z)$	$+$ distributes over $\oplus$
[$\alpha\oplus+$]	$\alpha x + \alpha y = \alpha x \oplus \alpha y$	
[$\oplus\oplus+$]	$x \oplus y \oplus (x+y) = x \oplus y$	

Note that Δ is not an identity for $\oplus$, but just for $+$.

We have the idempotency of $\oplus$ since no matter which subprogram the program $x \oplus x$ nondeterministically gives us, then they are the same, i.e., x; commutivity since our nondeterminism operator should treat its operands in an unordered fashion; associativity since we with to program agents having more than two possible behaviours.

We allow distributivity $x + (y \oplus z) = (x+y) \oplus (x+z)$ since the left-hand side says that it will accept the stimuli due to subprogram x but only accept the stimuli due to one or other of y or z and we do not know which. The right hand side says that we can accept the stimuli due to x and y or accept the stimuli due to x and z but not both. That is, both sides of the axiom state that the interactions contributed by x are always present together with either those of y or z and we do not know which.

We allow that $\alpha x + \alpha y = \alpha x \oplus \alpha y$ since if an α stimulus is given to either side of the axiom either x or y is the resulting program and the α stimulus has no control over which one results.

It may be supposed that we would also have the axiom $\alpha x \oplus \alpha y = \alpha(x \oplus y)$, that is, guarding distributing over $\oplus$. But the left-hand side nondeterministically gives us programs x or y on receipt of an z stimulus, i.e., after an α communication, whilst the right-hand side deterministically gives us the nondeterministic program $x \oplus y$. In terms of our acceptance semantics these programs are not equivalent since x is not equivalent to $x \oplus y$, and y is not equivalent to $x \oplus y$. The problem here is due to the different "levels" the nondeterminism appears at. Intuitively we would like the axiom $\alpha x \oplus \alpha y = \alpha(x \oplus y)$ but to ensure that $\alpha x \oplus \alpha y \sim \alpha(x \oplus y)$ would require a change in our acceptance semantics. We will not do this.

We have axiom $[\oplus \oplus +]$ since $x \oplus y \oplus (x+y)$ reacts in the same way as $x \oplus y$ to a given stimulus. If some stimulus α gets nondeterministically sent to x+y or x or y then the program (or *) which results will be the same as if it were nondeterministically sent to just x or y. The possible outcomes of $x+y$ are a subset of those for $x \oplus y$. The only way they differ is that $x \oplus y$ may produce an * where $x+y$ would not.

We have previously mentioned that we may introduce $\sum$ so that $\sum_{i=1,2} \alpha_i p_i = \alpha_1 p_1 + \alpha_2 p_2$. Similarly, $\sum\!\!\!\!\!\!\bigcirc_{i=1,2} \alpha_i p_i = \alpha_1 p_1 \oplus \alpha_2 p_2$. Now $\sum (= \Delta)$ is the identity for + and similarly, $\sum\!\!\!\!\!\!\bigcirc (\neq \Delta)$ is the identity for $\oplus$ and +. $\sum$ and $\sum\!\!\!\!\!\!\bigcirc$ are the "empty" choice and nondeterminism operators respectively and are meta-symbols which do not appear explicitly in our language but may be used for convenience. They are derived operators which <u>sugar</u> our language.

In the above we allow + to distribute over ⊕ but not vice-versa. The reason for this is that if we have both:

the existing law (I) $\quad x + (y \oplus z) = (x + y) \oplus (x + z)$

and its dual (II) $\quad x \oplus (y + z) = (x \oplus y) + (x \oplus z)$

then we produce an inconsistency in our set of axioms. An example illustrates this:

$(\alpha p_1 \oplus \beta p_2) + \gamma q$

$= (\alpha p_1 + \gamma q) \oplus (\beta p_2 + \gamma q)$ by (I)

$= (\alpha p_1 \oplus (\beta p_2 + \gamma q)) + (\gamma q \oplus (\beta 1_2 + \gamma q))$ by (II)

$= (\alpha p_1 \oplus \beta p_2) + (\alpha p_1 \oplus \gamma_q) + (\gamma q \oplus \beta p_2) + (\gamma q \oplus \gamma q)$ by (II)

$= (\alpha p_1 \oplus \beta p_2) + (\gamma q \oplus (\alpha p_1 + \beta p_2)) + \gamma q$ by + assoc., (II) and ⊕ indempotency

$= \left[(\alpha p_1 \oplus \beta p_2) + \gamma q\right] + \left[\gamma q \oplus (\alpha p_1 + \beta p_2)\right]$ by + assoc. and comm.

Now this is our first line composed with $[\gamma q \oplus (\alpha p_1 + \beta p_2)]$ using +. For this to hold either $[\gamma q \oplus (\alpha p_1 + \beta p_2)] = \Delta$, which it patently is not, and the identity law is used; or $\gamma q \oplus (\alpha p_1 + \beta p_2)$ is either equal to, or is a subDNF sum of $(\alpha p_1 \oplus \beta p_2) + \gamma q$, which is also false since the left-hand side has that a γ communication <u>may</u> possibly take place whilst the left-hand side says that it <u>always can</u> take place when provided with a γ stimulus, and idempotency would be used. Since we wish to allow law (I) then law (II) must produce this inconsistency and so (II) must be false. Note that p is a subDNF sum of q if every DNF clause in p is also a DNF clause in q.

We allow guarding to distribute over $\oplus$ but not over +. The reason for this is that $\oplus$ and + are clearly distinct as explained in Chapter 1 and so $x \oplus y \neq x + y$ provided $x \neq y$. Now $\alpha(x \oplus y) \neq \alpha(x+y)$ since following an α communication the left-hand side can perform the communications of either the x or y subprograms but we non-deterministically do not know which, whilst on the right-hand side any of the x and y communications may take place.

As $\alpha(x \oplus y) = \alpha x \oplus \alpha y = \alpha x + \alpha y$ using our axioms then $\alpha x + \alpha y \neq \alpha(x+y)$.

Using our acceptance semantics we can easily show that $x \oplus (y+z)$ and $(x \oplus y) + (x \oplus z)$ are not equivalent and that $\alpha x + \alpha y$ and $\alpha(x+y)$ are also not equivalent.

Now we introduce the axioms involving our concurrency operator.

$[\bullet_1]$ $\quad x \bullet x = x$ $\quad$ idempotent, for x in CNF

$[\bullet_2]$ $\quad x \bullet y = y \bullet x$ $\quad$ commutative

$[\bullet\bullet]$ $\quad x \bullet (y \bullet z) = (x \bullet y) \bullet z$ $\quad$ associative

$[\bullet +]$ for $x = \sum \mu_i x_i$, $\quad y = \sum \rho_j y_j$:

$$x \underset{LM}{\bullet} y = \sum_{\mu_i \notin M} \mu_i (x_i \underset{LM}{\bullet} y) + \sum_{\rho_j \notin L} \rho_j (x \underset{LM}{\bullet} y_i) + \sum_{\mu_i = \rho_j} (x_i \underset{LM}{\bullet} y_j)$$

$[\bullet \oplus]$ $x \bullet (y \oplus z) = x \bullet y \oplus x \bullet z$ $\quad$ where x has sort L and y has sort M

$\bullet$ distributes over $\oplus$.

Note that we normally do not need to subscript $\bullet$ as the sorts of its components will be understood. The $[\bullet +]$ law gives $\sum(=\Delta)$ if there exists no μ_i, ρ_j such that $\mu_i \notin M$ or $\rho_j \notin L$ or $\mu_i = \rho_j$. Δ will represent deadlock between the two components p and q.

We have the idempotency law for our concurrency operator $\bullet$ since identical programs will give stimulus to each other using all their labels leaving us with an identical program. This is the case provided that the component programs are in CNF, i.e., there is no $\oplus$ outermost. We do not have idempotency with DNF programs; consider $(x \oplus y) \bullet (x \oplus y)$ under axiom $[\bullet\ \oplus]$.

We have commutativity and associativity for $\bullet$ since it should be irrelevant the order in which component programs are composed.

For p and q in CNF then $p \bullet q$ should also be in CNF constructed out of guards whose labels are external to $p \bullet q$ and guards whose labels are internal to $p \bullet q$. The $\sum_{\mu_i \notin M} \mu_i(p_i \underset{LM}{\bullet} q)$ clause contributes to $p \bullet q$ those guards who appear in p but whose labels do not appear in the sort of q. Since only the p participates in this then the resulting program composed with the label is $p_i \underset{LM}{\bullet} q$. Similarly for labels in guards of q which are not in the sort of p. Finally, the clause $\sum_{\mu_i \notin \rho_j} \mu_i(p_i \underset{LM}{\bullet} q_j)$ contributes to $p \bullet q$ a guard whose label is the same as guards appearing in p <u>and</u> in q and as both p and q participate in this synchronisation the program $p_i \bullet q_j$ results which is composed by the guarding operation with label μ. Hence p and q have synchronised, exchanged μ_i stimuli, and have evolved to programs p_i and q_j which are recursively composed by the $\bullet$ operation.

If (1) we do not have any labels μ_i in guards of program p such that $\mu_i \notin M$, the sort of q; <u>and</u> if (2) we do not have any labels p_j in guards of q such that $p_j \notin L$, the sort of p; <u>and</u> if (3) there are no guards in p and in q having the same label then $p \underset{LM}{\bullet} q = \sum$, the "empty sum" which is the nullary operator Δ. Δ has sort $L \cup M$ and we can note that $\bullet$ is the only operation so far which changes sorts; by unioning the sorts of the components. The operations defined so far are sorted as follows:

$$\alpha : L \to L \text{ where } \alpha \in L$$

$$+ : L \times L \to L$$

$$\oplus : L \times L \to L$$

$$\bullet : L \times M \to L \cup M$$

Finally, the $[\bullet \oplus]$ law is present since p running concurrently with program $q \oplus r$ means that p will actually run concurrently with either subprogram q or subprogram r and we do not know which. $(p \bullet q) \oplus (p \bullet r)$ is the program where either p runs with q or p runs with r and the decision is made nondeterministically.

We have justified the laws, or axioms, informally. In a later chapter we prove that the laws actually hold with respect to our acceptance semantics. Thus, for the laws above, the left-hand side has the same meaning as the right-hand side and so we can quite happily use the laws to replace programs by semantically equivalent programs in any program context without changing the complete program meaning. This is the case since the laws satisfy our notion of equivalence which also happens to be a congruence.

In terms of the acceptance semantics and our notion of equivalence, we can prove that the following do <u>not</u> hold:

$$x \oplus (y + z) = (x \oplus y) + (x \oplus z) \quad \text{and} \quad \alpha(x + y) = \alpha x + \alpha y \; .$$

We have previously informally justified them as not being axioms in our language.

Two further operations require to be introduced into our language but we first introduce recursion and recursive definitions. These allow us to produce some interesting example programs using our language.

4. IDENTIFIERS, DECLARATIONS AND EXAMPLES

4.1 Identifiers and Declarations

To allow us to give practical examples, we extend the language W_{Σ_1} by giving it the ability to handle data structures. This does not change any of the preceding language philosophy but allows us to program, that is, to represent the behaviour of computing agents such as registers, memories, stacks and queues.

Let us introduce a set of identifiers I written using capitals, with which to name programs, and a constructor = to bind programs to identifiers. We therefore have our new language, the word algebra W_{Σ_2} where

$$\Sigma_2 = \Lambda \cup I \cup \{+, \oplus, \Delta, \bullet, =\} .$$

The = operation is of type $I \times Prog \rightarrow Dec$. An identifier in $p \in Prog$ is either the identifier currently being bound to p or else has previously been bound in p. The identifier after binding then names a program and thus has a sort, the same as the program. There are no restrictions on the sort of programs which we can bind with identifiers.

For $ID \in I$ and $p \in Prog$ we give meaning to a declaration by extending our acceptance semantics:

21.

$$\frac{(p,\alpha) \rightarrow p' \wedge (ID = p)}{(ID,\alpha) \rightarrow p'}$$

This construct permits us to define recursive programs, i.e., $P = \alpha P$.

Now we assume that after this declaration P identifies and has the same behaviour as αP. The declaration itself is not a program and we must compose it with a program to form a new program. We introduce a new operator where giving the alphabet $\Sigma_3 = \Sigma_2 \cup \{\underline{\text{where}}\}$. Composing a declaration with a program uses the where operation, of type:

$$(\mathrm{Prog}_1 \times \mathrm{Dec}) \rightarrow \mathrm{Prog}_2$$

Note that an identifier may not appear as a member of Prog_1 unless it has previously been declared by some member of Dec. Declarations and the use of where are really not necessary to our language and "sugar" it to make programs read more easily.

We can have the following programs:

$$\alpha P + \delta \Delta \quad \underline{\text{where}} \quad P = \alpha P + \beta \Delta$$

We assume here that the operators in alphabet Σ_1 bind more strongly than where, which in turn binds more strongly than = .

We may also nest declarations to get:

$$P \bullet Q \quad \underline{\text{where}} \quad P = \alpha P + \beta \Delta \quad \underline{\text{where}} \quad Q = \alpha \Delta$$

A derived operation and of type Dec × Dec → Dec can be defined as:

$$d_1 \ \underline{\text{and}} \ d_2 \overset{\text{def}}{=} d_1 \ \underline{\text{where}} \ d_2 \ .$$

The acceptance semantics for the extended language W_{Σ_3} is as for W_{Σ_2} with the addition of clause 22.

Let $\{m/n\}$ be the substitution operation in our semantics (not in our syntax) such that n gets replaced by m, then

22.

$$\frac{(p\{m/n\}, \alpha) \rightarrow p'}{(p \ \underline{\text{where}} \ \ n = m, \alpha) \rightarrow p'}$$

4.2 Data Structures

The ability to identify a program with a name allows us not only to write recursive programs but to introduce data structures. To effect the latter we allow identifiers to be not just names but names parameterised on some data structure.

We are not able to represent the communication of values in our language; we only can communicate, or synchronise, stimuli and this interaction indicates no more than that a synchronisation between two (or more) programs has taken place. For this reason our data structures will contain only boolean values though other types of values may be simulated.

Consider a boolean register. What behaviour does it have?

Well, if it is nonempty it can output its contents which will then remain unchanged or it can output a new boolean value and replace its contents by this new value.

Such a behaviour is given by a program identified by REG of sort $\{\varepsilon_1, \varepsilon_0, \sigma_1, \sigma_0\}$.

$$\mathrm{REG} = \varepsilon_1\ \mathrm{REG}(1) + \varepsilon_0\ \mathrm{REG}(0)\ \underline{\text{where}}$$

$$\mathrm{REG}(0) = \sigma_0\mathrm{REG}(0) + \varepsilon_1\mathrm{REG}(1) + \varepsilon_0\mathrm{REG}(0)$$

$$\underline{\text{and}}\quad \mathrm{REG}(1) = \sigma_1\mathrm{REG}(1) + \varepsilon_1\mathrm{REG}(1) + \varepsilon_0\mathrm{REG}(0)$$

This declaration can of course be combined with another program using the where operation.

A program REG is the initially empty register which can therefor only input a 1, represented by a stimulus at the ε_1 port, or a 0 represented by a stimulus at the ε_0 port. Since we cannot communicate the values 1 and 0 we have two separate ports which allow 1 or 0 stimuli to be effected, and so we have two input lines connecting this register program with other programs. We also have two output ports σ_1 and σ_0, since once a value has been loaded into the register our program must also have the ability to output the contents as well as load new contents. The identifiers REG(0) and REG(1) identify register programs whose contents are 0 and 1 respectively, the former being able to synchronise through the σ_0 port and the latter through the σ_1 port. This represents a 0 and 1 being output respectively.

In the above we have two input and two output ports representing the input and output of 1's and 0's. Conceptually we have only two ports ε and σ with the index indicating the value to be communicated. When "simulating" value communications like this, we do not distinguish between a sender and receiver. In fact, since more than two programs may synchronise on a particular label, this multiway synchronisation permits us to represent the broadcasting of a value to a number of different programs.

4.3 Other Examples

Memories.

We can construct memories out of registers. To do this we must give each register a separate identity. Let REG^i be of sort $\{\varepsilon^i_1, \varepsilon^i_0, \sigma^i_1, \sigma^i_0\}$ and so let it be defined as for REG but with label changes. Then define MEM a memory of sort $\bigcup\limits_{i=1,n} \{\varepsilon^i_1, \varepsilon^i_0, \sigma^i_0, \sigma^i_1\}$

$$MEM = REG^1 \bullet REG^2 \bullet \cdots \bullet REG^{n-1} \bullet REG^n \ \underline{\text{where}} \cdots .$$

An operation to produce instances of a generic program, will be introduced later. As the REG's in MEM have disjoint sorts they are not connected by the concurrency operation.

Stacks

We saw that a register was defined using an identifier parameterised on a tuple (or string) or booleans. A stack of sort $\{\delta_0, \delta_1, \sigma_0, \sigma_1\}$ may be defined by:

$$STACK(\varepsilon) = \delta_\theta \ STACK(0 \frown \varepsilon) + \delta_1 \ STACK(1 \frown \varepsilon)$$

$$\underline{\text{where}} \quad STACK(0 \frown n) = \sigma_0 \ STACK(n) + \delta_0 \ STACK(0 \frown 0 \frown n) + \delta_1 \ STACK(1 \frown 0 \frown n)$$

$$\underline{\text{and}} \quad STACK(1 \frown n) = \sigma_1 \ STACK(n) + \delta_0 STACK(1 \frown 0 \frown n) + \delta_1 STACK(1 \frown 1 \frown n)$$

Here ε is the empty string. Note that we do capture the behaviour of a stack here as 1's and 0's are "put" on and "taken" off the top of the stack. A queue may be defined similarly but we "put" on a different end of the string from where we "take" off.

<u>Counters</u>

A counter COUNT(i), parameterised on integer i, will have sort $\{$up, down, zero$\}$. As we do not (yet!) have a conditional construct in our language we define COUNT(i) by use of two clauses

$$\text{COUNT}(0) = \text{zero COUNT}(0) + \text{up COUNT}(1)$$

<u>and</u>

$$\text{COUNT}(n+1) = \text{down COUNT}(n) + \text{up COUNT}(n+2)$$

This counter keeps track of the number of ups that exceed the number of downs.

An alternative counter may count negatives as well. Let us call this program COUNTER(i) where i is the number of ups minus the number of downs. It will have sort $\{$up, down$\}$.

$$\text{COUNTER}(n) = \text{up COUNTER}(n+1) + \text{down COUNTER}(n-1)$$

It is the environment which generates the up and down stimuli and the COUNTER program will cooperate with whatever is in the environment to synchronise on a stimulus and evolve to a new program. We may wish the environment to interrogate the counter and discover its contents. As we have no value-passing mechanism in our language (unlike CSP, for instance) we have problems in getting the value of the contents "out." If we allow our counter to be bounded then we can have an out label for each integer. That is, a separate output line corresponds to each integer and if a synchronisation is made on one of these lines by some other program we may assume that this program now knows the contents of the counter.

Suppose our counter only counts positively, as COUNT(i), and has a maximum of m. Then COUNTM(i) will have sort $\{$contents, up, down, $\text{out}_0, \text{out}_1, \cdots, \text{out}_m\}$ and is defined by:

$$\text{COUNTM}(0) = \text{contents } \text{out}_0 \text{ COUNTM}(0) + \text{up COUNTM}(1)$$

<u>and</u>

$$\text{COUNTM}(m) = \text{contents } \text{out}_m \text{ COUNTM}(m) + \text{down COUNTM}(m-1)$$

<u>and</u>

$$\begin{aligned}\text{COUNTM}(m-1) = \text{contents } \text{out}_{m-1} &\text{ COUNTM}(m-2)\\ &+ \text{up COUNTM}(m)\\ &+ \text{down COUNTM}(m-2)\end{aligned}$$

The contents guard is really redundant since the interrogating program which we would compose with COUNTM using ● must be able to synchronise on all of $\text{out}_0, \cdots, \text{out}_m$ so that it will know the contents. Leaving the contents guard out has the same effect. We do not need a special label for zero contents; out_0 is suitable.

The language may be extended to allow values to be communicated whenever a synchronisation takes place. This opens up a much larger class of examples which can be easily programmed. As we are interested in the synchronisation and nondeterministic aspects of our language, we omit the value-passing features for the meantime.

4.4 The Dining Philosophers Problem

In this example we have two types of computing agents, philosophers and forks. We have the same number of philosophers and forks laid out around a table with philosophers and forks alternating. A philosopher is allowed to access only the forks on either side. The "problem" which this example illustrates is described as follows: to enable a philosopher to eat he must be in possession of both his neighbouring forks and the forks can be obtained in either order. Unfortunately this may cause a deadlock situation in which all philosophers have "picked up" their left-hand (or all their right-hand) forks. This means that all forks are accessed and no philosopher is able to obtain the two forks he needs to enable himself to eat; all philosophers starve. This is a problem involving shared resources, each fork being shared between two philosophers.

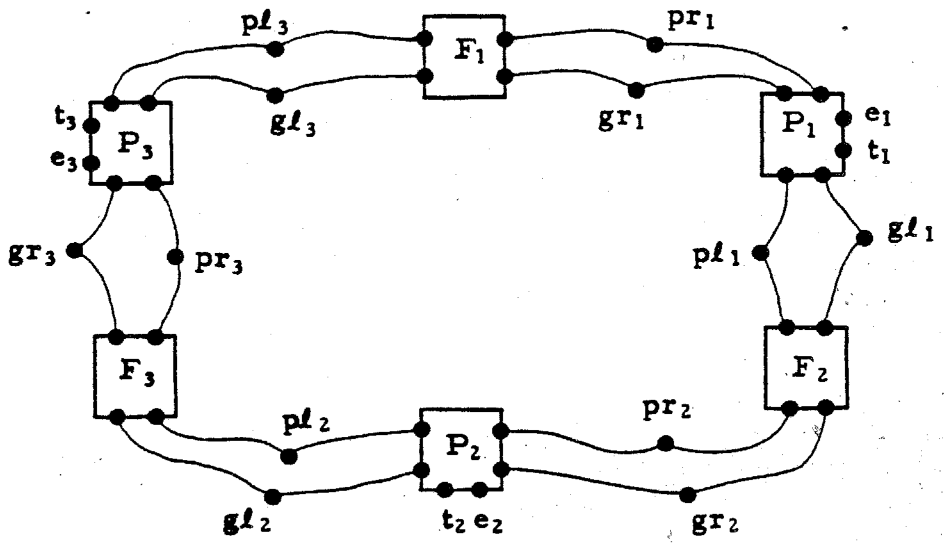
We shall program a "solution" to this "problem" in our language. The solution provides for philosophers to access the forks in such a way that the system does not deadlock. Our solution is not fair however; some philosophers may be prevented from accessing both forks forever and so starve. Fairness questions are outside the remit of our language and we believe that fairness is an implementation issue and so does not concern us. For instance, to make our dining philosopher's program fair, a centralised scheduler may be introduced to control the order in which philosophers access forks. Many algorithms can be adopted by this scheduler to ensure fairness.

A centralised scheduler, or controller, may be used to ensure the absence of a deadlock without even considering fairness. In [4] Hoare uses a "room" as a centralised controller. This room controls the number of philosophers active, or present, at any given instant. As long as the number of active philosophers is one less than the number of forks then the system will not deadlock.

We wish a distributed solution, that is, the behaviour of the philosophers and forks themselves should be such that when they interact the system as a whole does not deadlock.

Let us program this for a system of three philosophers and three forks. It is easy to see how the system can be extended to involve n philosophers and n forks.

The agents will be interlinked as follows.



The philosophers wish to pick up the forks on either side of them in either order, eat, then put the forks down in either order, then think and so on. Synchronisation via the gℓ and gr ports represents interaction between a philosopher and the left or right fork to pick the fork up while e and t ports are used by the philosopher programs to represent their desire to eak and think. The pr and pℓ ports are used to synchronise the action of placing down the left or right forks. This again, is an interaction between a fork and a philosopher.

The behaviour of the ith philosopher can be represented by the program P_i of sort $\{p\ell_i, g\ell_i, pr_i, gr_i, t_i, e_i\}$ defined by

$$P_i = g\ell_i\, gr_i\, P_i' + gr_i\, g\ell_i\, P_i'$$

<u>where</u>

$$P_i' = e_i(p\ell_i\, pr_i\, t_i\, P_i + pr_i\, p\ell_i\, t_i\, P_i)$$

The behavious of the forks is given by

$$F_i = gr_i\, pr_i\, F_i + g\ell(I \ominus 1)\ p\ell(i \ominus 1)\ F_i$$

Here $\ominus$ is subtraction modulo n, where n is the number of forks (and philosophers) in the system.

We see that the philosopher asks for both forks in either order and then eats. The forks are placed down in either order, he thinks and so on. A fork can be picked up by either of the philosophers on each side of it. When it is picked up this prevents the other philosopher gaining the fork until it has been placed down again.

For n = 3 we have a system pictured as above with

$$SYS = P_1 \bullet P_2 \bullet P_3 \bullet F_1 \bullet F_2 \bullet F_3$$

Here P_1 is P_i with the guards all relabelled by changing the index. This is performed using the $[\alpha/\beta]$ operator and produces instances of philosopher programs from the generic philosopher P_i. Forks are treated similarly.

We may apply axiom ($\bullet$ +) a number of times to SYS and we discover that one of the subprograms produced is

$$g\ell_1 \, g\ell_2 \, g\ell_3 \Delta$$

hence the system <u>may</u> deadlock. Other summands also result in the appearance of Δ whilst others do not. The summand above indicates that all of the forks have been picked up by the philosopher's left hands so preventing any philosopher gaining both forks and so preventing any philosopher from eating.

To prevent this we change the philosopher's behaviour so that before starting to pick up and place down the forks the philosopher reserves both forks and so prevents the philosophers which share his forks from gaining access to them. We introduce r_i guards on philosopher P_i and his neighbouring forks. To synchronise on this guard (effected by the $\bullet$ operator) the philosopher and both his forks must cooperate in a three-way synchronisation to reserve both forks.

We redefine our philosopher and fork programs to include this reservation guard. As the eating and thinking guards do not influence whether the system deadlocks or not, we will omit them. We now have program P_i of sort $\{r_i, g\ell_i, gr_i, p\ell_i, pr_i\}$

$$P_i = r_i\,(g\ell_i\,gr_i\,P_i' + gr_i\,g\ell_i\,P_i')$$

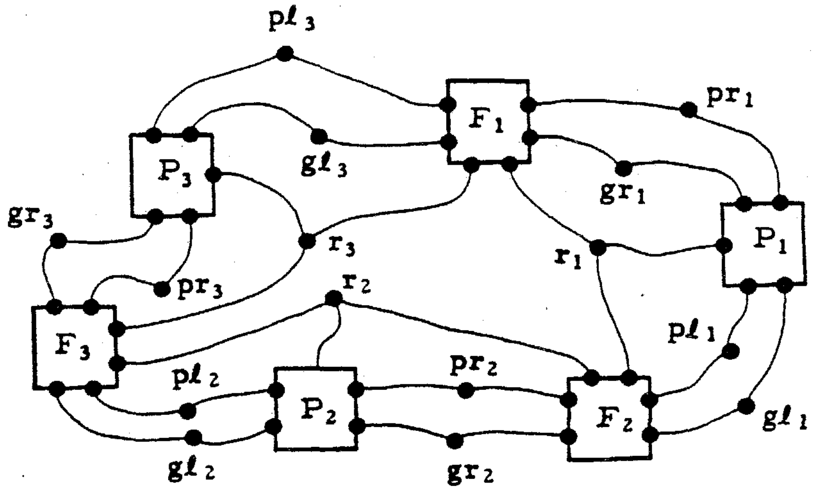
<u>where</u>

$$P_i' = p\ell_i\,pr_i\,P_i + pr_i\,p\ell_i\,P_i$$

and program F_i of sort $\{r_i,\ r_i \ominus 1,\ gr_i,\ pr_i,\ g\ell_{i\ominus 1}, p\ell_{i\ominus 1}\}$

$$F_i = r_i\,gr_i\,pr_i\,F_i + r_{i\ominus 1}\,g\ell_{i\ominus 1}\,p\ell_{i\ominus 1}\,f_i$$

Our constructed system SYS = $P_1 \bullet P_2 \bullet P_3 \bullet F_1 \bullet F_2 \bullet F_3$ is again built out of instances of the above generic programs. To illustrate where synchronisation may take place between the components we picture SYS as follows



We may now exhaustively use axiom [● +] to expand SYS. We can see that we do not obtain subprograms which terminate (with Δ) and so our system does not deadlock.

A problem here is that the expansion via [● +] is quite tiresome and soon produces a program of unmanageable complexity. We are soon unsure whether we have missed out some subprograms or not. For three philosophers and three forks, we can just about manage. We can progress far enough to see that we do not get subprograms similar to the $g\ell_1\, g\ell_2\, g\ell_3\, \Delta$ which we get in the original system, but a much larger expansion is needed to convince ourselves that Δ does not arise at all.

If our system contained a larger number of philosophers and forks than three then an expansion using axiom [● +] to check for the presence of Δ would be impossible. We would eve wish to show such a system free from deadlock for n philosophers and forks, for all n.

A methodology for this will be introduced in a later paper. It utilises the rigorous structure of interconnection among philosophers and forks and allows us to prove the absence of Δ by induction on n, the number of components in the system.

5. DEADLOCK AND TERMINATION

We introduce the nullary operator Δ as the identity of operator $+$, and so also the empty sum $\sum$. It has previously been mentioned that we are taking Δ to represent, in our language, the deadlock phenomena. Deadlock is a system property that exists when all the components of the system are mutually waiting for each other to perform some action which must take place before they can proceed. A classic example of this appears in the dining philosophers problem where a philosopher must access two resources, called forks, which he shares with different philosophers before he can proceed to eating. Deadlock arises when we have shared resources; that is, we have competition among agents which with to interact with a resource agent. Thus Δ may arise due to the definition of the $\bullet$ operator.

Suppose we have the programs P and Q with sorts $\{\alpha, \beta, \gamma\}$ and $\{\alpha, \beta\}$ respectively, which are defined by

$$P = \gamma\ \alpha\ B\ P_1 + \gamma\ P_2$$

$$Q = \beta\ \alpha\ Q_1$$

and we leave subprograms P_1, P_2 and Q_1 unspecified for the present. By axiom $[\bullet +]$ we have that

$$P \bullet Q = \gamma\,\Delta + \gamma(P_2 \bullet Q)$$

The α and β labels are internal while the γ label is external to $P \bullet Q$. As P wishes to perform a γ symchronisation with the environment and as γ is external then P can evolve to program $\alpha \beta P_1$ without any cooperation from Q. Now as α and β are both internal labels program $\alpha \beta P_1$ must receive an α stimulus from Q for a synchronisation to take place. But program Q wishes a β stimulus before a synchronisation can take place. As nothing else can happen $\alpha\beta P_1$ and Q become deadlocked so $\alpha\beta P_1 \bullet Q = \Delta$.

P may of course, on a receipt of a γ stimulus from the environment, proceed to program P_2. Suppose we define P_2 as

$P_2 = \beta \alpha P$ then by axiom $[\bullet +]$ we have that

$$P_2 \bullet Q = \beta \alpha (P \bullet Q_1)$$

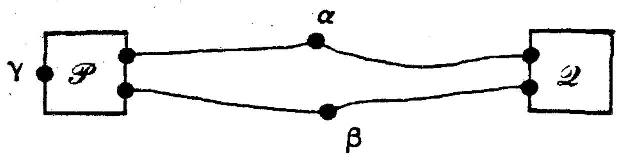
So after an external γ we have that synchronisation on the β followed by α labels takes place and the original programs P and Q have evolved to P and Q_1 respectively.

We now have that $P \bullet Q$ gives program

$\gamma\Delta + \gamma\beta\alpha(P \bullet Q_1)$ using axiom $[\bullet +]$ repeatedly.

Following one of the γ guards we get P and Q (actually P has evolved to $\alpha \beta P_1$) becoming deadlocked whilst following the other γ guard deadlock does not result and β and α synchronisations take place.

Our system composed of P and Q is pictured as follows:



In one case, following a γ, we have that "machine" $\mathscr{P}$ wishes to interact with $\mathscr{Q}$ on the α line whilst machine $\mathscr{Q}$ wishes to interact with $\mathscr{P}$ on the β line. Neither of these wishes may be satisfied and so Δ results. In the other case both $\mathscr{P}$ and $\mathscr{Q}$ wish to first of all interact, i.e., exchange stimuli, on the β line followed by an interaction on the α line. This can proceed via β and α *synchronisations* with the behaviour of machine $\mathscr{P}$ evolving to P and that of machine $\mathscr{Q}$ evolves to the behaviour represented by program Q.

In this example no separate resource program is competed for but programs P and Q can be thought of as resources as viewed by programs Q and P respectively. Deadlock results when both wish to access each other. If the α label is interpreted as "access Q" and β as "access P" then if P wishes to access Q while Q wishes to access P then neither P or Q is available to the other as a resource and we get deadlock. If, on the other hand, P wishes to synchronise on β, that is, it is offering itself as a resource to Q and Q also wishes to synchronise on β, that is, access P, then a β synchronisation is successfully performed and we do not have deadlock.

The above indicates how Δ gets introduced into programs as the result of applying the ● operator. We may also use Δ as the termination operator; the "good" termination operator where deadlock can be considered as "bad" *termination*. A program R which wishes to receive an α stimulus followed by a β stimulus and then successfully terminate in defined by

$$R = \alpha \beta \Delta$$

When we compose a terminated program Δ with some other, say program S, then Δ will cause $\Delta \bullet S$ to deadlock, possibly following a number of external guards, unless all of the guards in S are external to $\Delta \bullet S$.

As an example consider Δ of sort $\{\alpha\}$ and S or sort $\{\alpha, \beta, \gamma\}$ then if S is defined by

$$S = \beta \gamma S + \gamma S$$

we have $\Delta \bullet S$ giving program

$$\beta\gamma(\Delta \bullet S) + \gamma(\Delta \bullet S) \quad \text{of sort } \{\alpha, \beta, \gamma\}$$

which never produces Δ and so, by repeated use of axiom $[\bullet +]$ never terminates. Suppose Δ is as above but T of sort $\{\alpha, \beta, \gamma\}$ is defined by

$$T = \alpha \gamma T + \gamma T$$

then $\Delta \bullet T$ gives program $\gamma(\Delta \bullet T)$ by use of axiom $[\bullet +]$. Again, Δ does not result as an externally labelled γ guard can always appear.

But suppose we replace T by program U which is identical except that the + operator is replaced by $\oplus$ then

$$U = \alpha\, \gamma\, U \oplus \gamma\, U$$

and

$$\begin{aligned} \Delta \bullet U &= (\Delta \bullet (\alpha\gamma U)) \oplus (\Delta \bullet \gamma U) && \text{by axiom } [\bullet\ \oplus] \\ &= \Delta \oplus \gamma(\Delta \bullet U) && \text{by axiom } [\bullet\ +] \text{ twice} \end{aligned}$$

So program $\Delta \bullet U$ nondeterministically may deadlock or else react to a γ stimulus (of one were available from the environment) to evolve back into program $\Delta \bullet U$. Thus following the successful receipt of a γ stimulus again we have that deadlock may result and so on.

Two terminated programs when composed obviously give us deadlock, due again to axiom $[\bullet\ +]$. Note that the sort of the "resulting" Δ is the union of that of the two components

$$\Delta \bullet \Delta = \Delta$$

This is a theorem derived from axiom $[\bullet\ +]$.

Our language manipulates the representation of deadlock and termination in a manner corresponding to the behaviour of real systems. We use only one symbol to represent both deadlock and termination since in many ways they model the same phenomena.

In conclusion we have termination as a "wholesome" feature and a property of one computing agent and so of one program. Deadlock appears when we have two or more agents present and is thus a property of two or more interacting programs.

6. HIDING

6.1 Why We Require Hiding

We can define a queue by

$$Q(\varepsilon) = \delta_0 Q(0) + \delta_1 Q(1)$$

where

$$Q(n^\frown 1) = \sigma_1 Q(n) + \delta_0 Q(0^\frown n^\frown 1) + \delta_1 Q(1^\frown n^\frown 1)$$

and

$$Q(n^\frown 0) = \sigma_0 Q(n) + \delta_0 Q(0^\frown n^\frown 0) + \delta_1 Q(1^\frown n^\frown 0)$$

We assume that ε is the empty string and $\varepsilon ^\frown i = i ^\frown \varepsilon = i$. If we define another queue we would like to be able to ajoin them and produce a new queue.

Let us introduce a relabelling operator into our language. It is not strictly necessary but it allows us to produce instances of generic programs without the need to rewrite them

The post fixed operator $[\alpha/\beta]$ when applied to a program p changes each β label up to an α label. All other labels remain unchanged. We must ensure the α is not in the sort of p.

The following two axioms are sufficient to express the intended meaning of this relabelling operator:

$$[/+] \qquad \left(\sum_i \alpha_i p_i\right)[\beta/\alpha] = \left(\sum_{\alpha_i=\alpha} \beta(p_i[\beta/\alpha])\right) + \left(\sum_{\alpha_i \neq \alpha} \alpha_i(p_i[\beta/\alpha])\right)$$

where β is not a member of the sort of $\sum_i \alpha_i p_i$

It is clear how this operator behaves. The acceptance semantics for programs constructed using this relabelling operator is not given and is left as an exercise for the reader. We shall not prove the consistency of these two axioms. This, together with the definition of other axioms relating $[/]$ to the Δ, and $-\alpha$ operators, can be performed by the reader. Note that $[/]$ can change the sort of a program. If p has sort L then $p[\beta/\alpha]$ has sort $(L \cup \{\beta\}) - \{\alpha\}$. Here we assume that $\beta \notin L$.

As an example consider $(\alpha p_1 + \beta p_2)[\gamma/\alpha]$. This gives $\gamma p_1[\gamma/\alpha] + \beta p_2[\gamma/\alpha]$ where we replace all occurrences of α by γ. The operator $[\gamma/\alpha]$ is recursively applied to the renewal programs p_1 and p_2 so replacing all occurrences of α by γ in the whole program.

Now let us redefine our queue to give it a maximum size:

$$QN(0, \varepsilon) = \delta_0 QW(1,0) + \delta_1 QW(1,1)$$

<u>where</u>

$$QN(i, S^\frown 1) = \sigma_1 QN(i-1, S) + \delta_0 QN(i+1, 0^\frown S^\frown 1) + \delta_1 QN(i+1, 1^\frown S^\frown 1)$$

<u>and</u>

$$QN(i, S^\frown 0) = \sigma_0 QN(i-1, S) + \delta_0 QN(i+1, 0^\frown S^\frown 0) + \delta_1 QN(i+1, 1^\frown S^\frown 0)$$

<u>and</u>

$$QN(N, S^\frown 1) = \sigma_1 QN(N-1, S) \text{ \underline{and} } QN(N, S^\frown 0) = \sigma_0 QN(N-1, S)$$

Then $Q' = QN(\varepsilon)[\alpha_1/\sigma_1, \alpha_0/\sigma_0]$ is as for $QN(\varepsilon)$ but has the output renamed by α. Similarly, $Q'' = QM(\varepsilon)[\alpha_1/\delta_1, \alpha_0/\delta_o]$ is as for $QM(\varepsilon)$ but has the inputs renamed by α. We can now join up Q' and Q'' using our concurrency operation to get $SYS = Q' \bullet Q''$ which should behave as the single queue $Q(M+N)(\varepsilon)$ except that it contains an α connector which allows other programs to synchronise through it. We wish, after Q' and Q'' have been composed, to hide the α guards and so "internalise" them. The α is then internal to Q' and Q'' and cannot be provided with stimulus from without, i.e., the environment.

As another example of hiding take a binary semaphore. We may define this by

$$SEM = \sum_{i \leq i \leq n} p_i v_i \, SEM$$

where $S \epsilon M$ has sort $\bigcup_{1 \leq i \leq n} \{p_i, v_i\}$. Suppose we have two agents A and B who access a resource using α and β labels respectively and we wish A to mutually exclusively send an α stimulus followed by another α stimulus to the resource. Similarly B should generate a pair of β stimuli. Agent A would be defined as A = αA (and B similarly) but we add p and σ guards to their behaviour to allow the semaphore to control them. We can therefore redefine the agents to be

$$A = p_1\alpha\ \alpha\ v_1 A$$

$$B = p_2\beta\ \beta\ v_2\ B$$

The p_1, v_1 and p_2, v_2 labels guard the critical sections $\alpha\alpha$ and $\beta\beta$ respectively and these sections must send stimuli to the resource mutually exclusively.

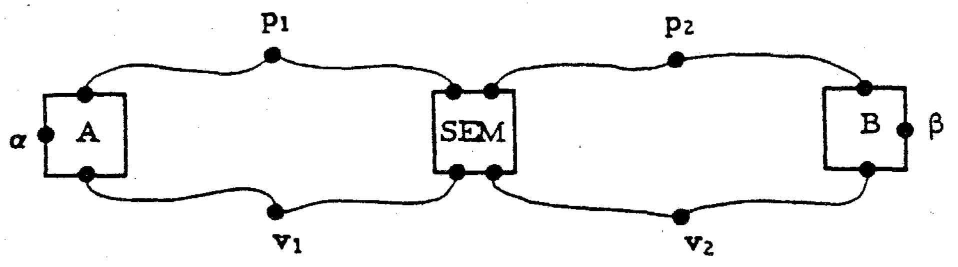
The constructed system is then:

$$\text{CONSYS} = A \bullet B \bullet \text{SEM}$$

where

$$\text{SEM} = p_1 v_1 \text{SEM} + p_2 v_2\ \text{SEM}$$

This can be pictured as follows when the programs are treated as machines



Now we wish the semaphore to control only how A and B access the resource (which is in the environment!) so we would wish to hide the p_1, p_2, v_1, and v_2 ports (actually they are connectors) and prevent further programs attaching onto them. We would like the above picture to have the p_1, p_2, v_1 and v_2 labels removed.

The operation $-\alpha$ when applied to a program hides the α guards but we wish that it leaves the rest of the behaviour of the program unchanged.

We introduce the following axioms to define hiding:

$$[-+_1]\ (\sum \alpha_i p_i)-\alpha = \Big(\sum_{\alpha_i \neq \alpha} \alpha_i(p_i-\alpha) + \sum_{\alpha_i = \alpha} (p_i-\alpha)\Big) \oplus \sum_{\alpha_i = \alpha} (p_i-\alpha)$$

$$[-\oplus]\ (\textstyle\bigoplus p_i)-\alpha = \bigoplus (p_i-\alpha)$$

As an example of the first axiom consider the following:

$$(\beta p_1 + \alpha p_2)-\alpha = (\beta(p_1-\alpha) + (p_2-\alpha)) \oplus (p_2-\alpha)$$

Here we assume α is internal to $\beta p_1 + \alpha p2$. If not then it is treated differently and how we do this is given later. As it is internal, the α guard represents the result of a synchronisation. Thus when hidden we do not know whether it may take place or not.

If a β stimulus comes from the environment then two things may happen; the β may be accepted or the internal α synchronisation may take place and prevent the β guard from receiving a stimulus. The $\oplus$ operator introduces this nondeterministic behaviour.

But suppose the environment does not produce a β stimulus then something may always happen; the internalised α synchronisation. That is why we use the + operator to compose the result of hiding the synchronisation with the guards that remain unchanged. The hiding operation is applied recursively to the programs that follow the guarding labels, both the hidden and unchanged ones.

How does this hiding work when applied to our semaphore example; that is, what program results? Let us use the [● +] axiom a number of times to expand CONSYS. Of course we could keep on applying CONSYS indefinately since none of the constituent programs of CONSYS terminate (with Δ) and when composed, Δ never results.

$$\text{CONSYS} = p_1((\alpha\alpha v_1 A) \bullet (p_2\beta\beta v_2 B) \bullet (v_1 SEM))$$

$$+ \; p_2((p_1\alpha\alpha v_1 A) \bullet (\beta\beta v_2 B) \bullet (v_2 SEM))$$

by [● +]

$$= p_1\alpha\alpha((v_1 A) \bullet (p_2\beta\beta v_2 B) \bullet (v_1 SEM))$$

$$+ \; p_2\beta\beta((p_1\alpha\alpha v_1 A) \bullet (v_2 B) \bullet (v_2 SEM))$$

by [● +] 4 times

$$= p_1\alpha\alpha v_1(A \bullet B \bullet SEM) + p_2\beta\beta v_2(A \bullet B \bullet SEM)$$

by [● +] twice

$$= p_1\alpha\alpha v_1 \text{ CONSYS} + p_2\beta\beta v_2 \text{ CONSYS}$$

The presence of the p's and v's prevents the α and β pairs interleaving which is what we require but we wish the p's and v's to be internalised. We shall first of all hide out p_1 and see what we get:

$$\mathrm{CONSYS} - p_1 = \Big(((\alpha\alpha v_1 \mathrm{CONSYS}) - p_1) + p_2((\beta\beta v_2 \mathrm{CONSYS}) - p_1)\Big)$$

$$\oplus\ (\alpha\alpha v_1 \mathrm{CONSYS}) - p_1 \qquad \text{by } [+-]$$

$$= \alpha\alpha v_1(\mathrm{CONSYS}-p_1) + p_2\beta\beta v_2(\mathrm{CONSYS}-p_1)$$

$$\oplus\ \alpha\alpha v_1(\mathrm{CONSYS}-p_1) \qquad \text{by } [\oplus-] \text{ and } [+-]$$

$$\mathrm{CONSYS}-p_1-p_2 = \Big(\alpha\alpha v_1(\mathrm{CONSYS}-p_1-p_2) + \beta\beta v_2(\mathrm{CONSYS}-p_1-p_2)\Big)$$

$$\oplus\ \beta\beta v_2(\mathrm{CONSYS}-p_1-p_2)\ \oplus \alpha\alpha v_1(\mathrm{CONSYS}-p_1-p_2)$$

$$\mathrm{CONSYS}-p_1-p_2-v_1-v_2 = \Big(\alpha\alpha(\mathrm{CONSYS}-p_1-p_2-v_1-v_2)$$

$$+\ \beta\beta(\mathrm{CONSYS}-p_1-p_2-v_1-v_2)\Big)$$

$$\oplus\ \beta\beta(\mathrm{CONSYS}-p_1-p_2-v_1-v_2)$$

$$\oplus\ \alpha\alpha(\mathrm{CONSYS}-p_1-p_2-v_1-v_2)$$

$$= \alpha\alpha(\mathrm{CONSYS}-p_1-p_2-v_1-v_2)$$

$$\oplus\ \beta\beta(\mathrm{CONSYS}-p_1-p_2-v_1-v_2) \qquad \text{by } [\oplus\oplus+]$$

Here we can see that the α and β pairs are uninterleaved as required. The nondeterminism operator is introduced since the environment after hiding has no control over how the semaphore controls the agents. The semaphore forces β's to the exclusion of α's and α's to the exclusion of β's and as the semaphore is now abstracted away so we have $\oplus$ introduced.

Hiding usually introduces $\oplus$ when applied to CNF programs and as explained above, this is to be expected. But suppose in our semaphore example we want to control the action of the agents A and B so that α and β pairs are not interleaved and we also wish that the shared resource (which may be composed on to the controlled system later) should have the ability to choose an α pair or a β pair. We do not have this when hiding p's and v's in CONSYS, as above, since the hiding introduces $\oplus$ and we now would require the two subprograms to be separated by +.

We can design a semaphore to synchronise directly on the agents α and β guards to produce a new constructed system NSYS which does not require the removal of additional guards. We require that NSYS = $\alpha\alpha$NSYS + $\beta\beta$NSYS.

If our semaphore is defined as

$$\mathrm{SEM} = \alpha\alpha\mathrm{SEM} + \beta\beta\mathrm{SEM}$$

and our agents are not altered by the addition of p and v guards, that is, they are defined by

$$A = \alpha A \quad \text{and} \quad B = \alpha B$$

then our constructed system is

$$\mathrm{SEM} \bullet A \bullet B = \alpha\alpha(\mathrm{SEM} \bullet A \bullet B) + \beta\beta(\mathrm{SEM} \bullet A \bullet B)$$

and so SEM $\bullet$ A $\bullet$ B = NSYS as required. Note that SEM, NSYS, and SEM $\bullet$ A $\bullet$ B are all identical programs, that is, they have the same meaning or behaviour.

6.2 The Hiding Semantics

Using our acceptor semantics we now formalise the meaning of our hiding operation. Let us assume that we are hiding <u>internal</u> labels. The external label hiding will be dealt with later.

We introduce the following notation:

$$(p, \alpha) = \{*\} \quad \text{for} \quad \forall q \cdot (p, \alpha) \rightarrow q \Rightarrow q = *$$

15. $$\frac{[(p, \beta) = \{*\}] \wedge [(p, \alpha) = \{*\}] \wedge [\alpha \in L_{pINT}] \wedge [p \text{ in CNF}]}{(p-\alpha, \beta) = \{*\}}$$

16. $$\frac{(p, \beta) \rightarrow p'}{(p-\alpha, \beta) \rightarrow p'-\alpha}$$

17.1 $$\forall_{n \geq 1} \frac{[(p, \alpha) \rightarrow p_1] \wedge [(p_1, \alpha) \rightarrow p_2] \wedge \cdots \wedge [(p_n, \beta) \rightarrow p'] \wedge [\alpha \in L_{pINT}] \ [p \wedge p_n \text{ in CNF}]}{(p-\alpha, \beta) \rightarrow p'-\alpha}$$

17.2 $$\frac{\forall_{n \geq 1} [(p, \alpha) \rightarrow p_1] \wedge [(p_1, \alpha) \rightarrow p_2] \wedge \cdots \wedge [(p_n, \beta) = \{*\}] \wedge [(p_n, \alpha) = \{*\}] \wedge [\alpha \in L_{pINT}] \wedge [p \wedge p_n \text{ in CNF}}{(p-\alpha, \beta) = \{*\}}$$

17.3 $$\frac{\forall_{n \geq 1} [(p, \alpha) \rightarrow p_1] \wedge [(p_1, \alpha) \rightarrow p_2] \wedge \cdots \wedge [(p_n - \alpha, \beta) \rightarrow *] \wedge [\alpha \in L_{pINT}] \wedge [p \wedge p_n \text{ in CNF}]}{(p-\alpha, \beta) \rightarrow *}$$

18. $$\frac{[(p, \beta) = *] \wedge [\alpha \in L_{p\ EXT}] \wedge [p \text{ in CNF}]}{(p-\alpha, \beta) = \{*\}}$$

19. $$\frac{(p-\alpha,\beta)\rightarrow *}{((p\oplus q)-\alpha,\beta)\rightarrow *}$$

20. $$\frac{(q-\alpha,\beta)\rightarrow *}{((p\oplus q)-\alpha,\beta)\rightarrow *}$$

Condition 15 states that if program p always produces * on an α stimulus and it always produces * on a β stimulus then program p-α always produces * on a β stimulus. As p is in CNF, $(p,\beta)\rightarrow * \Longleftrightarrow (p,\beta) = \{*\}$. For $p \oplus q$, i.e., a program in DNF, then condition 19 says that if p-α gives * on a β than so also does $(p \oplus q)$-α. We treat CNF and DNF programs differently here since for DNF r, $(r,\beta)\rightarrow * \not\Longleftrightarrow (r,\beta) = \{*\}$, unlike a CNF program.

Condition 16 states that even if program p can accept an α then provided p accepts a β and produces p' then p-α also accepts a β and it produces the program p-α. This holds for α being both internal and external.

If p produces * on a β then we do <u>not</u> have p-α producing * on a β since the program q say, which we get when p accepts the α, may accept a β or produce a program whose offspring may accept a β. It is only when a β is not accepted by p or any of its offspring which result from some number of α stimuli (possibly one) being forced at it, do we get p-α giving an * on receipt of a β. 17.2 does this for CNF p_n and 17.3 does this for DNF p_n. The need for both 17.2 and 17.3 is the same as for 15 and 19 (and 20).

Condition 17.1 states that if after some number n of α stimuli (possibly one) program p evolves into program p_n which produces p' on a β stimulus, then p-α produces p'-α on a β stimulus.

As we are hiding internal labels, then α's appear in q due to a synchronisation on α's taking place. When we hide them we wish to preserve the behaviours which result from such α synchronisations. Hence the behaviour which occurs after α synchronisation in p, i.e., the β stimulus on condition 17.1, should also occur in p-α.

The difference between internal and external guards is given in the next chapter but we can note here that condition 18 states that for external α and p in CNF, if p gives * then so also does p-α. It is similar to 15 except we ignore whether α's can be accepted or not, as α is external.

Conditions 19 and 20 indicate how we get * after hiding on DNF programs. 16 applies to both CNF and DNF p and so indicates how a program results when a hidden DNF program receives a stimulus.

To see how the semantics works for hiding let us try some examples:

(1) $((\gamma p + \beta q)\text{-}\alpha, \gamma) \rightarrow p\text{-}\alpha$ by 16 as $(\gamma p + \beta q, \gamma) \rightarrow p$

$((\gamma p + \beta q)\text{-}\alpha, \beta) \rightarrow q\text{-}\alpha$ similarly

(2) $((\alpha\gamma p + \beta q)\text{-}\alpha, \gamma) \rightarrow p\text{-}\alpha$ by 17.1 as $(\alpha\gamma p+\beta q) \rightarrow \gamma p$, $(\gamma p, \gamma) \rightarrow p$.

$((\alpha\gamma p + \beta q)\text{-}\alpha, \beta) \rightarrow q\text{-}\alpha$ by 16 as $(\alpha\gamma p + \beta q, \beta) \rightarrow q$

$((\alpha\gamma p + \beta q)\text{-}\alpha, \beta) \rightarrow *$ by 17.2 as γp is in CNF and $(\alpha\gamma p + \beta q, \alpha) \rightarrow \gamma p$ and $(\gamma p, \beta) \rightarrow *$.

and if δ is in the sort of our program then:

$((\alpha(\gamma p \oplus \delta r) + \beta q)\text{-}\alpha, \delta) \rightarrow *$ by 17.3 as

$((\alpha\gamma p + \beta q), \alpha) \rightarrow (\gamma p \oplus \delta r)$

and $(\gamma p, \delta) \rightarrow *$ giving

$((\gamma p \oplus \delta r)\text{-}\alpha, \delta) \rightarrow *$ by 19.

6.3 Internal and External Guards

We wish to be able to distinguish between guards, i.e., labels, which are produced by the concurrency operator ● when two or more programs synchronise, and those which do not. The former are internal guards and the latter are external guards.

To do this we change the notion of sort from being a set of labels to a pair of sets of labels. The first set are external labels and the second set are the internal labels. These two sets are disjoint and when unioned together produce our previous notion of sort.

Initially most of the sort labels will be external unless we wish to designate some as internal. Our operators are also sorted and will now be as follows:

$$\alpha \ : \langle L_1, L_2\rangle \rightarrow \langle L_1, L_2\rangle$$

$$\bullet \ : \langle L_1, L_2\rangle \times \langle M_1, M_2\rangle \rightarrow \langle L_1 \cup M_1 - P, L_2 \cup M_2 \cup P\rangle$$

$$\text{where } P = (L_1 \cup L_2) \cap (M_1 \cup M_2)$$

$$-\alpha \ : \langle L_1, L_2\rangle \rightarrow \langle L_1 - \{\alpha\}, L_2 - \{\alpha\}\rangle$$

$$+ \ : \langle L_1, L_2\rangle \times \langle L_1, L_2\rangle \rightarrow \langle L_1, L_2\rangle$$

$$\oplus \ : \langle L_1, L_2\rangle \times \langle L_1, L_2\rangle \rightarrow \langle L_1, L_2\rangle$$

$$[\beta/\alpha] : \langle L_1, L_2\rangle \rightarrow \langle (L_1 \cup \{\beta\}) - \{\alpha\}, (L_2 \cup \{\beta\}) - \{\alpha\}\rangle$$

It is the concurrent composition which produces new internal guards.

We have seen that hiding internal guards preserves the program which follows the guard (with hiding occurring in it as well). The reason for this is that an internal guard results from a synchronisation and we wish to preserve that behaviour which follows from this synchronisation. A synchronisation is an interaction among programs and once hidden we do not know whether it will occur or not and so need to allow for all possibilities; one is that it does occur and so the acceptance behaviour which occurs after the synchronisation should also occur in the hidden program.

But suppose we hide an external guard, one which does not appear due to a synchronisation having been effected. Then it does not make sense to allow the hidden program to accept in the same way as for internal guards. Since labels are used as "synchronisation points" an external guard has labels which have not been used in synchronisation.

Thus an interaction has not occurred on an external guard and when hidden the following program that which is guarded by the "to be hidden" label is also lost.

The hiding here prevents entrance to the program which is protected by the guard. We therefore define hiding for external label α by

$$\left(\sum \alpha_i p_i \right) - \alpha = \sum_{\alpha \neq \alpha_i} \alpha_i \, (p_i - \alpha)$$

and

$$\left(\textstyle\bigoplus p_i \right) - \alpha = \textstyle\bigoplus ((p_i) - \alpha)$$

and we see that, compared to the definition for internal labels, we do not include the (hidden versions of the) subprograms which are guarded by the α label.

Our semantics for external hiding is given by conditions 16 and 18 above, for successful and unsuccessful receipt of a stimulus respectively.

<u>Examples</u>

(i) $(\alpha p - \alpha, \beta) \rightarrow *$ since $(\alpha p, \beta) \rightarrow *$

(ii) $((\alpha p + \beta q) - \alpha, \beta) \rightarrow q - \alpha$ since $(\alpha p + \beta q, \beta) \rightarrow q$

$((\alpha p + \beta q) - \alpha, \gamma) \rightarrow *$ since $(\alpha p + \beta q, \gamma) \rightarrow *$

(iii) $((\beta \alpha p + \gamma q) - \alpha, \beta) \rightarrow \alpha p) - \alpha$ since $(\beta \alpha p + \gamma q, \beta) \rightarrow \alpha p$

(iv) $((\alpha p \oplus \beta q) - \alpha, \beta) \rightarrow *$ as $((\alpha p) - \alpha, \beta) \rightarrow *$, by (i) above.

$((\alpha p \oplus \beta q) - \alpha, \beta) \rightarrow q{-}\alpha$ as $((\beta q) - \alpha, \beta) \rightarrow q{-}\alpha$

The definitions given for hiding are axioms of our language. We have the following set:

$$[-+_1]\ (\sum \alpha_i p_i) - \alpha = \Big(\sum_{\alpha_i \neq \alpha} \alpha_i (p_i - \alpha) + \bigotimes_{\alpha_i = \alpha} (p_i - \alpha) \Big) \oplus \sum_{\alpha_i = \alpha} (p_i - \alpha)$$

where $\alpha \in L_{INT}$

$$[-+_2]\ (\sum \alpha_i p_i) - \alpha = \sum_{\alpha_i \neq \alpha} \alpha_i (p_i - \alpha) \quad \text{where} \quad \alpha \in L_{EXT}$$

$[--]\ p - \alpha - \beta = p - \beta - \alpha$

$[-\oplus]\quad (p \oplus q) - \alpha = (p - \alpha) \oplus (q - \alpha)$

$[-\bullet]\quad (p \bullet q) - \alpha = (p - \alpha) \bullet (q - \alpha)$ if $\alpha \notin L_{p \bullet q\, INT}$

The first two axioms define internal and external hiding on CNF programs; $[--]$ states that it is immaterial the order in which we hide while $[-\oplus]$ says that hiding a program in DNF is the same as if we form the program from the hidden subprograms using $\oplus$. The $[-\bullet]$ axiom states that concurrently composing then hiding is the same as hiding the components and then concurrently composing, provided that what is hidden is an external label of $p \bullet q$.

These axioms together with the preceding ones are later shown to satisfy our equivalence relation.

It should be noted that some of the complexity in defining our acceptance semantics over hiding is in the need to treat CNF and DNF programs differently, particularly when * is a result. The problem stems from the semantics not differentiating between how $p \oplus q$ and $p + q$ react to a stimulus which either p or q react acceptably to, i.e., do not give *. Also, the semantics have to distinguish between how such DNF and CNF programs produce *. It is possible that a semantics using modal operators to express "sometimes" and "always" may produce a simpler, cleaner semantics.

6.4 Hiding in our Dining Philosopher's Example

In our "solution" to the dining philosopher's problem we introduced a reservation guard to each philosopher and his neighbouring forks to prevent deadlock. If we abstract out these reservation guards from the composite system SYS, what program are we left with?

For three philosophers and three forks we use axiom [+●] to get that

$$\begin{aligned} SYS = {} & r_1(g\ell_1\, gr_1\, S_1 + gr_1\, g\ell_1\, S_1) \\ & + r_2(g\ell_2\, gr_2\, S_2 + gr_2\, g\ell_2\, S_2) \\ & + r_3(g\ell_3\, gr_3\, S_3 + gr_3\, g\ell_3\, S_3) \end{aligned}$$

where $S_i = p\ell_i\, pr_i\, SYS + pr_i\, p\ell_i\, SYS$ for $i = 1, 2, 3$.

Using axiom $[-+_1]$ a number of times to hide internal guard r_1 gives us:

$$\begin{aligned} SYS - r_1 = \Big(& (g\ell_1\, gr_1 (S_1 - r_1) + gr_1\, g\ell_1 (S_1 - r_1) \\ & + r_2 (g\ell_2\, gr_2\, (S_2 - r_1) + gr_2\, g\ell_2\, (S_2 - r_1)) \\ & + r_3 (g\ell_3\, gr_3\, (S_3 - r_1) + gr_3\, g\ell_3\, (S_3 - r_1))\Big) \\ \oplus \Big(& r_2 (g\ell_2\, gr_2\, (S_2 - r_1) + gr_2\, g\ell_2\, (S_2 - r_1)) \\ & + r_3 (g\ell_3\, gr_3\, (S_3 - r_1) + gr_3\, g\ell_3\, (S_3 - r_1))\Big) \end{aligned}$$

where

$$(S_i - r_1) = p\ell_i\, pr_i\, (SYS - r_1) + pr_i\, 1\ell_i (SYS - r_1)$$

Using axiom $[-+_1]$ to hide the r_2 and then the r_3 guards gives us:

$$\begin{aligned} SYS - r_1 - r_2 - r_3 = & (R_1 + R_2 + R_3) \\ & \oplus (R_1 + R_2) \oplus (R_1 + R_3) \oplus (R_2 + R_3) \\ & \oplus (R_1 \oplus R_2 \oplus R_3 \end{aligned}$$

where $\quad R_i = g\ell_i\, gr_i\, R_i' + gr_i\, g\ell_i\, R_i'$

and $\quad R_i' = p\ell_i\, pr_i\, (SYS - r_1 - r_2 - r_3) + pr_i p\ell_i (SYS - r_1 - r_2 - r_3).$

Using axiom [⊕ ⊕ +] three times we get:

$$SYS - r_1 - r_2 - r_3 = (R_1 + R_2 + R_3) \oplus R_1 \oplus R_2 \oplus R_3 .$$

We now need to use another axiom here; the extension of axiom [⊕ ⊕ +] to three components rather than two.

Axiom [⊕ ⊕ +] states that

$$x \oplus y \oplus (x + y) = x \oplus y .$$

Let us replace this axiom by the following axiom, or rather, family of axioms.

$$\left[\bigoplus \sum \right] \quad \bigoplus x_i \oplus \sum x_i = \bigoplus x_i , \quad \text{for all programs } x_i \text{ and for all i.}$$

We now use the instance of this class of axioms for i = 3. This gives

$$SYS - r_1 - r_2 - r_3 = R_1 \oplus R_2 \oplus R_3 .$$

This constructed system can be seen to be free from deadlock by inspection. We see that the hiding operation introduces ⊕ to a program SYS which previously did not contain such nondeterminism constructs. Whether our hidden system gives program R_1 or R_2 or R_3 now depends on some internalised choice mechanism, the r_i guards, which we have abstracted away from.

The informal justification for axiom $[\oslash\!\!\!\!\!\!\sum \sum]$ just extends that for $[\oplus \oplus +]$. In our appendix we prove that axiom $[\oplus \oplus +]$ is sound with respect to our semantic interpretation of the language. We believe that a generalised form of this proof would also show the soundness of $[\oslash\!\!\!\!\!\!\sum \sum]$.

7. CONCLUDING REMARKS

The formal system presented here allows us to represent communication and concurrency features as found in systems of interacting computing agents. This corresponds to how the functional concepts in serial programs are represented using the calculus. Properties such as termination and deadlock can be expressed in the formalism. Proofs of equivalence type properties can be performed while the framework can also be used to reason about deadlock features, for instance. Other proof tools need to be developed; one approach is to enable ourselves to perform induction on the number of components in a system, the dining philosophers problem for example. Work along this line has been fairly successful and will be reported in a future paper.

One problem met in concurrent programming is how to deal with the interleaving of synchronisation and computation features. One approach, as adopted by Campbell and Habermann, uses Path Expressions [1] to remove the synchronisation and control constructs from the rest of the program. The computation and control features are then separated. To a lesser extent monitors [3] also perform this function.

The approach we adopt is to have synchronisation as the primitive language feature with the more usual computation being added to this core. The simplest example of this approach, as presented in this paper, excludes values and so computation. It is not difficult though to add in value-passing and computation features. These are present in a related formal system; the CCS of Milner [8]. Both CCS and our formal system are developed from a process model of concurrency [7] where value-passing is present. Related work includes that of Hoare [4] together with the work of Hoare and others on models of CSP [5].

We have adapted the concept of experimentation as used by Milner and Hennessay [2] in the construction of our notion of acceptance semantics. This opperational approach is due originally to Landin [6], who used an abstract machine to represent the semantics of a language. A less concrete operational semantics involving relations rather than abstract machines was introduced by Plotkin in [9]. This is closer to our notion of acceptance semantics.

One possible deficiency of this work is in the use of interleaving to deal with concurrent action. Future work will attempt to produce a model which does not rely on such arbitrary interleaving. Further proof methodologies remain to be developed and this should allow us to test our formal system on some large and realistic examples. This empirical approach should, it is hoped, justify our design of model and if not, to illustrate how it could be altered to better represent reality.

8. APPENDIX I: The Equivalence Relation is a Congruence

We define $\sim$ to be the intersection $\bigcap_n \sim_n$ over n where

$p \sim_0 q$ if $p, q \in W_\Sigma$ and of sort L

$p \sim_{n+1} q$ iff

$\forall \alpha \in L.$ (a) $(p, \alpha) \rightarrow * \Rightarrow (q, \alpha) \rightarrow *$

(b) $(q, \alpha) \rightarrow * \Rightarrow (p, \alpha) \rightarrow *$

(c) $(p, \alpha) \rightarrow p' \Rightarrow \exists\, q' \cdot (q, \alpha) \rightarrow q',\ p' \sim_n q'$

(d) $(q, \alpha) \rightarrow q' \Rightarrow \exists\, p' \cdot (p, \alpha) \rightarrow p',\ p' \sim_n q'$

Note that p' and q' are program variables and so cannot be *. Thus we wish to show that for any words m and n in W_Σ then

$$m \sim n \Rightarrow \forall \text{ contexts } C\,[\ ] \cdot C[m] \sim C[n].$$

Contexts are formed using the guarding, choice, nondeterminism, hiding and concurrency operations. To show congruence it is therefore sufficient to show that for m, n and q of sort L
m ~ n implies:

(1) $\forall q \cdot m + q \sim n + q$

(2) $\forall q \cdot m \oplus q \sim n \oplus q$

(3) $\forall \gamma \in L \cdot \gamma m \sim \gamma n$

(4) $\forall \gamma \in L \cdot m - \gamma \sim n - \gamma$

(5) $\forall p \cdot m \bullet p \sim n \bullet p$

Proof. Assume $m \sim n$. Then for some $\alpha \in L$, the sort of m and n we have

1) Show for $m + q$. Assume m, α and q in CNF. If not we use $[+ \oplus]$ to get for some CNF component of m that $(m+q, \alpha) \rightarrow *$ if $(m_1+q, \alpha) \rightarrow *$. Then show that $(m_1+q, \alpha) \rightarrow *$ implies $(n_1+q, \alpha) \rightarrow *$ for some CNF n_1, a component of n. This is just the CNF case below.

1.a) Case $(m + q, \alpha) \rightarrow *$ then by 15 we have that $(m, \alpha) \rightarrow *$ and $(q, \alpha) \rightarrow *$. As $m \sim n$ then $(n, \alpha) \rightarrow *$ hence $(n + q, \alpha) \rightarrow *$ by 15, as required.

1.b) Case $(n + q, \alpha) \rightarrow *$, as (1.a).

1.c) Case $(m + q, \alpha) \rightarrow p$ where $p \neq *$.

Then by 3 and 4 either $(m, \alpha) \rightarrow p$ or $(q, \alpha) \rightarrow p$.

1.c.1) $(m, \alpha) \rightarrow p$ and as $m \sim n$ then $\exists p'$ such that $(n, \alpha) \rightarrow p'$ and $p \sim p'$ and so $(n + q, \alpha) \rightarrow p'$ by 3, as required.

1.c.2) $(q, \alpha) \rightarrow p$ and by 4 $(n + q, \alpha) \rightarrow p$. As $p \sim p$ then we have what is required.

1.d) Case $(n+q, \alpha) \rightarrow p$ where $p \neq *$, as 1.c)

We now have that if $m \sim n$ then $\forall q \cdot (m+q) \sim (n+q)$.

2) Show for $m \oplus q$. Assume m, n and q in CNF.

2.a) Case $(m \oplus q, \alpha) \rightarrow *$ then by 6.2 and 7.2 either $(m, \alpha) \rightarrow *$ or $(q, \alpha) \rightarrow *$, respectively.

2.a.1) $(m, \alpha) \rightarrow *$ and as $m \sim n$ then $(n, \alpha) \rightarrow *$. By 6.2 $(n + q) \rightarrow *$, as required.

2.a.2) $(q, \alpha) \rightarrow *$ implies $(n + q) \rightarrow *$ bu 7.2.

2.b) Case $(n \oplus q, \alpha) \rightarrow *$, as 2.a)

2.c) Case $(m \oplus q, \alpha) \rightarrow p$ where $p \neq *$. By 6.1 and 7.1 either $(m, \alpha) \rightarrow p$ or $(q, \alpha) \rightarrow p$, respectively.

2.c.1) $(m, \alpha) \rightarrow p$. As $m \sim n$ then $(n, \alpha) \rightarrow p'$ and $p \sim p'$. By $(n \oplus q, \alpha) \rightarrow p'$, as required.

2.c.2) $(q, \alpha) \rightarrow p$ and by 7.1 $(n \oplus q) \rightarrow p'$, as required.

2.d) Case $(n \oplus q, \alpha) \rightarrow p$, as 2.c).

So have that $m \sim n$ implies $(m + q) \sim (n + q)$ for all q.

3) Show for γm.

3.a) Case $(\gamma m, \alpha) \rightarrow *$ implies $\alpha \neq \gamma$ by 2. We thus have $(\gamma n; \alpha) \rightarrow *$.

3.b) Case $(\gamma n, \alpha) \rightarrow *$, as above.

3.c) Case $(\gamma m, \alpha) \rightarrow m$, from 1 with $\alpha = \gamma$. Then $(\gamma n, \alpha) \rightarrow n$ and as $m \sim n$ we have required result.

3.d) Case $(\gamma n, \alpha) \rightarrow n$, as 3.c). Thus $m \sim n$ implies $\gamma m \sim \gamma n$.

4) Show for m-γ. Assume m and n are in CNF, otherwise axiom $[-\oplus]$ can be used to get CNF terms.

4.a) $(m\text{-}\gamma, \alpha) \rightarrow *$ then either

4.a.1) $(m, \alpha) \rightarrow *$ and $(m, \gamma) = \{*\}$ and γ is internal then as $m \sim n$, so $(n, \alpha) \rightarrow *$ and $(n, \gamma) = \{*\}$ and again γ is internal (as m and n have the same sorts). Thus $(n, \alpha) \rightarrow *$.

4.a.2) $\exists n$ s.t. $\cdot[(m, \gamma) \rightarrow m_1] \wedge [(m, \gamma) \rightarrow m_2] \wedge \cdots$

$$\cdots \wedge [(m_n, \gamma) \rightarrow *] \wedge [(m_n, \alpha) \rightarrow *]$$

and α is internal to m (and so also n). As $m \sim n$ then $(n, \gamma) \rightarrow n_1$ and $m_1 \sim n_1$, in which case $(n_1, \gamma) \rightarrow n_2 \cdots$ and $(n_n, \gamma) \rightarrow *$ and $(n_n, \alpha) \rightarrow *$. Hence $(n\text{-}\gamma, \alpha) \rightarrow *$, as required.

4.a.3) $(m, \alpha) \rightarrow *$ and α is external to m (and so also to n). As $m \sim n$ then $(n, \alpha) \rightarrow *$ and so $(n\text{-}\gamma, \alpha) \rightarrow *$, as required.

4.b) as for 4.a) by symmetry

4.c) $(m\text{-}\gamma, \alpha) \rightarrow p\text{-}\gamma$ then either

4.c.1) $(m, \alpha) \rightarrow p$. As $m \sim n$ then $\exists i$ such that $m \sim_{i+1} n$ and $(n, \alpha) \rightarrow q$ with $p \sim_i q$ and $(n\text{-}\gamma, \alpha) \rightarrow q\text{-}\alpha$. Now $p \sim_0 q$ (always true) and let us assume that $p\text{-}\gamma \sim_i q\text{-}\gamma$. This is sufficient as $(n\text{-}\gamma, \alpha) \rightarrow q\text{-}\alpha$ and $p\text{-}\gamma \sim_i q\text{-}\gamma$ to give $m\text{-}\gamma \sim_i n\text{-}\gamma$ and so $m\text{-}\gamma \sim n\text{-}\gamma$ if other clauses hold.

This follows by induction on the index of our equivalence relation and is used in some of the following clauses.

4.c.2) $\exists n$ s.t. $[(m, \gamma) \rightarrow m_1] \wedge [(m_1, \gamma \rightarrow m_2] \wedge \cdots$
$\cdots \wedge [(m_{n-1}, \gamma) \rightarrow m_n] \wedge [(m_n, \alpha) \rightarrow p]$ and α is internal to m (and so to n). As $m \sim n$ then $\exists\, n_1$ such that $(n, \gamma) \rightarrow n_1$ and $m_1 \sim n_1$. Hence $(n_1, \gamma) \rightarrow n_2 \cdots$ and $(n_{n-1}, \alpha) \rightarrow n_n$ and $(n_n, \gamma) \rightarrow q$ with $p \sim q$. Using an argument as in 4.c.1) we get $(n\text{-}\gamma, \alpha) \rightarrow q\text{-}\alpha$ with $m\text{-}\gamma \sim n\text{-}\gamma$ if other clauses hold.

4.d) as for 4.c) by symmetry. Thus $\forall \gamma \cdot m \sim n$ implies $m\text{-}\gamma \sim n\text{-}\gamma$ where γ lies in the sort of m and so of n).

5) Show for $m \bullet p$. Let m and n be in CNF. Axiom $[\bullet \oplus]$ can be used on DNF terms to get CNF components on which to reason as follows:

5.a) $(m \bullet p, \alpha) \rightarrow *$ then by 12 and 13 either

5.a.1) $(m, \alpha) \rightarrow *$ and so $(n, \alpha) \rightarrow *$ by $m \sim n$ and by 12, $(n \bullet p, \alpha) \rightarrow *$.

or 5.a.2) $(p, \alpha) \rightarrow *$ and so by 13 $(n \bullet p, \alpha) \rightarrow *$.

5.b) follows 5.a) by symmetry.

5.c.1) $(m \bullet p, \alpha) \rightarrow m' \bullet p$ and $(m, \alpha) \rightarrow m'$ and α is not in the sort of p. Now as $m \sim n$ then $(n, \alpha) \rightarrow n'$ and $m' \sim n'$. Here we assume $m' \bullet p \sim_0 n' \bullet p$ and $m' \bullet p \sim_i n' \bullet p$. As $(n, \alpha) \rightarrow n'$ and α not in sort of p then $(n \bullet p, \alpha) \rightarrow n' \bullet p$. Thus $m \bullet p \sim_{i+1} n \bullet p$ and so also $m \bullet p \sim n \bullet p$ if other clauses for $\bullet$ hold.

5.c.2) $(m \bullet p, \alpha) \rightarrow m \bullet p'$ and $(p, \alpha) \rightarrow p'$ and α is not in sort of m. As $m \sim n$ then $(n \bullet p, \alpha) \rightarrow n \bullet p'$. Assume $m \bullet p' \sim_0 n \bullet p'$ and $m \bullet p' \sim_i n \bullet p'$ and so $m \bullet p \sim_{i+1} n \bullet p$ and also $m \bullet p \sim n \bullet p$ if other clauses for $\bullet$ hold.

5.c.3) $(m \bullet p, \alpha) \rightarrow m' \bullet p'$ and $(m, \alpha) \rightarrow m'$ and $(p, \alpha) \rightarrow p'$. As $m \sim n$ then $(n, \alpha) \rightarrow n'$ and $m' \sim n'$. Assume $m' \bullet p' \sim_0 n' \bullet p'$ and $m' \bullet p' \sim_i n' \bullet p'$ and so we have $m' \bullet p \sim_{i+1} n \bullet p$ and so also $m \bullet p \sim n \bullet p$ if other clauses for $\bullet$ hold.

5.d) follows 5.c) by symmetry.

We therefore have, for all programs p, that $m \sim n$ implies $m \bullet p \sim n \bullet p$. By 1) to 5) we have that our equivalence relation $\sim$ is a congruence and we can now replace parts (or subprograms) of programs in our language by equivalent parts without changing the meaning of the program as a whole.

What programs in our language are equivalent, or to rephrase this question, how do we construct equivalent programs?

Our axioms define an equivalence relation and we would like that this equivalence is the same as $\sim$. We now show that our axioms are sound with respect to $\sim$; that is, if $a = b$ is an equality arrived at by using the axioms then $a \sim b$. We are then able to replace program a by program b without changing meaning (as $a \sim b$). Our axioms are thus consistent with our notion of equivalence (in terms of our acceptance relation). We would also like that our set of axioms be complete; that is, if $a \sim b$ then we have $a = b$ using the axioms alone. This is believed to be the case but a proof remains to be performed. The proof of consistency of the axioms now follows.

9. APPENDIX II: Consistency of the Axioms

We shall show that if $p = q$ using our axioms, then $p \sim q$.

It is sufficient to show that every axiom gives us an equivalence.

Proof. We shall assume that expression variables x and y in the following are in CNF. If they were in DNF conditions 6, 7, 19 and 20 are used to access their CNF components, and we then need reason o about these.

Axioms $[+_1]$, $[+_2]$, $[++]$ and $[+\Delta]$ are obvious. As an example let us deal with $[+\Delta]$.

$[+\Delta]$ a) $(x + \Delta, \alpha) \rightarrow *$ implies $(x, \alpha) \rightarrow *$ and $(\Delta, \alpha) \rightarrow *$ by 5. The latter is always true.

b) $(x, \alpha) \rightarrow *$ implies $(x + \Delta, \alpha) \rightarrow *$ by 5 since $(\Delta, \alpha) \rightarrow *$ always holds, and x is in CNF.

c) $(x + \Delta, \alpha) \rightarrow x'$ implies $(x, \alpha) \rightarrow x'$ by 3, since we never have $(\Delta, \alpha) \rightarrow x'$.

d) $(x, \alpha) \rightarrow x'$ implies $(x + \Delta, \alpha) \rightarrow x'$ by 3.

All four clauses follow from relation conditions 5 and 3 and $x + \Delta \sim x$ as required.

$[\oplus_1]$ a) $(x \oplus x, \alpha) \rightarrow *$ implies that $(x, \alpha) \rightarrow *$

b) $(x, \alpha) \rightarrow *$ implies that $(x \oplus x, \alpha) \rightarrow *$

c) $(x \oplus x, \alpha) \rightarrow x'$ implies that $(x, \alpha) \rightarrow x'$

d) $(x, \alpha) \rightarrow x'$ implies $(x \oplus x, \alpha) \rightarrow x'$

using conditions 6 and 7. Axioms $[\oplus_2]$ and $[\oplus\, \oplus]$ again follow immediately from conditions 6 and 7 and are omitted.

$[+\, \oplus]$ a) $(x + (y \oplus z), \alpha) \rightarrow *$ implies that $(x, \alpha) \rightarrow *$ and $(y \oplus z, \alpha) \rightarrow *$ and this latter implies that either $(y, \alpha) \rightarrow *$ or $(z, \alpha) \rightarrow *$. Hence $(x, \alpha) \rightarrow * \wedge (y, \alpha) \rightarrow *$ or $(x, \alpha) \rightarrow * \wedge (z, \alpha) \rightarrow *$, in which case $((x+y) \oplus (x+z), \alpha) \rightarrow *$.

b) $((x+y) \oplus (x+z), \alpha) \rightarrow *$ implies that $(x + (y \oplus z), \alpha)$ as for a) above.

c) $(x + (y \oplus z), \alpha) \rightarrow p$ implies that $(x, \alpha) \rightarrow p$ or $(y \oplus z, \alpha) \rightarrow p$, the latter again giving that either $(y, \alpha) \rightarrow p$ or $(z, \alpha) \rightarrow p$.

c.1) if $(x, \alpha) \rightarrow p$ then $(x + y, \alpha) \rightarrow p$ and so $((x + y) \oplus (x+z), \alpha) \rightarrow p$.

c.2) if $(y, \alpha) \rightarrow p$ then $(x + y, \alpha) \rightarrow p$ and so $((x+y) \oplus (x+z), \alpha) \rightarrow p$.

c.3) if $(z, \alpha) \rightarrow p$ then $(x+z, \alpha) \rightarrow p$ and so $((x+y) \oplus (x+z), \alpha) \rightarrow p$.

d) $((x+y) \oplus (x+z), \alpha) \rightarrow p$ and so either $(x+y, \alpha) \rightarrow p$ or $(x+z, \alpha) \rightarrow p$. So either $(x, \alpha) \rightarrow p$ or $(y, \alpha) \rightarrow p$ or $(z, \alpha) \rightarrow p$.

d.1) $(x, \alpha) \rightarrow p$ and so $(x + (y \oplus z), \alpha) \rightarrow p$

d.2) $(y, \alpha) \rightarrow p$ and so $(y \oplus z, \alpha) \rightarrow p$ then $(x + (y \oplus z), \alpha) \rightarrow p$.

d.3) as d.2).

$x + (y \oplus z) \sim (x+y) \oplus (x+z)$ follows. In the above, conditions 3, 4, 5, 6 and 7 are used.

[$\oplus\oplus+$] a) $(x \oplus y \oplus (x+y), \alpha) \rightarrow *$ implies either

a. 1) $(x, \alpha) \rightarrow *$ and so $(x \oplus y, \alpha) \rightarrow *$, by 6. 2

a. 2) $(y, \alpha) \rightarrow *$ and so $(x \oplus y, \alpha) \rightarrow *$, by 7. 2 and 6. 2

a. 3) $(x+y, \alpha) \rightarrow *$ and so $(x, \alpha) \rightarrow *$ and $(y, \alpha) \rightarrow *$ hence $(x \oplus y, \alpha) \rightarrow *$, by 7. 2 and 5.

b) $(x \oplus y, \alpha) \rightarrow *$ implies either, by 6. 2,

b. 1) $(x, \alpha) \rightarrow *$ and so $(x \oplus y \oplus (x+y), \alpha) \rightarrow *$

b. 2) as for 6. 1)

c) $(x \oplus y \oplus (x+y), \alpha) \rightarrow p$ implies either

c. 1) $(x, \alpha) \rightarrow p$ and so $(x \oplus y, \alpha) \rightarrow p$ by 6. 1,

c. 2) $(y, \alpha) \rightarrow p$ and so $(x \oplus y, \alpha) \rightarrow p$ by 6. 1 and 7. 1

c. 3) $(x+y), \alpha) \rightarrow p$ and so either $(x, \alpha) \rightarrow p$ in which case $(x \oplus y, \alpha) \rightarrow p$ or $(y, \alpha) \rightarrow p$ and again $(x \oplus y, \alpha) \rightarrow p$. By 3, 4, 6. 1 and 7. 1.

d) $(x \oplus y, \alpha) \rightarrow p$ implies either

d. 1) $(x, \alpha) \rightarrow p$ and so $(x \oplus y \oplus (x+y), \alpha) \rightarrow p$, by 6. 1 or

d. 2) $(y, \alpha) \rightarrow p$, as for d. 1).

$x \oplus y \oplus (x+y) \sim x \oplus y$ then follows.

[$\bullet$] a) $(x \bullet y, \alpha) \rightarrow *$ implies that

a. 1) by 12, $(x, \alpha) \rightarrow *$ and by 13, $(y \bullet x) \rightarrow *$

a. 2) by 13, $(y, \alpha) \rightarrow *$ and by 12, $(y \bullet x) \rightarrow *$.

b) as for a)

c) $(x \bullet y, \alpha) \rightarrow p$ implies that

c. 1) $\alpha \notin L_y$ and $(x, \alpha) \rightarrow x'$ and $p = x' \bullet y$. Assume $x' \bullet y \sim_0 y \bullet x'$ and $x' \bullet y \sim_o y \bullet x'$ then condition 9 gives $(y \bullet x, \alpha) \rightarrow y \bullet x'$, with $y \bullet x' \sim x' \bullet y$ by induction on $\sim$.

c. 2) $\alpha \notin L_x$ and $(y, \alpha) \rightarrow y'$ and $p = x \bullet y'$, similarly.

c. 3) $\alpha \notin L_x \cap L_y$ and $(x, \alpha) \rightarrow x'$ and $(y, \alpha) \rightarrow y'$ and $p = x' \bullet y'$. As for c. 1) using 11.

d) as for c)

Then $x \bullet y \sim y \bullet x$.

[$\bullet\ \bullet$] similar to [$\bullet$] using conditions 9, 10, 11, 12 and 13.

[$\bullet$ +] for $x = \sum \mu_i x_i$ and $y = \sum \rho_j y_j$ then

a) $(x \bullet y, \alpha) \rightarrow *$ implies that either

a. 1) by 12 for CNF x, $(x, \alpha) = \{*\}$ in which case either

a. 1. 1) $\left(\sum_{\mu_i \notin L_y} \mu_i (x_i \bullet q), \alpha \right) = \{*\}$ by 5, for $\alpha \in L_x$ and $\alpha \notin L_y$ or

a. 1.2) $\left(\sum_{\mu_i = x_j} \mu_i (x_i \bullet y_j), \alpha \right) = \{*\}$ for $\alpha \in L_x \cap L_y$ by 5.

a. 2) by 13 for CNF y, $(y, \alpha) = \{*\}$. As for a. 1).

By condition 5 this makes the right hand side of axiom [$\bullet$ +] give $\{*\}$ on receipt of an α stimulus, via our relation. The three clauses making up the right hand side are mutually exclusive in their reaction to stimuli since we have $\alpha \notin L_y, \alpha \notin L_x$ and $\alpha \in L_x \cap L_y$.

b) Let us call the right hand side of axiom [$\bullet$ +] rhs. Then

(rhs, α) $\rightarrow *$ if all three clauses only give $*$ on receipt of an α. As explained above this arises when either

b.1) $\left(\sum_{\mu_i \notin L_y} \mu_i(x_i \bullet y), \alpha \right) = \{*\}$ and $\alpha \notin L_y$

b.2) $\left(\sum_{\rho_j \notin L_x} \rho_j(x \bullet y_j), \alpha \right) = \{*\}$ and $\alpha \notin L_x$

b.3) $\left(\sum_{\substack{\mu_i = \rho_j \\ \mu_i \in L_x \cap L_y}} \mu_i(x_i \bullet y_j), \alpha \right) = \{*\}$ and $\alpha \notin L_x \cap L_y$

By conditions 12 and 13 and above $(x \bullet y, \alpha) = \{*\}$ as required.

c) $(x \bullet y, \alpha) \rightarrow p$ implies that

c.1) $\alpha \notin L_y$ and $(x, \alpha) \rightarrow x'$ and $p = x' \bullet y$, by 9. Then

$\left(\sum_{\mu_i \notin L_y} \mu_i(x_i \bullet y), \alpha \right) \rightarrow x \bullet y$ where $\mu_i = \alpha$ and $x_i = x'$

and so (rhs, α) $\rightarrow x' \bullet y$.

or c.2) $\alpha \notin L_x$ and $(y, \alpha) \rightarrow y'$ and $p = x \bullet y'$, by 10. As for c.1).

or c.3) $\alpha \in L_x \cap L_y$ and $(x, \alpha) \rightarrow x'$ and $(y, \alpha) \rightarrow y'$, by 11. As for c.1).

d) $(\text{rhs}, \alpha) \rightarrow p$ implies, using conditions 3 and 4, that

d.1) $(\sum_{\mu_i \notin L_y} \mu_i(x_i \bullet y), \alpha) \rightarrow x' \bullet y$ where $\mu_i = \alpha$ and $x_i = x'$.

As $\alpha x'$ is a summand of x then by 9 $(x \bullet y, \alpha) \rightarrow x' \bullet y$.

d.2) $(\sum_{\rho_j \notin L_x} \rho_j(x \bullet y_j), \alpha) \rightarrow x \bullet y'$. Similar to d.1)

d.3) $(\sum_{\mu_i = \rho_j} \mu_i(x_i \bullet y_j), \alpha) \rightarrow x' \bullet y'$. Similar to d.1).

By a), b), c) and d) we have that

$$x \bullet y \sim \text{rhs} \text{ , as required.}$$

[$\bullet \oplus$] a) $(x \bullet (y \oplus z), \alpha) \rightarrow *$ implies that either $(x, \alpha) \rightarrow *$ by 12, or $(y \oplus z, \alpha) \rightarrow *$ by 13.

a.1) $(x, \alpha) \rightarrow *$ in which case $(x \bullet y, \alpha) \rightarrow *$ and $(x \bullet z, \alpha) \rightarrow *$ by 12, hence $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow *$ by 6.2 and 7.2.

a.2) $(y \oplus z, \alpha) \rightarrow *$ and so either $(y, \alpha) \rightarrow *$ or $(z, \alpha) \rightarrow *$ by 6.2 and 7.2 respectively.

a.2.1) $(y, \alpha) \rightarrow *$ and by 12 $(x \bullet y, \alpha) \rightarrow *$ so by 6.2 $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow *$.

a.2.2 follows similarly.

b) $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow *$ implies either $(x \bullet y, \alpha) \rightarrow *$ or $(x \bullet z, \alpha) \rightarrow *$.

b.1) $(x \bullet y, \alpha) \rightarrow *$ and by conditions 12 and 13 either $(x, \alpha) \rightarrow *$ or $(y, \alpha) \rightarrow *$.

b.1.1) $(x, \alpha) \rightarrow *$ and so $(x \bullet (y \oplus z), \alpha) \rightarrow *$ by 12.

b.1.2) $(y, \alpha) \rightarrow *$, so by 6 $(y \oplus z, \alpha) \rightarrow *$. By 13 $(x \bullet (y \oplus z), \alpha) \rightarrow *$.

b.2) $(x \bullet z, \alpha) \rightarrow *$ follows as for 6.1).

c) $(x \bullet (y \oplus z), \alpha) \rightarrow p$ implies either

c.1) $(x, \alpha) \rightarrow x'$ and $\alpha \notin L_y$ by 9 with $p = x' \bullet (y \oplus z)$. Thus $(x \bullet y, \alpha) \rightarrow x' \bullet y$, so $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow x' \bullet y$ by 6.1. and as $L_y = L_z$ then
$(x \bullet z, \alpha) \rightarrow x' \bullet z$, so $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow x' \bullet z$ by 7.1.

Then $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow x' \bullet y \oplus x' \bullet z$ by 14, and $x' \bullet (y \oplus z) \sim x' \bullet y \oplus x' \bullet z$ by an inductive argument on indexed equivalences as used in proofs above ($[\bullet]$ for instance). Or

c.2) $(y \oplus z, \alpha) \rightarrow y'$ and $\alpha \in L_x$ by 10 with $p = x \bullet y'$.
Conditions 6.1 and 7.1 imply that $(y, \alpha) \rightarrow y'$ or $(z, \alpha) \rightarrow z'$

c.2.1) $(y, \alpha) \rightarrow y'$ and $(x \bullet y, \alpha) \rightarrow x \bullet y'$ by 10 and so $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow x \bullet y'$ by 6.1.

c.2.2) $(z, \alpha) \rightarrow z'$ follows similarly. Or

c.3) $(x, \alpha) \rightarrow x'$ and $(y \oplus z, \alpha) \rightarrow y'$ by 11, with $p = x' \bullet y'$. Either $(y, \alpha) \rightarrow y'$ or $(z, \alpha) \rightarrow y'$ by 6.1 and 7.1.

c.3.1) $(y, \alpha) \rightarrow y'$ and $(x \bullet y, \alpha) \rightarrow x' \bullet y'$ by 11,

$(x \bullet y \oplus x \bullet z, \alpha) \rightarrow x' \bullet y'$ by 6.1.

c.3.2) $(z, \alpha) \rightarrow y'$, follows similarly.

d) $(x \bullet y \oplus x \bullet z, \alpha) \rightarrow p$ implies either $(x \bullet y, \alpha) \rightarrow p$ by 6.1 or $(x \bullet z, \alpha) \rightarrow p$ by 7.1.

d.1) $(x \bullet y, \alpha) \rightarrow p$ in which case either

d.1.1) $(x, \alpha) \rightarrow x'$ and $\alpha \notin L_y$, with $p = x' \bullet y$.
As $L_z = L_y$, $(x \bullet (y \oplus z), \alpha) \rightarrow x' \bullet (y \oplus z)$.
Now $x' \bullet (y \oplus z) \sim (x' \bullet y \oplus x' \bullet z)$ by an inductive argument on indexed equivalences. By 14,
$(x \bullet (y \oplus z), \alpha) \rightarrow x' \bullet y$.

d.1.2) $(y, \alpha) \rightarrow y'$ and $\alpha \notin L_x$, with $p = x \bullet y'$. By 6.1,
$(y \oplus z, \alpha) \rightarrow y'$ and as $\alpha \notin L_x$ $(x \bullet (y \oplus z), \alpha) \rightarrow x \bullet y'$ by 10.

d.1.3) $(x, \alpha) \rightarrow x'$ and $(y, \alpha) \rightarrow y'$, with $p = x' \bullet y'$.
By 6.1, $(y \oplus z, \alpha) \rightarrow y'$ hence
$(x \bullet (y \oplus z), \alpha) \rightarrow x' \bullet y'$, by 11.

d.2) $(x \quad z, \alpha) \rightarrow p$, similar to d.1). By above
$x \bullet (y \oplus z) \sim x \bullet y \oplus x \bullet z$

[$\alpha \oplus +$] Show that $\alpha x + \alpha y = \alpha x \oplus \alpha y$.

a) $(\alpha x + \alpha y, \gamma) \rightarrow *$ implies by 2 and 5 that $\alpha \neq \gamma$. By 2, $(\alpha x, \gamma) \rightarrow *$ and $(\alpha y, \gamma) \rightarrow *$ and $(\alpha x \oplus \alpha y, \gamma) \rightarrow *$ by 6.2 and 7.2.

b) $(\alpha x \oplus \alpha y, \gamma) \rightarrow *$ implies by 6.2 and 7.2 that either b.1) $(\alpha x, \gamma) \rightarrow *$ and by 2, $\alpha \neq \gamma$. Then $(\alpha y, \gamma) \rightarrow *$ and $(\alpha x + \alpha u, \gamma) \rightarrow *$ by 5. Or

b.2) $(\alpha y, \gamma) \rightarrow *$, as above.

c) $(\alpha x + \alpha y, \gamma) \rightarrow p$ and by 3 and 4 $(\alpha x, \gamma) \rightarrow p$ or $(\alpha x, \gamma) \rightarrow p$, respectively.

c.1) $(\alpha x, \gamma) \rightarrow p$ and by 6.1 $(\alpha x \oplus \alpha y, \gamma) \rightarrow p$.

c.2) $(\alpha x, \gamma) \rightarrow p$ follows by symmetry.

d) $(\alpha x \oplus \alpha y, \gamma) \rightarrow p$ and by 6.1 and 7.1 either $(\alpha x, \gamma) \rightarrow p$ or $(\alpha y, \gamma) \rightarrow p$, respectively.

d.1) $(\alpha x, \gamma) \rightarrow p$ and by 3, $(\alpha x + \alpha y, \gamma) \rightarrow p$.

d.2) $(\alpha y, \gamma) \rightarrow p$ similarly.

For all $\beta \in L_x - \{\alpha\}$ with $x = \sum a_i x_i$.

$[- +_1]$ Let $\left(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i - \alpha) + \bigoplus_{\alpha_i = \alpha} (p_i - \alpha)\right) \oplus \bigoplus_{\alpha_i = \alpha} (p_i - \alpha)$ be rhs.

a) $(x-\alpha, \beta) \to *$ implies either

a.1) $(x, \beta) \to *$ and $(x, \alpha) \to *$ by 15. As $x = \sum \alpha_i x_i$ then $\nexists$ no x' such that $(x, \alpha) = x'$. $\bigoplus_{\alpha_i = \alpha} (x_i - \alpha)$ is then the null $\oplus$ sum. This derived operation in its null form is in effect an identity for both + and $\oplus$. As $(x, \beta) \to *$ then $\left(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i - \alpha), \beta\right) \to *$. Finally, $(\text{rhs}, \alpha) \to *$.

a.2) $\exists n \geq 1$ such that

$(x, \alpha) \to x_1 \wedge \cdots \wedge (x_{n-1}, \alpha) \to x_n \wedge (x_n, \beta) \to *$ by 7.2. As $(x_n \alpha) \to *$ and $(x_n, \beta) \to *$ then $(x_n - \alpha, \beta) \to *$ by 15. Let us assume $x - \alpha \sim_i$ rhs for $i \leq n$. Now either $x_{n-1} = \sum \alpha_i x_{n-1_i}$, in which case

a.2.1) $(x_{n-1} - \alpha) \sim_0 \left(\sum_{\alpha_i \neq \alpha} \alpha_i(x_{n-1_i} - \alpha) + \bigoplus_{\alpha_i = \alpha} (x_{n-1_i} - \alpha)\right)$ $\oplus \sum_{\alpha_i = \alpha} (x_{n-1_i} - \alpha)$. As $(x_{n-1}, \alpha) \to x_n$ then $\exists\, \alpha_i = \alpha$ with $x_{n-1_i} = x_n$. As $(x_n - \alpha, \beta) \to *$ so $(x_{n-1_i} - \alpha, \beta) \to *$. By 6.2, as $\bigoplus$ is derived from $\oplus$, so $\left(\bigoplus_{\alpha_i = \alpha} (x_{n-1_i} - \alpha), \beta\right) \to *$ and by 6.2 again $(x_{n-1} - \alpha, \beta) \to *$. Or

a.2.2 $x_{n-1} = y \oplus z$. By axiom $[- \oplus]$ we have $(x_{n-1} - \alpha) = y - \alpha \oplus z - \alpha$ and can apply it repeatedly to get some y (or z) of form $\sum \alpha_i x_{n-1_i}$. We then follow a.2.1) to again get $(x_{n-1} - \alpha, \beta) \to *$.

This procedure can be repeated to get $(x_1 - \alpha, \beta) \rightarrow *$ with $(x, \alpha) \rightarrow x_1$. As $x = \sum \alpha_i x_i$ then for some i, $\alpha_i = \alpha$ with $x_i = x_1$. We therefore have $(\not\sum x_i - \alpha, \beta) \rightarrow *$ by 6.2 and by 6.2 again have $(\text{rhs}, \beta) \rightarrow *$.

a.3) 17.3 gives a similar chain but with x_n in DNF. We argue as for a.2) for one of its CNF components.

b) $(\text{rhs}, \beta) \rightarrow *$ implies either b.1) or b.2) by 6.2 and 7.2.

b.1) $(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i - \alpha) + \not\sum_{\alpha_i = \alpha} (x_i - \alpha), \beta) \rightarrow *$. Condition 5 gives us that either

b.1.1) $(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i - \alpha), \beta) \rightarrow *$ and $(\not\sum_{\alpha_i = \alpha} (x_i - \alpha), \beta) \rightarrow *$.
The latter gives $(\alpha x_i, \beta) \rightarrow *$ by 2 and the former gives $(\sum_{\alpha_i \neq \alpha} \alpha_i x_i, \beta) \rightarrow *$ by 3, 4, and 16. By 3 and 4 again and 17.2 we have $(x - \alpha, \beta) \rightarrow *$. Or

b.1.2) $(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i - \alpha), \beta) \rightarrow *$ and $\sum_{\alpha_i = \alpha}$ is null, that is, there is no $\alpha_i = \alpha$. In this case $(x, \alpha) \rightarrow *$ and 15 gives $(x - \alpha, \beta) \rightarrow *$.

b.1.3) $(\sum_{\alpha_i = \alpha} (x_i - \alpha), \beta) \rightarrow *$ and $\sum_{\alpha_i = \alpha}$ is null. Thus all α_i's $= \alpha$ and as $(\sum_{\alpha_i = \alpha} (x_i - \alpha), \beta) \rightarrow *$ 17.2 gives $(p - \alpha, \beta) \rightarrow *$.

b.2) $(\sum_{\alpha_i = \alpha} (x_i - \alpha), \beta \rightarrow *$. As for b.1.3).

c) $(x-\alpha, \beta) \rightarrow p$ implies that either

c.1) $(x, \beta) \rightarrow p'$ and $p = p' - \alpha$. As $p = \sum \alpha_i p_i$ $\exists$ some α_i such that $\alpha_i = \beta$ and $p_i = p'$, by 3. For this i, $(\alpha_i(p_i-\alpha), \beta) \rightarrow (p'-\alpha)$ by 1. By 3, $(\sum_{\alpha_i \neq \alpha} \alpha_i(p_i-\alpha) + \sum_{\alpha_i = \alpha} (p_i-\alpha), \beta) \rightarrow (p'-\alpha)$ and by 6.1, $(\text{rhs}, \beta) \rightarrow p'-\alpha$. Or

c.2) $\exists$ some $n \geq 1$ such that $(x, \alpha) \rightarrow x_1 \wedge \cdots \wedge (x_{n-1}, \alpha) \rightarrow x_n$ and $(x_n, \beta) \rightarrow p'$. Let us assume $p-\alpha \sim_i$ rhs for $i \leq n$. By 16, $(x_n-\alpha, \beta) \rightarrow p'-\alpha$, and as $(x_{n-1}, \alpha) \rightarrow x_n$ then $(x_{n-1}-\alpha, \beta) \rightarrow p'-\alpha$ using equivalence for $i \leq n$, as we did in a). Repeating this for i up to n we get $(x_1-\alpha, \beta) \rightarrow p'-\alpha$.

As $(x, \alpha) \rightarrow x_1$ and $x = \sum \alpha_i x_i$ then $\exists$ some i such that $\alpha_i = \alpha$ and $x_i = x_1$. As $(x_1-\alpha, \beta) \rightarrow \alpha$ then $(\sum_{\alpha_i = \alpha} (x_i-\alpha), \beta) \rightarrow p'-\alpha$ by 6.1. Again by 6.1, $(\text{rhs}, \beta) \rightarrow p'-\alpha$.

d) $(\text{rhs}, \beta) \rightarrow p$ implies that either

d.1) $(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i-\alpha) + \sum_{\alpha_i = \alpha} (x_i-\alpha), \beta) \rightarrow p$ and either

d.1.1) $(\sum_{\alpha_i \neq \alpha} \alpha_i(x_i-\alpha), \beta) \rightarrow p$ in which case $\exists$ some i such that $\alpha_i = \beta$ and $p = (x_i-\alpha)$. So $(\alpha_i x_i, \beta) \rightarrow x_i$, and $(x, \beta) \rightarrow x_i$. Condition 16 then gives $(x-\alpha, \beta) \rightarrow x'-\alpha$.

d.1.2) $(\sum_{\alpha_i=\alpha} (x_i-\alpha), \beta) \rightarrow p$ and by 6.1).

$\exists$ Some i such that $((x_i-\alpha), \beta) \rightarrow p$ and $\alpha = \alpha_i$.

As $(\alpha_i x_i, \alpha) \rightarrow x_i$ then by 3,

$(\sum \alpha_i x_i, \alpha) \rightarrow x_i$, i.e., $(p, \alpha) \rightarrow x_i$.

As $(x_i-\alpha, \beta) \rightarrow p$ then by 16 $(x_i, \beta) \rightarrow p'$ with $p = p'-\alpha$.

Using $(p, \alpha) \rightarrow x_i$ and 16.1 we have $(p-\alpha, \beta) \rightarrow p'-\alpha$.

d.2) $(\sum_{\alpha_i=\alpha} (x_i-\alpha), \beta) \rightarrow p$, as for d.1.2. We therefore have

$$p-\alpha \sim \Big(\sum_{\alpha_i\neq\alpha} \alpha_i(p_i-\alpha) + \sum_{\alpha_i=\alpha} (p_i-\alpha)\Big) \oplus \sum_{\alpha_i=\alpha} (x_i-\alpha) \text{ for internal } \alpha.$$

$[-+_2]$ Let $x = \sum \alpha_i x_i$.

a) $(x-\alpha, \beta) \rightarrow *$ implies, by 16, that

$(x, \beta) \rightarrow *$. By 5, for all i, $(\alpha_i x_i, \beta) \rightarrow *$. Thus

$(\sum_{\alpha_i\neq\alpha} \alpha_i(x_i-\alpha), \beta) \rightarrow *$.

b) $(\sum_{\alpha_i\neq\alpha} a_i(x_i-\alpha), \beta) \rightarrow *$ implies that for all i such that $\alpha_i \neq \alpha$,

$(\alpha_i(x_i-\alpha), \beta) \rightarrow *$. In this case there is no i such that

$\alpha_i = \beta$ in $\sum \alpha_i x_i$. By 2 and 5, $(x, \beta) \rightarrow *$.

c) $(x-\alpha, \beta) \rightarrow p$ implies that $(x, \beta) \rightarrow p'$ with $p = p'-\alpha$, by 16.

Then $\exists$ some i such that $\alpha_i = \beta$, and $x_i = p'$, and

$(\alpha_i(x_i-\alpha), \beta) \rightarrow p'-\alpha$ by 16. By 3, $(\sum_{\alpha_i\neq\alpha} \alpha_i(x_i-\alpha), \beta) \rightarrow p'-\alpha$.

d) $(\sum_{\alpha_i\neq\alpha} a_i(x_i-\alpha), \beta) \rightarrow p$ implies, by 3, that $\exists$ i such that

$\alpha_i \neq \alpha$ and $(\alpha_i(x_i-\alpha), \beta) \rightarrow p$. By 16, $(\alpha_i x_i, \beta) \rightarrow p'$ with

$p = p'-\alpha$, and by 3 $(p, \beta) \rightarrow p'$. This gives $(p-\alpha, \beta) \rightarrow p'-\alpha$ by 16.

$p-\alpha \sim \sum_{\alpha_i\neq\alpha} \alpha_i(p_i-\alpha)$ for external α.

[- ●] We show $(x \bullet y) - \alpha \sim (x-\alpha) \bullet (y-\alpha)$ for $\alpha \in L_x$ and $\alpha \notin L_y$. Because of this $y-\alpha = y$. The case with $\alpha \in L_y$ and $\alpha \notin L_x$ follows by symmetry. Case internal α:

a) $(x \bullet y) - \alpha, \beta) \rightarrow *$ implies either

a. 1) $((x \bullet y), \beta) \rightarrow *$ and $(x \bullet y), \alpha) \rightarrow *$, by 15. In this case either

a. 1. 1) $(x, \beta) \rightarrow *$, by 12 and as $(x,\alpha) \rightarrow *$ so $(x-\alpha,\beta) \rightarrow *$ by 15.

By 12, $((x-\alpha) \bullet (y, \beta) \rightarrow *$.

a. 1. 2) $(y, \beta) \rightarrow *$. By 13, $((x-\alpha) \bullet (y-\alpha), \beta) \rightarrow *$.

a. 2) $\exists$ some $n \geq 1$ such that

$((x \bullet y), \alpha) \rightarrow p_1 \wedge \cdots \wedge (p_{n-1}, \alpha) \rightarrow p_n$ and $(p_n, \alpha) \rightarrow *$ and $(p_n, \beta) \rightarrow *$ by 17.2. By 9 we have $(x, \beta) \rightarrow x_1$ and $(x_1, \alpha) \rightarrow x_2 \cdots$ where $p_i = x_i \bullet y$ (as $\alpha \notin L_y$).

As $(x_n, \alpha) \rightarrow *$ and $(x_n, \beta) \rightarrow *$ we have by 17.2 that $(x-\alpha, \beta) \rightarrow *$. By 12, $((x-\alpha) \bullet y, \beta) \rightarrow *$. Or

a. 3) By 17.3 we have a similar chain but with p_n in DNF. Follows a. 2) for some CNF component.

b) $((x-\alpha) \bullet y, \beta) \rightarrow *$ implies either

b. 1) $(x-\alpha, \beta) \rightarrow *$, in which case either

b. 1. 1) $(x, \alpha) \rightarrow *$ and $(x, \beta) \rightarrow *$ and by 12, $(x \bullet y, \alpha) \rightarrow *$ and $(x \bullet y, \beta) \rightarrow *$. By 15, $((x \bullet y)-\alpha, \beta) \rightarrow *$.

b. 1. 2) $\exists$ some $n \geq 1$ such that $(x, \alpha) \rightarrow x_1 \wedge \cdots \wedge (x_{n-1}, \alpha) \rightarrow x_n \wedge (x_n, \alpha) \rightarrow *$ and $(x_n, \beta) \rightarrow *$, by 17.2. As $\alpha \notin L_y$, by 9 we have

$(x \bullet y, \alpha) \rightarrow (x_1 \bullet y) \wedge \cdots (x_{n-1} \bullet y, \alpha) \rightarrow (x_n \bullet y)$ and $((x_n \bullet y), \alpha) \rightarrow *$ and $((x_n \bullet y), \beta) \rightarrow *$. By 17.2, $((x \bullet y)\text{-}\alpha), \beta) \rightarrow *$.

b.1.3) by 17.3 we have a similar chain with DNF x_n. As for b.1.2) with a CNF component.

b.2) $(y, \beta) \rightarrow *$ and by 13, $(x \bullet y, \beta) \rightarrow *$.

b.2.1) $(x, \alpha) \rightarrow *$ and as $\alpha \notin L_y$, $(x \bullet y, \alpha) \rightarrow *$ and by 15, $((x \bullet y)\text{-}\alpha), \beta) \rightarrow *$.

b.2.2) $(x, \alpha) \rightarrow x_1 \wedge \cdots \wedge (x_n, \alpha) \rightarrow * \wedge (x_n \beta) \rightarrow *$
As $\alpha \notin L_y$ then $(x \bullet y, \alpha) \rightarrow x_1 \bullet y \wedge \cdots \wedge (x_n \bullet y, \alpha) \rightarrow *$ and $(x_n \bullet y, \beta) \rightarrow *$. By 17.2, $((x \bullet y)\text{-}\alpha, \beta) \rightarrow *$.

c) $((x \bullet y)\text{-}\alpha, \beta) \rightarrow p$ implies

c.1) $(x \bullet y, \beta) \rightarrow p'$ and $p = p'\text{-}\alpha$, by 16.

c.1.1) $\beta \notin L_y$ and 9 gives
$(x, \beta) \rightarrow p''$ with $p' = p'' \bullet y$. By 16,
$(x\text{-}\alpha, \beta) \rightarrow p''\text{-}\alpha$ and as $\beta \notin L_y$ 9 gives
$(x\text{-}\alpha \bullet y\text{-}\alpha, \beta) \rightarrow (p''\text{-}\alpha) \bullet (y\text{-}\alpha)$. Now
$(p'' \bullet y)\text{-}\alpha \sim (p''\text{-}\alpha) \bullet (y\text{-}\alpha)$ by an inductive argument on our indexed definition of ~, as required.

c.1.2) $\beta \notin L_x$, as for c.1.1) by symmetry using 10.

c.1.3) $\alpha \notin L_x \cap L_y$, similar to c.1.1) using II.

c.2) $\exists$ some n such that $(x \bullet y, \alpha) \rightarrow x_1 \bullet y \wedge \cdots \wedge$ $\wedge ((x_{n-1} \bullet y), \alpha) \rightarrow x_n$ and $(x_n \bullet y, \alpha) \rightarrow *$ and $(x_n \bullet y, \beta) \rightarrow p'$ with $p = p' - \alpha$. For this $(x, \alpha) \rightarrow x_1 \wedge \cdots$ $\wedge (x_{n-1}, \alpha) \rightarrow x_n$ and $(x_n, \alpha) \rightarrow *$, by 9. Now $(x_n \bullet y, \beta) \rightarrow p'$ in three ways by 9, 10, and 11.

c.2.1) $\beta \notin L_y$ and 9 gives $(x_n, \beta) \rightarrow p''$ with $p' = p'' \bullet y$. By 17.1 $(x-\alpha, \beta) \rightarrow p''-\alpha$ and as $\beta \notin L_y$, by 9 $(x-\alpha \bullet y, \beta) \rightarrow (p''-\alpha) \bullet y$. Now $(p'' \bullet y)-\alpha \sim (p''-\alpha) \bullet y$ by an inductive argument on the indexed definition of $\sim$, as required.

c.2.2) $\beta \notin L_x$ follows similarly using 10.

c.2.3) $\beta \in L_x \cap L_y$ follows using 11.

d) $((x-\alpha) \bullet y, \beta) \rightarrow p$ implies either

d.1) $(x-\alpha, \beta) \rightarrow p'$, $\beta \notin L_y$ with $p = p' \bullet y$, by 9. This arises with either .

d.1.1) $(x, \beta) \rightarrow p''$ where $p' = p''-\alpha$. As $\beta \notin L_y$ 9 gives $(x \bullet y, \beta) \rightarrow p'' \bullet y$ and 16 gives $((x \bullet y)-\alpha, \beta) \rightarrow$ $(p'' \bullet y)-\alpha$. As $p = (p''-\alpha) \bullet y$ then $(p''-\alpha) \bullet y \sim$ $(p'' \bullet y)-\alpha$ by induction on definition of $\sim$.

d.1.α) $\exists$ n where $(x, \alpha) \rightarrow x_1 \wedge \cdots \wedge (x_{n-1}, \alpha) \rightarrow x_n$ and $(x_n, \alpha) \rightarrow *$ and $(x_n, \beta) \rightarrow p''$ where $p' = p''-\alpha$. As $\alpha, \beta \notin L_y$, $(x \bullet y, \alpha) \rightarrow (x_1 \bullet y) \wedge \cdots \wedge (x_{n-1} \bullet y, \alpha)$ $\rightarrow x_n \bullet y$ and $(x_n \bullet y, \alpha) \rightarrow *$ and $(x_n \bullet y, \beta) \rightarrow p'' \bullet y$. By 17.1, $((x \bullet y)-\alpha, \beta) \rightarrow (p'' \bullet y)-\alpha$. Now

$p = (p''-\alpha) \bullet y \sim (p'' \bullet y)-\alpha$, by induction on definition of $\sim$.

d.2) $(y, \beta) \to p'$, $\beta \notin L_x$, follows in a simpler way.

d.3) $(x-\alpha, \beta) \to p'$ and $(y, \beta) \to p''$. Follows in a similar manner to d.1).

We now have $(x \bullet y)-\alpha \sim (x-\alpha) \bullet y$ for internal α such that $\alpha \notin L_y$. The case with $\alpha \notin L_x$ follows by symmetry.

Case external α:- Again assume x and y in CNF as laws $[- \oplus]$ and $[\oplus \bullet]$ can be used to get this.

a) $((x \bullet y)-\alpha, \beta) \to *$ implies by 18 that $(x \bullet y, \beta) \to *$ Either $(x, \beta) \to *$ or $(y, \beta) \to *$.

a.1) $(x, \beta) \to *$ and by 18, $(x-\alpha, \beta) \to *$ and $(x-\alpha \bullet y, \beta) \to *$, by 12.

a.2) $(y, \beta) \to *$ and by 13, $(x-\alpha \bullet y, \beta) \to *$.

b) $(x-\alpha \bullet y, \beta) \to *$. By 12 and 13 either

b.1) $(x-\alpha, \beta) \to *$ in which case $(x, \beta) \to *$, by 18. Then $(x \bullet y, \beta) \to *$ by 12 and $((x \bullet y)-\alpha, \beta) \to *$ by 18, as α is external.

b.2) $(y, \beta) \to *$ and so $(x \bullet y, \beta) \to *$ by 13, and $((x \bullet y)-\alpha, \beta) \to *$ by 18.

c) $((x \bullet y)-\alpha, \beta) \to p$ implies by 16 that $(x \bullet y, \beta) \to p'$ where $p = p'-\alpha$.

c.1) By 9, $(x, \beta) \to p''$ and $\beta \notin L_y$ with $p' = p'' \bullet y$ then $(x-\alpha, \beta) \to p''-\alpha$, by 16 and $(x-\alpha \bullet y, \beta) \to (p''-\alpha) \bullet y$ by 9.

Now $p = (p'' \bullet y)-\alpha \sim (p''-\alpha) \bullet y$ by induction on definition of $\sim$.

c.2) By 10, $(y, \beta) \rightarrow p''$ and $\beta \notin L_x$. As for c.1) using 10.

c.3) By 11, $(x, \beta) \rightarrow p''$ and $(y, \beta) \rightarrow p'''$. As for c.1) using 11.

d) $((x-\alpha) \bullet y, \beta) \rightarrow p$ implies that

d.1) $(x-\alpha, \beta) \rightarrow p'$ and $\beta \notin L_y$, with $p = p' \bullet y$. By 16 we have $(x, \beta) \rightarrow p''$ where $p' = p''-\alpha$. By 9, $(x \bullet y, \beta) \rightarrow p'' \bullet y$ and by 16 $((x \bullet y)-\alpha, \beta) \rightarrow (p'' \bullet y)-\alpha$. Now $p = (p''-\alpha) \bullet y \sim (p'' \bullet y)-\alpha$ by induction on definition of $\sim$.

d.2) $(y, \beta) \rightarrow p'$ and $\beta \notin L_x$, with $p = (x-\alpha) \bullet p'$ and $((x \bullet y), \beta) \rightarrow x \bullet p'$, by 10. $(x \bullet y)-\alpha, \beta) \rightarrow (x \bullet p')-\alpha$ by 16. Now $p = (x-\alpha) \quad p' \sim (x \bullet p')-\alpha$ by induction on $\sim$.

d.3) $(x-\alpha, \beta) \rightarrow p'$ and $(y, \beta) \rightarrow p''$ follows in a similar fashion to d.1) using 11.

Now have that $(x \bullet y)-\alpha \sim (x-\alpha) \bullet y$ for external α. The case with $\alpha \notin L_x$, i.e., $(x \bullet y)-\alpha \sim x \bullet (y-\alpha)$ follows by symmetry.

[- ⊕] Show $(x \oplus y)-\alpha \sim (x-\alpha) \oplus (y-\alpha)$ for CNF x and y.

a) $((x \oplus y)-\alpha, \beta) \rightarrow *$ if either

a.1) by 19, $(x-\alpha, \beta) \rightarrow *$ and so $((x-\alpha) \oplus (y-\alpha), \beta) \rightarrow *$, by 6.2.

or a.2) by 20, $(y-\alpha, \beta) \rightarrow *$. As for a.1)

b) $((x-\alpha) \oplus (y-\alpha), \beta) \rightarrow *$ implies that either

b.1) by 6.2, $((x-\alpha), \beta) \rightarrow *$ hence, $((x-\alpha) \oplus (y-\alpha), \beta) \rightarrow *$, by 19.

b.2) by 7.2, $((y-\alpha), \beta) \rightarrow *$. As for b.1).

c) $((x \oplus y) -\alpha, \beta) \rightarrow p$ implies either

c.1) for internal or external α :- by 16, $(x \oplus y, \beta) \rightarrow p'$ with $p = p'-\alpha$. Either

c.1.1) $(x, \beta) \rightarrow p'$ by 6.1, and
$(x-\alpha, \beta) \rightarrow p'-\alpha$ by 16, and
$((x-\alpha) \oplus (y-\alpha), \beta) \rightarrow p'-\alpha$ by 6.1.

c.1.2) $(y, \beta) \rightarrow p'$ by 7.1. As for c.1.1).

c.2) for internal α only:- by 17.1 $\exists$ some n such that $(x \oplus y, \alpha) \rightarrow p_1 \wedge (p_1, \alpha) \rightarrow p_2 \wedge \cdots \wedge (p_n, \beta) \rightarrow p'$ with $p = p'-\alpha$. Either

c.2.1) by 6.1, $(x, \alpha) \rightarrow p_1$ and by above $(x-\alpha, \beta) \rightarrow p$ and
$((x-\alpha) \oplus (x-\alpha), \beta) \rightarrow p'$, by 6.1. Or

c.2.2) by 7.1, $(y, \alpha) \rightarrow p_1$. As for c.2.1).

d. $((x-\alpha) \oplus (y-\alpha), \beta) \rightarrow p$ which implies that either

d.1) by 6.1, $(x-\alpha, \beta) \rightarrow p$. Now either

d.1.1) for internal and external α:- by 16, $(x, \beta) \rightarrow p'$ where $p = p'-\alpha$. Now $(x \oplus y, \beta) \rightarrow p'$ by 6.1 and
$((x \oplus y)-\alpha, \beta) \rightarrow p$ by 16.

d.1.2) for internal α only:- by 17.1 $\exists$ some n such that $(x, \alpha) \to p_1 \wedge \cdots \wedge (p_n, \beta) \to p'$ where $p = p'-\alpha$. By 6.1, $(x \oplus y, \alpha) \to p_1$ and $((x \oplus y)-\alpha, \beta) \to p$ follows by 17.1.

d.2) by 7.1, $(y-\alpha, \beta) \to p$. As for d.1). We now have that $(x \oplus y)-\alpha \sim (x-\alpha) \oplus (y-\alpha)$.

[--] Show $x-\alpha-\beta \sim x-\beta-\alpha$. Case CNF x; that is, $x = \sum_i \gamma_i x_i$.

a) Case internal α and β:- $(x-\alpha-\beta, \gamma) \to *$ if either, for CNF $x-\alpha$,

a.1) by 15, $(x-\alpha, \gamma) = \{*\}$ and $(x-\alpha, \beta) = \{*\}$ for CNF $x-\alpha$. Now $(x-\alpha, \gamma) = \{*\}$ if either

a.1.1) by 15, $(x. \gamma) = \{*\}$ and $(x, \alpha) = \{*\}$. Now $(x-\alpha, \beta) = \{*\}$ if $(x, \beta) = \{*\}$ since $(p, \alpha) = \{*\}$. By 15, $(x-\beta, \gamma) = \{*\}$ and $(x-\beta, \alpha) = \{*\}$ and $(x-\beta-\alpha, \gamma) = \{*\}$, by 15 again.

a.1.2) by 17.2, $\exists$ n such that $(x, \alpha) \to p_1 \wedge \cdots \wedge (p_n, \alpha) \to *$ and $(p_n, \gamma) \to *$. As $(x-\alpha, \beta) \to *$ and $(x, \alpha) \to p_1$, i.e., $(p, \alpha) \neq \{*\}$, $\exists$ m such that $(x, \alpha) \to q_1 \wedge \cdots \wedge (q_m, \alpha) \to *$ and $(q_m, \beta) \to *$. Then, $(x-\beta, \gamma) \to *$ by 17.2. As $(x, \alpha) \to q_1 \wedge \cdots$ then $(x-\beta, \alpha) \to q_1-\beta \wedge \cdots \wedge (q_m-\beta, \alpha) \to *$ and $(q_m-\beta, \gamma) \to *$, by 16 and 15 or 19 and 20 depending on whether q_m is in CNF or DNF. This gives $(x-\beta-\alpha, \gamma) \to *$ by 17.2.

a.2) by 17.2 $\exists$ some n such that

$(x-\alpha,\beta) \to p_1 \wedge \cdots \wedge (p_n,\beta) \to * \wedge (p_n,\gamma) \to *$ and p_n is in CNF. Suppose $n = 1$, then $(p_1,\beta) \to *$ and $(p_1,\gamma) \to *$ and p_1 is in CNF. Now $(x-\alpha,\beta) \to p_1$ in one of two ways:-

a.2.1) by 16, $(x,\beta) \to p'$ and $p_1 = P'-\alpha$. As $p'-\alpha$ is in CNF then so is p'. Now $(p'-\alpha,\beta) \to *$ either by

a.2.1.1) $(p',\beta) = \{*\}$ and $(p',\alpha) = \{*\}$ and $(p'-\alpha,\gamma) \to *$ by 15 with $(p',\gamma) = \{*\}$ and $(p',\alpha) = \{*\}$ and by 17.2 and $(x,\beta) \to p'$, $(x-\beta,\gamma) = \{*\}$ and $(x-\beta,\alpha) = \{*\}$ and $(x-\beta-\alpha,\gamma) \to *$, by 15.

or a.2.1.2) $\exists$ r such that, by 17.2, $(p',\alpha) \to p'_1 \wedge \cdots \wedge (p'_r,\alpha) = \{*\}$ and $(p'_r,\beta) = \{*\}$ and $(p'-\alpha,\gamma) \to *$ with some m such that $(p',\alpha) \to p'_1 \wedge \cdots \wedge (p'_m,\alpha) = \{*\}$ and $(p'_m,\gamma) = \{*\}$. Now as $(x,\beta) \to p'$ we have $(x-\beta,\alpha) \to p'_1$. By 16, we have $(p'_1-\beta,\alpha) \to p'_2-\beta \wedge \cdots \wedge (p'_r-\beta,\alpha) = \{*\} \wedge (p'_m-\beta,\gamma) = \{*\}$. Then $r = m$ as $(p'_r,\alpha) = \{*\}$ and $(p'_m,\alpha) = \{*\}$ and 17.2 gives $(x-\beta-\alpha) \to *$, as required.

a.3) for DNF x-α, then x-α = x_1-α $\oplus$ x_2-α for some x_1 and x_2, by axiom [- $\oplus$]. Then (x-α-β, γ) $\rightarrow$ * if either (x_1-α-β, γ) $\rightarrow$ * or (x_2-α-β, γ) $\rightarrow$ * by 19 and 20, and we can assume x_1 and x_2 are in CNF or else we repeat the above argument. Then it is only necessary to show (x'-α-β, γ) $\rightarrow$ * for some CNF x' and the other cases follow by symmetry.

Case internal α and external β:-

(x-α-β, γ) $\rightarrow$ * if (x-α, γ) = {*} for CNF x-α. Follows a.1) above. Again we only need consider the CNF case.

Case external β and internal α:-

Follows by only considering certain of the cases above.

Case external α and external β:-

Here (x-α-β, γ) $\rightarrow$ * if (x-α, γ) = {*} which arises when (x, γ) = {*} by 18. Then (x-β, γ) = {*} and (x-β-α, γ) = {*} again by 18 as we only need consider CNF expression. Hence (x-β-α, γ) $\rightarrow$ *. We now have completed part a) of the proof.

Part b) follows by symmetry.

c) (x-β-α, γ) $\rightarrow$ p implies either

c.1) by 16, (x-β, γ) $\rightarrow$ p' where p = p' - α implies either

c.1.1) by 16, (x, γ) $\rightarrow$ p'' where p' = p'' - β. By 16, (x-α, γ) $\rightarrow$ p''-α and (x-α-β, γ) $\rightarrow$ p''-α-β, by 16 again. Now p = p''-β-α $\sim$ p''-α-β by an inductive argument on $\sim$.

c.2) by 17.1, $(x-\beta, \alpha) \rightarrow p_1 \wedge (p_1, \alpha) \rightarrow p_2 \wedge \cdots \wedge (p_n, \gamma) \rightarrow p'$ where $p = p'-\alpha$. Suppose $n = 1$, then $(p_1, \gamma) \rightarrow p'$. Now $(x-\beta, \alpha) \rightarrow p_1$ if either

c.2.1) by 16, $(x, \alpha) \rightarrow p_1'$ with $p_1 = p_1'-\beta$. As $(p_1'-\beta, \gamma) \rightarrow p'$ then by 16 $(p_1', \gamma) \rightarrow p''$ where $p' = p''-\beta$. As $(x, \alpha) \rightarrow p_1'$ and $(p_1', \gamma) \rightarrow p''$ we get $(x-\alpha, \gamma) \rightarrow p''-\alpha$ by 17.1 and $(x-\alpha-\beta, \gamma) \rightarrow p''-\alpha-\beta$, by 16. Now $p = p''-\beta-\alpha \sim p''-\alpha-\beta$ by our inductive argument on $\sim$. Or

c.2.2) by 17.1, $\exists$ some m where $(x, \beta) \rightarrow x_1 \wedge \cdots \wedge (x_m, \alpha) \rightarrow p_1'$, where $p_1 = p_1'-\beta$ and by 16 $(x-\alpha, \beta) \rightarrow x_1 \wedge (x_1-\alpha, \beta) \rightarrow x_2 \wedge \cdots \wedge (x_{m-1}-\alpha, \beta) \rightarrow x_m$. As $n = 1$, $(p_1'-\beta, \gamma) \rightarrow p'$ and so $(p_1', \gamma) \rightarrow p''$ by 16. As we also have $(x_m, \alpha) \rightarrow p_1'$ then 17.1 gives $(x_m-\alpha, \gamma) \rightarrow p''-\alpha$ and by the $(x-\alpha, \beta) \rightarrow x_1$ string above, and 17.1 we get $(x-\alpha-\beta, \gamma) \rightarrow p''-\alpha-\beta$. Now $p = p''-\beta-\alpha$ and we use an inductive argument to get $p''-\beta-\alpha \sim p''-\alpha-\beta$. We

We have shown b.2) for $n = 2$ by showing for the two ways in which $(x-\beta, \gamma) \rightarrow p_1$. For any i, $(p_i, \alpha) \rightarrow p_{i+1}$ in a similar way, and so for any n we will have $2 \times n$ cases where each is like c.2.1) or c.2.2). Then for any n, we have c.2).

d) $(x-\alpha-\beta, \gamma) \rightarrow p$ follows exactly as c) by symmetry.

We now have $p-\alpha-\beta \quad \sim \quad p-\beta-\alpha$, as required.

REFERENCES

[1] R. Campbell and A. Habermann, "The Specification of Process Synchronisation by Path Experssions," Lecture notes in Computer Science, Vol. 16, Springer-Verlag, 1974

[2] M. Hennessy and R. Milner, "On Observing Nondeterminism and Concurrency," Lecture Notes in Computer Science, Vol. 85, Springer-Verlag, 1980

[3] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM 17, 10, 1974

[4] C.A.R. Hoare, "Communicating Sequential Processes," Comm. ACM 21, 8, 1978

[5] C.A.R. Hoare, S. Brookes and W. A. Roscoe, "A Theory of Communicating Sequential Processes," Programming Research Group, Oxford University

[6] P.J. Landin, "The Mechanical Evaluation of Expressions," Computer J. 6, 4, 1964

[7] G. Milne and R. Milner, "Concurrent Processes and their Syntax," J. ACM 26, 2, 1979

[8] R. Milner, "A Calculus of Communicating Systems," to be published by Springer-Verlag in Lecture Notes in Computer Science

[9] G. Plotkin, "Call-by-name, call-by-value and the λ-calculus," J. Theoretical Computer Science 1, 1, 1974

ERRATA

Page 7

line 4 - "$\alpha r_1 + \beta r_2$" should read "$\alpha r + \beta s$"

Page 8

line 2 - "$\oplus$ roots" should read "0 roots"

para. 3, line 6 - "event x" should read "event α"

Page 9

diagram 1 - The α on the left-hand-side of the leftmost box should be replaced by a γ.

Page 16

para. 2, line 1 - "$(\alpha,p,\alpha) \rightarrow p$" should be replaced by "$(\alpha p,\alpha) \rightarrow p$"

Page 43

line -6 - "$g\ell(I \ominus 1)\ p\ell(i \ominus 1)\ F_i$" should read

"$g^{\ell}_{(i \ominus 1)}\ p^{\ell}_{(i \ominus 1)}\ F_i$"

Page 45

line 8 - "$r_i \ominus 1$" should read "$r_{i\ominus 1}$"

Page 52

after the last line add in the following:

$$[/\oplus]\ \left(\sum_i p_i\right)[\beta/\alpha] = \left(\sum_i (p_i[\beta/\alpha])\right)$$

Page 71

line 4 - "the calculus" should be replaced by " the λ calculus"