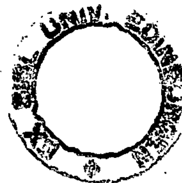
A MATHEMATICAL MODEL OF CONCURRENT COMPUTATION

George Johnstone Milne

DOCTOR OF PHILOSOPHY

UNIVERSITY OF EDINBURGH

1977



ABSTRACT

A mathematical model is presented in which we can understand and discuss the behaviour of concurrent computing agents such as interconnecting hardware modules, operating system components and parallel programs. It is shown that it is natural to represent computing agents in a "value-passing" framework rather than by using a global store. A computing agent will be modelled by a process which is the set of communication capabilities which it can make with some external environment. Operations on processes including those involving synchronised communication are described. These reflect the way in which composite agents are constructed from their intercommunicating sub-agents.

Examples of how processes may be used to model both hardware and software computing agents are given. Proof techniques involving computation induction which allows us to reason about processes and the agents they represent in a concise manner are also given, together with a uniform method of modelling the scheduling of a number of computing agents. Two scheduling techniques involving this method are presented and they are shown to be equivalent. This result is used in a final example where we use the process model to produce two equivalent denotational semantics for a concurrent programming language involving path expressions.

CONTENTS

Chapter 0: INTRODUCTION 1

Chapter 1: THE PROCESS MODEL 12

1.1 The process domain 12

1.2 Definitions 17

1.3 Processes 19

1.4 Communication between processes 19

1.5 Other process operators 22

Chapter 2: PROCESSES AS MODELS OF COMPUTING AGENTS 25

2.1 Registers and memories 25

2.2 Other data structures - queues and stacks 28

2.3 Semaphores 29

2.4 A hardware example 31

Chapter 3: THE DOMAIN 40

3.1 The weak powerdomain 40

3.2 Choice of domain 42

3.3 Continuity of the set-forming operation 43

Chapter 4: PROOFS IN THE PROCESS MODEL 46

4.1 Induction techniques 46

4.2 The elementwise technique 48

4.3 An example concerning queues 50

4.4 Process function composition 54

4.5 Proof techniques involving the $\|$ combinator 57

4.6 The card reader theorem 61

Chapter 5:	PROCESSES AS A FLOW ALGEBRA	67
5.1	Flow algebras	68
5.2	Processes are a flow algebra	69
Chapter 6:	TERMINATING PROCESSES	86
6.1	Representing termination	86
6.2	Computation trees	87
6.3	Combinators for terminating processes	90
6.4	Properties of termination combinators	93
6.5	Termination and deadlock	96
Chapter 7:	SCHEDULING PROCESSES	100
7.1	The scheduling technique	100
7.2	Requirements for a scheduler	103
7.3	Acceptors	105
7.4	Linearisation of schedulers	110
Chapter 8:	PRODUCING SCHEDULERS AND USING THE SCHEDULING TECHNIQUE	114
8.1	The second readers/writers problem	114
8.2	Adding scheduling lines to processes	117
8.3	Synchronisers in the second readers/writers problem	120
8.4	The dining philosophers problem	122
Chapter 9:	MULTI-LEVEL SCHEDULING	127
9.1	Multiple schedulers	127
9.2	The & combinator	128

9.3 Multi-level scheduling 130

9.4 Denotational semantics of PPL 134

9.5 Alternative semantics for PPL 139

CONCLUDING REMARKS 143

APPENDIX 145

REFERENCES 174

CHAPTER 0

INTRODUCTION

Recent advances in hardware technology, resulting in a reduction in the cost of processors, suggest that systems of computing agents containing many processors linked together will become commonplace in the future. It is therefore desirable that computation among these intercommunicating agents be fully understood and allowed to influence the design of computer systems and concurrent programming languages. In this dissertation a mathematical model is presented in which we reason about concurrency independent of implementation constraints and the quantitative considerations such as efficiency which implementation demands.

0.1 Aims of the work

The work reported in this dissertation endeavours to produce a mathematical model of concurrent computation based on the notion of communication and to justify this model in an empirical manner. The model is required to encapsulate the fundamental concepts of concurrency in such a way that we may comprehend the behaviour of concurrent computation, and reason formally about it. We also desire to use an algebraic approach when modelling computing agents; the behaviour of composite agents should be represented by the composition of those objects which represent the subagents. If we restrict our attention to a minimal set of composition operations we may then aim to model large systems of communicating computing agents whose behaviour is not at all obvious (due to concurrency) by modelling their less complicated and hence better understood components. The operations must also be capable of constructing

a model for the whole system of agents in a well-defined, intuitive manner, reflecting the way in which such systems are built; by connecting their components. This approach utilises the semantic concepts of Scott and Strachey [Sco 2] by modelling a computing agent by an abstract object; its meaning. We call such an object a process, which is similar in many ways to a function. The meaning of a system or net of agents is also a process and will be the homomorphic image of the net, where the homomorphism is from an algebra of nets to an algebra of processes both of which belong to a certain category, the category of flow algebras.

This aim of the model will enable us to relate a real computing agent with a mathematical object, similar in degree of abstraction to a function, which expresses its behaviour. We wish our model to be equally applicable to both hardware and software agents since we require to model that behaviour visible to the "outside world" and not the intensional details of how the behaviour was arrived at. Whether an agent is implemented by hardware, software or some combination of both we may wish to model it at varying levels of detail. Our model should satisfy this requirement. It should also be capable of being used as a specification language in which the desired behaviour of agents to be implemented may be described. This is particularly important for complex hardware agents since we may then be able to model the implemented agent and construct a proof that the implementation satisfies its design. Our process algebra will also be a semantic domain in which the meaning of programming languages involving concurrency can be represented. Hence existing language features for concurrency should be suitably modelled in our framework, but we should point out that the design of language features for concurrency is not an aim of this work although the

detailed study of concurrency should influence such design.

Our model will be designed so that certain characteristics of concurrency, such as deadlock and scheduling, will be dealt with in a straightforward, intuitive fashion capturing the underlying notions of such features. For example, existing language constructs for dealing with scheduling such as semaphores (Dijkstra [Dij 1]), conditional critical regions (Hoare [Hoa 2]), monitors (Brinch Hansen [Bri 1]), and path expressions (Campbell and Habermann [Cam]) should all be modelled as instances of some general scheduling technique. Another requirement of the model is that we should be able to use it to reason about computing agents without extra entities such as the assertions used in the proof techniques for serial algorithms used by Floyd [Flo] and Hoare [Hoa1] . We aim to carry out such reasoning on agents in terms only of their meanings, which are processes in our model. Proof techniques will also be required to allow us to reason lucidly and concisely about our computing agents. This will utilise the work of Scott [Sco 1] on induction for recursive functions suitably extended to cater for concurrency.

0.2 Background

The concept of process as given by Bekic [Bek] and Milner [Mil 3] was the starting point in the development of our model. Our processes are thus considered to be objects with input and output properties while concurrency is represented by the non-determinate interleaving of concurrent execution sequences. Non-determinism arises in parallelism since the relative speed of concurrent computing agents is left unspecified in our model; indeed we do not attempt to model "real" time but rather the sequence

in which computation takes place. Rather than using an external object such as Milner's oracle [Mil 3] to arbitrate among possible execution sequences we require the meaning of concurrent computation to be a set of all possible interleavings of component execution sequences. We use the work of Scott [Sco 1] on continuous functions and its extension to non-determinism by the powerdomain constructions of Plotkin [Plo 1] and Smyth [Smy]. These constructions allow us to define processes as sets of communication capabilities, thus taking communication as the primitive notion of our model. Communication between two computing agents is then modelled by combining two processes (to produce a new process) using an operator which encapsulates the notion of synchronised communication between agents as well as non-deterministic interleaving. We take the position that communication between two processes occurs only when both processes request it. This approach is due to Milner and differs from that of Kahn [Kah 1] for instance, where a buffer exists on each communication line removing "timing" constraints and the need for processes to be synchronised.

Processes and the minimal set of operators on them are shown in this dissertation to form an algebra; a Flow Algebra. Following the work of Goguen, Thatcher, Wagner and Wright [Gog] on initial algebra semantics we define a Flow Algebra of Nets (similar to systems of agents where we indicate only where and not how communication takes place) which is shown by Milner [Mil 4] to be initial in a category of Flow Algebras; hence our Flow Algebra of Processes is one possible semantics (via a homomorphism of flow algebras) for the syntactic net algebra. The algebra of processes thus gives meaning to computation among communicating agents.

The notation used to denote processes is in part due to Plotkin [Plo 1] while the powerdomain construction we actually use is that of Smyth [Smy], which is a development of Plotkin's work on powerdomains, slightly extended by Milner [Mil 1]. Previous work associated or related to that reported in this dissertation is mentioned in the following section.

0.3 Existing models of concurrent computation

A number of existing models of concurrent computation are mentioned here to indicate the present state of the art.

Petri Nets and Control Nets [Yoe] are models used to capture the synchronisation and flow of control properties of systems of computing agents which involve some degree of concurrency. A Petri Net consists of place and transition nodes joined alternately by directed arcs (which is known as a bipartite Petri Graph) together with a marking of place nodes. Place nodes represent conditions of the system which either do or do not hold, while transition nodes correspond to possible changes of conditions; that is, changes in the state of the system. The marking of places, via a function m: places $\rightarrow \{0, 1\}$ for Boolean Petri Nets or m: places $\rightarrow N$ for Integer Petri Nets, indicates those conditions which hold. A given marking of the net evolves into another marking by using firing conventions which together with the markings interpret, or give meaning to, the graph.

Control Nets are similar to Petri Nets but contain linkage places where subnets can be attached to one another to form composite control nets. Basic control nets may be used to represent agents in a concurrently computing system while the joining together of such nets represents the construction of the system. Places are here

used to indicate when single well-defined computing tasks belonging to agents have completed, i.e. the preceding transitions have fired. Both forms of net are used to model the flow of control among a number of computing agents. Properties such as the possibility of deadlock in a system may be proved in these net models using axioms to represent given markings of the net together with an axiomatisation of the firing rules. With both the Petri and Control Net models the meaning of a graph is given by an interpretation which indicates how computation among a system of agents progresses. Such an interpretation corresponds to a process in our model, which gives meaning to an object in the Flow Algebra of Nets. But only properties of the synchronisation of agents may be represented by these models; they do not indicate what is computed by a system of agents but rather in what order computation proceeds.

A model which is similar to those above in that it captures the flow of control in a system, but also represents the computation which takes place, is the Data Flow Schema model of Dennis [Den]. To model a system of agents using such a computation schema we require a Data Flow Graph and an interpretation. The graph consists of both operational and decision nodes joined by directed arcs; the former parameterised on values and the latter on booleans. Such a fixed graph defines the "type" of these elements and how they are connected but not how values are produced. The interpretation gives meaning to the graph; it assigns functions and predicates to the operational and decision elements respectively. Agents of a system are modelled by functions while the synchronisation properties are encapsulated by predicates. Only systems with determinate behaviour (due to synchronisation) are modelled and this enables the meaning of such systems to be represented

as a function from a tuple of input values to a tuple of output values, but restricts those features of concurrency which can be modelled. Two systems of computing agents will then be equivalent if their schema determine the same function. The behaviour of a system as execution progresses is modelled by a sequence of snapshots where each snapshot shows the flow graph with certain boolean and data values placed on arcs. The decision nodes fire in the same way as marked places in a Petri Net, and cause the evolution of snapshots. As with Control Nets we can construct Flow Graphs, representing the semantics of a concurrent program say, by composing sub-Flow Graphs which model parts of the program. But no combinator is defined for composition of nets in either the Control Net or the Data Flow Schema model; this lack of an algebraic approach in constructing a model results in the model not reflecting how systems of agents are constructed. Our process model does reflect the way in which systems are built. Another failing of the Data Flow model is that we are limited in how we may use it to reason about parallel computation; only deterministic computation may be considered.

A model of computation which has some similarity to our process model is the actor model of Hewitt [Hew]. The behaviour of a computing agent may be modelled by an activator, which is a set of events totally ordered with respect to their occurrence. The behaviour of a system of agents is then given by a partially ordered set of all events belonging to component activators. An event is the passing of a message actor to a target actor. Actors are rather vague objects but a target actor may be thought of as a LISP procedure while the message actor is some sort of value. A continuation, similar to that used in denotational semantics by

Wadsworth [Str], may be part of the message actor. This can be used by the target actor as a message destination to which the "result" of the event may be passed as a message, or it may be ignored and the target may transmit messages to some other actor which it favours. The behaviour of a system of actors is described by a set or sets of partially ordered events, called histories. A system of computing agents is then modelled by the behaviour of some system of actors.

In [Gre 2] Greif uses the actor model to specify the semantics of programs involving concurrency. The synchronisation properties of parallel programs are modelled by using partial orders to explicitly state the occurrence of events. Parallelism is handled by the non-deterministic interleaving of concurrent operations. Thus a number of histories may be needed to define the semantics of a concurrent program; one history for each interleaving. Properties of particular actors may be specified by axioms, and these must be satisfied by histories containing events involving those actors. Axioms such as these together with synchronisation invariants may be used to prove that concurrent programs, or systems of agents, satisfy various required synchronisation properties. These invariants are shown to hold over the partial orders of all the histories used to specify the meaning of the program, or the behaviour of the system. The actor model resembles the process model in that communication is primitive and computing agents such as registers may be modelled by the actions they can take; input and output of values. When using the actor model to reason about computation, proofs are carried out on properties of objects in the model rather than on these objects themselves. The actor model suffers from not being defined mathematically and in that too many primitive concepts such as actors, events, activators etc., are needed by it. The result is a rather imprecise

model lacking the conceptual simplicity which makes a model well understood and easily usable.

A number of approaches have been made in the production of suitable semantics for programming languages which involve some degree of concurrency. Those of Dennis and Greif as mentioned above are just two, but others exist.

Kahn [Kah 1] models concurrent computationby using buffers and he restricts his attention to strictly determinate programs. Buffers being determinate in behaviour preserve this property. The semantics of such a program is given by a function from the history of input communication lines to the history of output communication lines. The history of a communication line is the possibly infinite traffic of information of the same type witnessed by an observer on the line. This approach is similar in some respects to our model (and that of R. Milne which is mentioned below) since abstractions (i.e. functions) are discussed in proofs, unlike the axiomatic approach where constructs such as invariants are used. In Kahn and MacQueen [Kah 2] a language for creating networks of processes, and the channels which interconnect them, is produced using the notions of Kahn's model. No synchronisation before communication is necessary since buffers are used, and waiting may occur on empty buffers. The syntax of the language restricts programs to being determinate, hence it is irrelevant whether they are implemented on many or single processor machines.

In [Mil 2] R. Milne mentions how denotational semantics in the style of Scott and Strachey [Sco 2] may be extended to programs involving parallelism.

In such programs execution of one portion of the program may be interleaved with the execution of another portion at the end of any of the "indivisible" operations. These operations are not displayed

in the syntax of the language but will be apparent in their semantics. At any point in the body of a semantic equation there is a continuation. When execution passes from one of the concurrent operations to another the continuation corresponding to the original operation must be preserved since control will return to it, and the continuation of the other takes over. The flow of control is determined by a "roster" (similar to Milner's oracle [Mil 3]) which is interrogated by a function at the end of each indivisible operation to determine which continuation is to be considered next. This yields the next indivisible operation, so allowing this scheduling function to be invoked again for a further continuation and so on. Proof techniques involving denotational semantics which were developed for sequential programs apply equally well to this treatment of parallel programs. A criticism of this approach is that rosters are rather concrete objects external to the program, which impose determinism onto something which may be inherently non-deterministic. This criticism also applied to Milner's oracle approach [Mil 3].

Owicki and Gries [Owi 2] provide an axiomatic semantics in the style of Hoare [Hoa 1] for parallel programming languages containing conditional critical regions together with axiomatic techniques for proving certain properties of these programs, such as partial correctness, termination and deadlock. In [Owi 1] Owicki extends this treatment to programs involving monitors. These techniques involve auxiliary variables being added to parallel programs to enable correctness to be proved, and this requirement for added entities is a general criticism one may make of axiomatic proof techniques. In our process model correctness proofs are constructed utilising processes (the meaning of computing agents) which themselves enter into the proofs.

The axiomatic approach as used by Owicki, Greif and in the Petri Net model relies on extra mathematical objects such as assertions, inference rules, invariants, and event histories entering into proofs. Such models therefore differ fundamentally from ours.

Recent language proposals by Hoare [Hoa 3], Wirth [Wir 1] and Brinch Hansen [Bri 2] provide a means by which concurrent computation may be expressed in a uniform, well-defined way. This compares favourably with previous attempts at producing languages for concurrency which usually involved ad hoc extensions of existing serial languages; a parallel command along with language features (such as monitors) to synchronise access to shared data structures was all that was provided. The work of Hoare [Hoa 3] in particular contains close parallels with our model; he treats communication among concurrent computations as the basic notion of his language, which involves input and output. This reliance on communication rather than assignment to a global memory is a main feature of our model. It seems to reflect much more naturally (than a global store) the behaviour which takes place among a system of communicating agents which compute in parallel.

CHAPTER 1

THE PROCESS MODEL

The model is constructed to enable the behaviour of computing agents which involve concurrency to be represented in an unambiguous mathematical framework on which we can reason formally.

Computing agents are modelled by _processes_, which are members of a _process domain_. The process domain, which is described in this chapter, involves "value-passing" rather than a global store.

Value-passing as representing the behaviour of parallel computation is also used by Kahn [Kah1], where he gives the semantics of a simple parallel programming language; but we do not treat the queueing of values on communication lines as a primitive notion in our model.

The only primitive notion in our model is _synchronised communication_. A process will consist of a set of _communication capabilities_, and communication between two processes will only take place when both processes contain "suitably matched" capabilities.

Concurrency between agents is modelled by a process combinator which allows two processes to communicate if possible and forms a new set of communication capabilities from the component capabilities using arbitrary interleaving. We therefore explicate parallelism among agents by non-determinism; we interleave all possible communication sequences of two processes in all possible ways.

1.1 The process domain

The agents which we intend to model may be either hardware agents, software agents or some combination of both. Typically these agents

consist of a number of components which compute concurrently and which communicate among themselves and an external environment - the "outside world".

Such an agent is the DEC card reader described in [Dig] which consists of four distinct, intercommunicating units: a status register, a data buffer register, the actual reading mechanism and the card reader driver. The first three components are hardware whilst the latter, which controls the action of the hardware by interrogation of the status register, can be thought of as a program.

This card reader may be pictured by the following <u>net</u>:

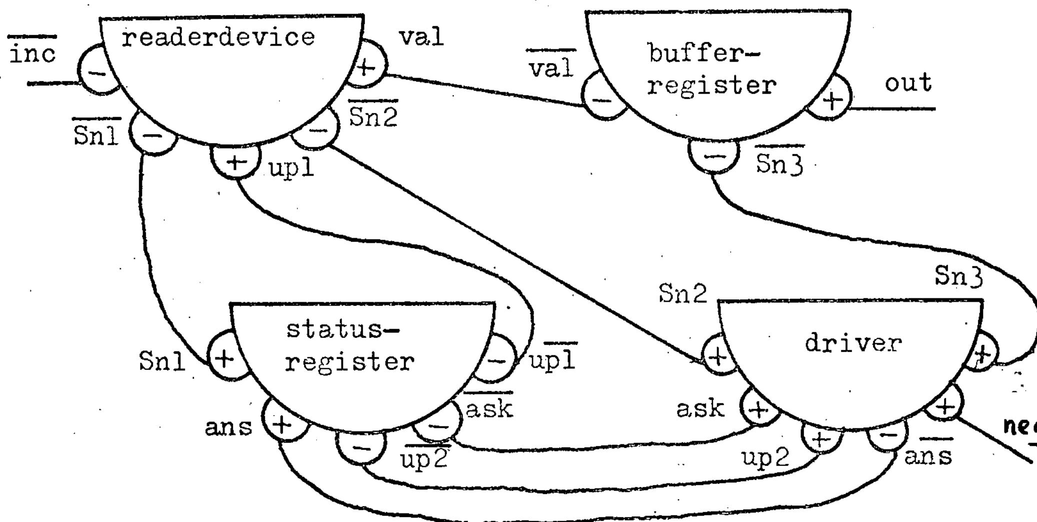


FIGURE 1.1

The specification of the card reader and the detailed interpretation of a net need not concern us here; the card reader will be met again in the next chapter and Milner describes an algebra of nets in [Mil 4].

This net is constructed from four <u>nodes</u>, each of which corresponds to one of the distinct units in our card reader. Each node contains

pictured by:

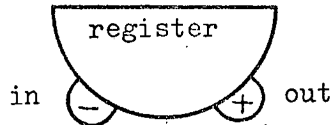


FIGURE 1.2

and modelled by a process reg which contains two communication capabilities; an input one and an output one.

We require that both capabilities have equal opportunity to communicate with the "outside world". Hence reg will be defined to be the set containing the input and output capabilities. We therefore consider the behaviour of a register as being non-deterministic since a register may behave in one of two ways which is not determined by itself but by its environment.

Supposing that the process reg is a member of the <u>domain</u> P (where a domain will be defined later but may be thought of as a set) then $P \cong \mathcal{P}(D)$, where $\mathcal{P}(D)$ is the powerdomain of D and denotes a certain subset of the powerset of D.

As mentioned above, the input and output capabilities of reg communicate with the environment on distinct lines. Thus the <u>capability domain</u> D may be defined by $D = I+O$ where I and O are domains and + is the disjoint sum. What are I and O, the domains containing the input and output capabilities respectively for reg?

Once a register has carried out an input communication we will require it to be identical to the original register except that it will have an amended memory, which now holds the input value. If an output communication has taken place, the register should be identical to that before communication. This suggests that

$$I = V \to P \quad \text{and} \quad O = V \times P$$

where V is some domain of values, e.g. N if we have an integer register.

But an arbitrary agent whose behaviour we wish to model may contain more than one input line and more than one output line, with the values input and output on these lines belonging to different value domains. Take for example the agent readerdevice pictured in Figure 1.1. If we assume that the sign of a port, either - or +, indicates that the port is to be used to input or output values respectively, then readerdevice has three input ports and two output ports, labelled as in Figure 1.1.

The sort of a process is defined to be the set of labels which name the ports of its corresponding node. This set is used to index the members of the process; that is we label each capability by some member of its sort. This label then indicates on which line the capability wishes to communicate.

The process which models readerdevice will then be in some domain

$$P_L = \mathcal{P}(D_L) \quad \text{where } L \text{ is its sort and}$$

where D_L consists of five summands, one for each input and output port of readerdevice.

To aid writing domain equations and to make proofs involving processes clearer (Chapter 4) we adopt the convention of writing each summand of domain P_M as $U_\alpha \times (V_\alpha \to P_M)$ where this summand corresponds to the port labelled by α. The summands which correspond to an input port α will have $U_\alpha = 1$, where $1 = \{o\}$ is a constant domain, and the summand which corresponds to the output port β will have $V_\beta = 1$.

The domain name in a domain equation can then be subscripted by a sort, and as each summand in this equation corresponds to one

member of the sort, we can define our process domain as

$$P_L = \mathcal{P}(\sum_{\lambda \in L} (U_\lambda \times (V_\lambda \rightarrow P_L)))$$

1.2 Definitions

A domain D will be an ω-algebraic consistently complete partial order (ccpo). For D to be a cpo then there is a partial order $\sqsubseteq$ over D such that (a) D has a mimimum element $\perp_D$ and (b) each directed set $X \subseteq D$ has a least upper bound (lub) $\sqcup X \in D$, and (b) each pair $x,y \in D$ having an upper bound in D has a least upper bound in D. For this ccpo D to be ω-algebraic we require that (c) the set of finite elements of D is countable and (d) every element of D is the lub of a directed set of finite elements of D. An element e is finite if for all directed $X \subseteq D$, $e \sqsubseteq \sqcup X \Rightarrow e \sqsubseteq x$ for some $x \in X$.

The Cartesian product of two domains is a domain while for any denumerable indexing set L and any family $\{D_\lambda \mid \lambda \in L\}$ of domains, the indexed sum $S = \sum_{\lambda \in L} D_\lambda$ of the family is also a domain, where $S = \{\langle\lambda, d\rangle \mid \lambda \in L,\ d \in D_\lambda\} \cup \{\perp_S\}$ in which $s \sqsubseteq s'$ iff $s = \perp_D$ or $s = \langle\lambda, d\rangle$, $s' = \langle\lambda, d'\rangle$ and $d \sqsubseteq d'$. $\langle\lambda, d\rangle \in S$ shall be written as $\lambda{:}d$ to aid legibility. Our indexing sets will be sorts, in the following sense:

If Σ is a denumerable alphabet of names, with $\bar{\Sigma}$ the denumerable alphabet of complementary names which are disjoint from Σ and in bijection with it such that $\alpha (\in \Sigma) \mapsto \bar{\alpha} (\in \bar{\Sigma})$; the set of labels is $\Lambda = \Sigma \cup \bar{\Sigma}$. Any finite subset of Λ is a sort.

Given this definition of label we let:

(i) α, β .... range over Σ,

(ii) $\bar{\alpha}, \bar{\beta}$ .... range over $\bar{\Sigma}$,

(iii) λ, μ.... range over Λ, while

(iv) α and $\bar{\alpha}$ will be complementary labels,

(v) $\bar{\lambda}$ is the complement of λ,

(vi) $\bar{\bar{\alpha}} = \alpha$,

(vii) the sign of label λ is sign $\lambda = +$ if $\lambda \in \Sigma$ and sign $\lambda = -$ if $\lambda \in \bar{\Sigma}$,

(viii) the name of a label is given by name α = name $\bar{\alpha} = \alpha$.

For any pair D, E of domains, the function domain $D \rightarrow E$ is the set of continuous functions from D to E under the ordering

$$f \sqsubseteq f' \Leftrightarrow \forall d \in D.\ fd \sqsubseteq f'd.$$

A function f from D to E is <u>continuous</u> if, for every directed subset X of D, $\{f(x) \mid x \in X\}$ is directed in E and equal to $f(\sqcup x)$.

The powerdomain $\mathcal{P}(D)$ for any domain D is also a domain, and is the set of certain subsets of the powerset of D under an ordering which will be defined in Chapter 3.

The domain P_L of processes of sort L (where $L \subseteq \Lambda$) is defined as the minimal solution of the isomorphism

$$P_L \cong \mathcal{P}\left(\sum_{\lambda \in L} (U_\lambda \times (V_\lambda \rightarrow P_L))\right)$$

such that the function proc: $P_L \rightarrow P_L$ which is defined as the least solution of

$$\text{proc}\ (p) = \phi(\text{proc})(p) = \{\!\!\{ \lambda : \langle u, \text{proc} \circ f \rangle \mid \lambda : \langle u, f \rangle \in p \}\!\!\}$$

is the identity function over P_L. The notation used here will be explained later but $\{\!\!\{\ \}\!\!\}$ can just be taken as the usual set forming brackets.

A pair of domains U_λ, V_λ are assigned to each label $\lambda \in L$. This assignment is such that $U_\lambda = V_{\bar{\lambda}}$ and $U_{\bar{\lambda}} = V_\lambda$.

That such solutions exist for isomorphisms not involving $\mathcal{P}$ is due to Scott [Sco 1], and for those containing $\mathcal{P}$ is due to both Plotkin [Plo 1] and Smyth [Smy].

1.3 Processes

Suppose p of Sort L is some member of domain P_L then $\lambda:\langle u,f\rangle$ will be a typical member of p, where $\lambda \in L$, $u \in U_\lambda$ and $f \in V_\lambda \rightarrow P_L$.

This capability can then indulge in an exchange of values with another capability belonging to some other process. $u \in U_\lambda$ is exchanged for some $v \in V_\lambda$. This is communication along the λ line.

Capabilities may be considered to be either input capabilities or output capabilities corresponding to the summands in which they belong. If $\lambda:\langle u,f\rangle$ is an input capability then $U_\lambda = 1$ (where $1 = \{o\}$, a "constant" domain) and $\lambda:\langle u, f\rangle$ will be written as $\lambda:\langle o,f\rangle$. If $\lambda:\langle u,f\rangle$ is an output capability then $V_\lambda = 1$ and $1 \rightarrow P_L \cong P_L$, hence we may write $\lambda:\langle u,f\rangle$ as $\lambda:\langle u,Kp'\rangle$.

$K = \lambda x.\lambda y.x$ is the constant combinator and $p' \in P_L$.

When both $U_\lambda = V_\lambda = 1$ then the communication which may take place along the λ communication line will be pure synchronisation, and no significant value exchange takes place. Capabilities labelled by such a λ will be known as synchronisers.

This notation for writing capabilities allows us to write processes. Considering the register example above we may define the process reg: $\{in, out\}$ by:

$$reg = \{\!\{ in : \langle o, \lambda z.\text{register } z\rangle \}\!\} \text{ where}$$

register : $Z \rightarrow P_{\{in,out\}}$ is defined recursively by

$$\text{register } z = \{\!\{ in : \langle o,\lambda z.\text{register } z\rangle,\ out : \langle z, K(\text{register } z)\rangle \}\!\}$$

This example will be mentioned in more detail in the following chapter.

1.4 Communication between processes

In the above we used the concept of a net to represent computing agents. The net in Figure 1.1 does not say how the components of a card reader communicate but only indicates where possible communication

may take place. This net then indicates the sort of the process which models the card reader.

Suppose $\alpha:\langle u,f\rangle\in p$ where $p\in P_L$, then $\alpha\in L$. We may write $p\in P_L$ as p:L and ambiguously represent the domain by the sort of those processes in it.

Since process p is a set, p may use this capability $\alpha:\langle u, f\rangle$ to communicate with some other process q, or it may use other capabilities to communicate with q, or it may not communicate with q. If this capability is used by p when it communicates with q then we can think of value $u\in U_\alpha$ being emitted by p and sent to q in exchange for some $v\in V_\alpha$ being received from q.

Process p will now transform itself into renewal fv, which is itself a process, and can continue to communicate.

Communication between two processes is defined as taking place between pairs of capabilities belonging to the two processes. The capability $\alpha:\langle u,f\rangle\in p$ will communicate with the capability $\bar{\alpha}:\langle v,g\rangle\in q$. The label α and its complement $\bar{\alpha}$ indicate the ability of members of processes p and q to communicate. If p:L and q:M then $\alpha\in(L\cap\bar{M})\cup(\bar{L}\cap M)$ if a communication on the α line is to take place, but communication will not actually take place unless both p and q contain capabilities as above. Communication is then the synchronised exchange of values between two processes.

If $L = \{\alpha, \bar{\gamma}\}$ and $M = \{\bar{\alpha}, \beta, \bar{\delta}\}$ then p and q may be pictured by the nodes

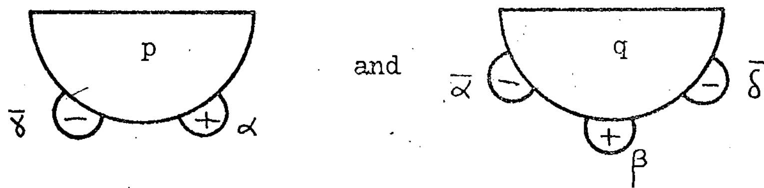


FIGURE 1.3

The process which is the outcome of p and q communicating will be pictured by

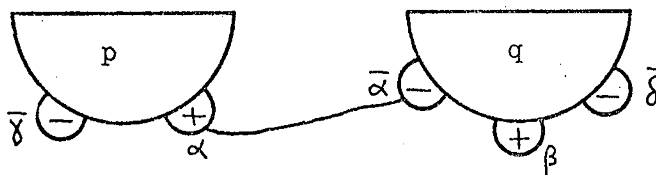


FIGURE 1.4

where two ports are joined by an arc when they have complementary labels. This arc is a communication line between p and q named by α .

Figure 1.4 pictures the process $p|q$ which is of sort $\{\alpha,\bar{\alpha},\bar{\gamma},\beta,\bar{\delta}\}$, and $|$ is our communication combinator between processes.

The combinator $|$ should let members of p and members of q communicate with each other in all possible ways such that a capability from p and a capability from q have complementary labels. Should p and q not contain any such members then no communication will take place. As we handle parallelism by non-determinism all capabilities of p and q should be interleaved in all possible ways.

This leads to the following definition for $|_{L,M} : P_L \times P_M \to P_{L \cup M}$; which is usually written as $| : L \times M \to L \cup M$:

$$\begin{aligned} p|q \quad = \quad & \{\!\!\{ \lambda:\langle u,\lambda v.(fv|q)\rangle \mid \lambda \in L,\ \lambda:\langle u,f\rangle \in p \}\!\!\} \\ & \cup \{\!\!\{ \lambda:\langle x,\lambda y.(p|gy)\rangle \mid \lambda \in M,\ \lambda:\langle x,g\rangle \in q \}\!\!\} \\ & \cup\, U\, \{\!\!\{ (fx|gu) \mid \lambda \in L \wedge \bar{\lambda} \in M, \\ & \qquad\qquad \lambda:\langle u,f\rangle \in p, \\ & \qquad\qquad \bar{\lambda}:\langle x,g\rangle \in q \}\!\!\} \end{aligned}$$

The first two clauses in this definition give $p|q$ the communication capabilities of both p and q, but with the renewal of $p|q$ (rather than the renewal of p or q), which is formed from the renewal of one of the processes and the other process itself using

| recursively. These two clauses of $p|q$ model concurrency between the computing agents modelled by processes p and q. They include those interleavings where p (or q) excludes all communications of q (or p). This combinator is therefore not "fair".

The third clause takes pairs of members of p and q in all possible ways such that the members have complemented labels. The capabilities produced by this clause also contribute to the set of capabilities of $p|q$. This clause models the communication between those agents modelled by processes p and q.

The combinator | will be shown to be both commutative and associative in Chapter 5.

1.5 Other Process operators

We may wish to restrict the capability of a process to communicate along a certain line. We therefore require an operator defined on a given name which removes from a process and its renewals capabilities labelled by this name and its complement. We have for any name α the unary operation:

$$\backslash\alpha : L \rightarrow L-\{\alpha,\bar{\alpha}\} \text{ defined by}$$

$$p\backslash\alpha = \{\!\{ \lambda:\langle u,\lambda v.(fv\backslash\alpha)\rangle \mid \lambda\notin\{\alpha,\bar{\alpha}\},\ \lambda:\langle u,f\rangle\in p \}\!\}$$

For our processes p and q of Figure 1.3, $p\backslash\alpha$ will be pictured by:

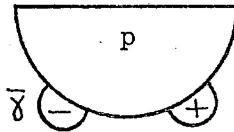


FIGURE 1.5

; while $(p|q)\backslash\alpha$ will be pictured by :

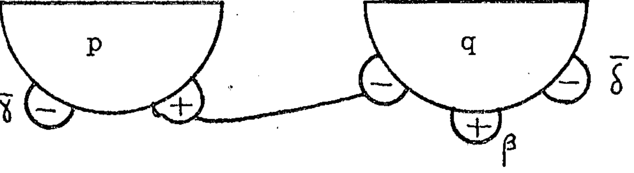


FIGURE 1.6

Thus a label on a port indicates that some other port (from another node) may be linked to it. When labels α and $\bar{\alpha}$ are omitted from ports as in Figure 1.6 then only p and q will be able to communicate with each other on the α line. After these communications have taken place, no other process will be able to communicate with (p|q) along this line.

This operation of restricting communication on a given line to certain pairs of processes and no others, that is dedicating a given communication line to a pair of processes, is a concept which is frequently met in computing systems. A derived operator $\| : L \times M \rightarrow L \cup M - (L.M)$ which does just this may be defined by

$$p \| q \quad = \quad (p|q)\backslash\alpha_1\backslash\alpha_2 \ldots \backslash\alpha_n \quad \text{where}$$
$$L.M \quad = \quad (L \cap \bar{M}) \cup (\bar{L} \cap M) \quad = \{\alpha_1, \bar{\alpha}_1, \ldots, \alpha_n, \bar{\alpha}_n\}.$$

Alternatively, $\|$ may be defined directly by

$$p \| q \;=\; \{\!\!\{ \lambda : \langle u, \lambda v.(fv \| q)\rangle \;|\; \lambda \in L - L.M, \lambda : \langle u, f\rangle \in p \}\!\!\}$$
$$\cup \{\!\!\{ \lambda : \langle x, \lambda y.(p \| gy)\rangle \;|\; \lambda \in M - L.M, \lambda : \langle x, g\rangle \in q \}\!\!\}$$
$$\cup \, U \{\!\!\{ (fx \| gu) \;|\; \lambda, \bar{\lambda} \in L.M, \lambda : \langle u, f\rangle \in p, \bar{\lambda} : \langle x, g\rangle \in q \}\!\!\}$$

Theorem 1.5.1

These two definitions of $\|$ are equivalent. This theorem is proved in the appendix, together with the proofs of the following two theorems.

Theorem 1.5.2 (Commutivity)

$$p \parallel q = q \parallel p$$

Theorem 1.5.3 (Associativity)

for p:L, q:M, r:N where $(L \cap M \cap \overline{N}) \cup (L \cap \overline{M} \cap N) \cup (\overline{L} \cap M \cap N) = \emptyset$

$$p \parallel (q \parallel r) = (p \parallel q) \parallel r$$

The restriction on the sorts of processes p, q and r in theorem 1.5.3 is seen to be necessary if we consider the nets of a counter example.

Suppose that $L = \{\alpha, \beta\}$, $M = \{\alpha, \overline{\gamma}\}$ and $N = \{\overline{\alpha}, \gamma, \overline{\delta}\}$; then $L \cap M \cap \overline{N} = \{\alpha\}$. The net corresponding to $p \parallel (q \parallel r)$ will look like:

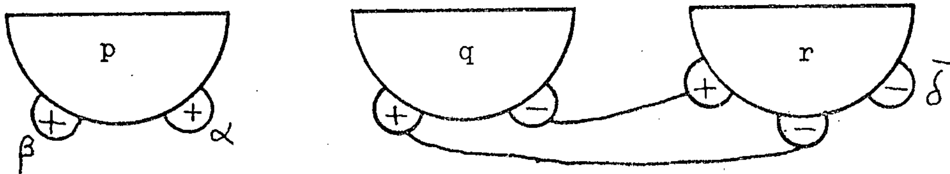


FIGURE 1.7

while the net corresponding to $(p \parallel q) \parallel r$ will look like:

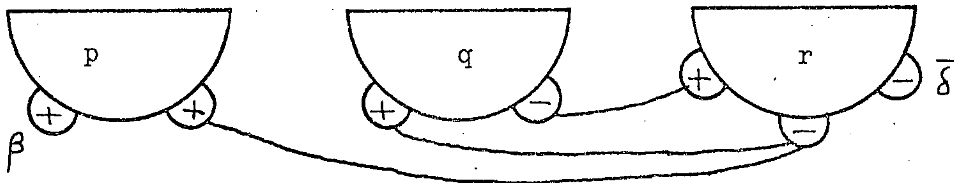


FIGURE 1.8

These two nets are distinct, thus $p \parallel (q \parallel r) \neq (p \parallel q) \parallel r$ for the sorts given above.

This form of communication appears to be appropriate when modelling some computing systems. $p \parallel q$ may equal $\emptyset$ and this in some sense corresponds to system deadlock and will be mentioned in Chapter 6.

CHAPTER 2

PROCESSES AS MODELS OF COMPUTING AGENTS

Processes, together with the operations $|$, $\|$ and $\backslash\alpha$, model the concurrency and communication met in systems of computing agents. These agents may be hardware agents, software agents or some combination of both, but no distinction is made when producing processes to model these agents. This is because we model the behaviour of agents, and the behaviour of any agent should be independent of its implementation.

Since processes can model hardware agents, we can use processes to specify the desired behaviour of hardware. Some actual hardware may also be described by a process and if these processes are equal then we have that this hardware is consistent with the specification. Generally, agents will be considered equivalent if they have identical processes modelling them.

In this chapter we model certain well understood computing concepts, such as memory, as well as some hardware, a card reader.

2.1 Registers and memories

We have defined a register process reg in the previous chapter. Let us redefine this slightly by changing its sort, and letting its name be indexed by an integer constant.

$\text{reg}_i \;:\; \{\alpha_i, \gamma_i\}$ is defined by

$\text{reg}_i \;= \{\!\{\alpha_i : \langle o, \lambda z.\ \text{register}_i z \rangle\}\!\}$

; where $\text{register}_i : \mathbb{Z} \to P_{\{\alpha_i, \gamma_i\}}$ is defined by

$\text{register}_i z = \{\!\{\alpha_i : \langle o, \lambda y.\text{register}_i y\rangle, \gamma_i : \langle z, K(\text{register}_i z)\rangle\}\!\}$.

$U_{\alpha_i} = V_{\gamma_i} = 1$ and $U_{\gamma_i} = V_{\alpha_i} = \mathbb{Z}$. If $\mathbb{Z}$ equals the flat cpo of

integers then we have an integer register, similarly for real values.

The process reg_i is defined in terms of the function $register_i$ which is parameterised on a local memory of the same type as the register itself, and which is used to hold its "contents".

We may assume that the first action a register may perform is to input a value to initialise itself. This is modelled by reg_i which only has one input capability.

Once a register contains some value we assume that it can either output this value or input another. This is modelled by the process $register_i z$, for the value z being held in our local memory.

The process reg_i is indexed by an integer constant i to enable the labels of its capabilities to be similarly indexed. The reason for this is that we wish to consider memories as consisting of a number of registers. We will then model the memory by modelling the registers and then combining them by using $|$. Each register process will be required to have a distinct input and output line.

Given register processes reg_i, $1 \leq i \leq n$, we can construct a memory process of n registers using the $|$ combinator. As the sorts L and M of any pair of these processes are such that $L.M = \emptyset$, then by the definitions of $|$ and $\|$ we have that $(reg_i | reg_j) = (reg_i \| reg_j)$ for $1 \leq i,j \leq n$.

Hence our memory process m_n may be defined by:

$$m_n \;:\; M_n = \blacksquare\{reg_i \mid 1 \leq i \leq n\} \;=\; \|\{reg_i \mid 1 \leq i \leq n\}$$

where $M_n = \bigcup\{R_i \mid 1 \leq i \leq n\}$ and $R_i = \{\alpha_i, \gamma_i\}$

and $\blacksquare$ and $\|$ are defined as follows:

Definition

$$\|\{p_i \mid a \leq i \leq b\} = p_a \mid p_{a+1} \mid \dots \mid p_{b-1} \mid p_b$$

for integers a, b.

Definition

$$\|\{p_i \mid a \leq i \leq b\} = p_a \| p_{a+1} \| \dots \| p_{b-1} \| p_b$$

for integers a, b.

If p:L is a process which wishes to access some or all of the registers which constitute m_n then $L \cap \overline{M}_n \neq \emptyset$.

If we wish this memory to be local to p and inaccessible to any other process then we construct

$$(p \mid m_n) \setminus M_n,$$

where $\setminus M_n$ abbreviates $\setminus\alpha_1 \setminus\gamma_1 \dots \setminus\alpha_n \setminus\gamma_n$.

If $L = L' \cup M_n$, then

$$(p \mid m_n) \setminus M_n = p \parallel m_n.$$

By use of this abbreviated removal combinator $\setminus M_n$ we can explicitly control whether memory is to be local to a process (or processes), or global to any number of processes. If M_n is to be global then we just construct

$$p \mid m_n.$$

We may also treat part of m_n as local memory and part as global memory.

If $N_K = \{\alpha_i, \gamma_i \mid i \in K\}$ then

$(p \mid m_n) \setminus N_{local}$ is such that

$m_n = m_{local} \mid m_{global}$ where

$m_K = \|\{reg_i \mid i \in K\}$ and $local \subseteq \{1, 2 \dots n\}$ and

$global = \{1, 2 \dots n\} - local$.

Memories holding values other than integers can be constructed analogously, and so also for composite memories.

We can see from this example that we have a large amount of flexibility when modelling memory due to the properties of our process model. We can also see that we do not model agents in terms of a global memory but should we wish to model a memory it is treated as any other agent and modelled by a process.

2.2 Other data structures - queues and stacks

As with a register, a non-empty queue can be considered to have non-deterministic behaviour; that of sending and receiving values. The processes which model queues and stacks will utilise auxiliary memory (such as the process $\text{register}_1 z$ does with a single element z).

An integer queue process uses Z^*, a set of finite sequences of integers, as its auxiliary store for which we assume that the following functions are strict:

$\frown$: $Z^* \times Z \to Z^*$; addition of a new element to the right-hand end of a sequence

$\frown$: $Z \times Z^* \to Z^*$; as above, but adds to the left-hand end

where Z is the flat cpo of integers.

An initially empty queue process (which may become arbitrarly large) which receives elements on its α line and sends elements on its $\bar{\beta}$ line is defined as follows:

$$q_{\bar{\beta}\alpha}(\varepsilon) : \{\alpha, \bar{\beta}\} \quad , \quad \text{where } q_{\bar{\beta}\alpha} : Z^* \to P_{\{\alpha, \bar{\beta}\}}$$

is defined by

$$q_{\bar{\beta}\alpha}(\varepsilon) \quad = [\![\alpha : \langle o, \lambda z. q_{\bar{\beta}\alpha}(\varepsilon \frown z) \rangle]\!]$$

$$q_{\bar{\beta}\alpha}(z_1 \frown s_1) = [\![\alpha : \langle o, \lambda z. q_{\bar{\beta}\alpha}(z_1 \frown s_1 \frown z) \rangle , \bar{\beta} : \langle z_1, K(q_{\bar{\beta}\alpha}(s_1)) \rangle]\!]$$

where $\varepsilon \in Z^*$ is the empty sequence of Z values such that

$$\varepsilon \curvearrowleft z = z = z \curvearrowright \varepsilon .$$

The subscripts on the function name $q_{\bar{\beta}\alpha}$ make the sort of the resulting process explicit; namely $\{\alpha, \bar{\beta}\}$. One of the labels is barred and the other is unbarred since they name output and input capabilities respectively.

An integer stack process is similar but it does not use the function $\curvearrowright$.

$$\text{stack}_{\bar{\beta}\alpha}\ (\varepsilon) : \{\alpha, \bar{\beta}\} \text{ where } \text{stack}_{\bar{\beta}\alpha} : Z^* \to P_{\{\alpha, \bar{\beta}\}}$$

is defined by

$$\text{stack}_{\bar{\beta}\alpha}\ (\varepsilon) = \{\!\{ \alpha : \langle o, \lambda z.\ \text{stack}_{\bar{\beta}\alpha}(\varepsilon \curvearrowleft z) \rangle \}\!\}$$

$$\text{stack}_{\bar{\beta}\alpha}\ (s_1 \curvearrowleft z_1) = \{\!\{ \alpha : \langle o, \lambda z.\ \text{stack}_{\bar{\beta}\alpha}((s_1 \curvearrowleft z_1) \curvearrowleft z) \rangle, \bar{\beta} : \langle z_1, K(\text{stack}_{\bar{\beta}\alpha} \cdots \cdots (s_1)) \rangle \}\!\}$$

Data structures will in general be modelled by processes which send and receive values via an auxiliary store. This store is only an abstract concept which is used to model the behaviour of a real data structure correctly. The type of this store and the manner in which values are removed from it and update it determine the "type" of data structure which we are modelling.

2.3 Semaphores

As an example of a process which consists only of synchronisers and is used only to synchronise some communication among other processes, consider Dijstra's semaphores.

A semaphore may be modelled by the process sem(z) : $\bar{S}$, for some $z > 0$ initially, where $S = \{p_1, p_2, v\}$ and where $\text{sem} : Z \to P_{\bar{S}}$ is defined by

$$\text{sem}(z) = \{\!\{ \bar{p}_1 : \langle o, K(\text{sem}'\ (z-1)) \rangle,\ \bar{v} : \langle o, K(\text{sem}(z+1)) \rangle \}\!\} \text{ if } z > o$$

$$\text{sem}(o) = \{\!\{ \bar{p}_1 : \langle o, K(\text{sem}(z-1)) \rangle, \bar{v} : \langle o, K(\text{sem}(z+1)) \rangle \}\!\}$$

$$\text{sem}(z) = \{\!\{ \bar{p}_1 : \langle o, K(\text{sem}(z-1)) \ , \bar{v} : \langle o, K(\text{sem}'\ (z+1)) \rangle \}\!\} \text{ if } z < o$$

where $sem'(z) = \{\bar{p}_2 : \langle o, K(sem(z)) \rangle\}$.

This definition by cases could also have been expressed as a single equation using a conditional function and the union operation.

In the above

$$U_{\bar{p}_1} = V_{\bar{p}_1} = U_{\bar{p}_2} = V_{\bar{p}_2} = U_{\bar{v}} = V_{\bar{v}} = 1.$$

This definition reflects Dijksta's definition of semaphores [Dij 1] in which an agent Q that performs a P operation (on the semaphore) decreases the value of the semaphore by 1. If the resulting value is non-negative then Q can complete a P operation which it performs in two parts (and is modelled by two communications on the p_1 and p_2 lines). If the resulting value is negative then Q may only perform the first part (modelled by a p_1 communication) of the P operation and may then be thought of as waiting.

An agent Q' that performs a V operation on a semaphore increases its value by 1. If the resulting value is positive then the operation has no further effect, if the resulting value is non-positive any one of the waiting processes can then be allowed to continue.

A semaphore such as this is used to control a number of agents, each of which can communicate with it. These agents may be modelled by the processes $q_1, \ldots, q_n$ each of which contain $\{p_1, p_2, v\}$ in their sort.

Any process which communicates with sem(z) and is awaiting a communication on its line labelled by p_2 will be waiting for "enough" other processes to communicate with the semaphore on their v lines.

As a semaphore may be used to control how a number of agents access a resource we may model this resource by process r. The positive integer initially assigned to process sem(z), namely z, indicates the maximum number of processes from $q_1, \ldots, q_n$ which may access resource r in parallel. If we wish this maximum access number to be m, where $m \leq n$ then we build

$((q \mid sem(m)) \setminus S) \mid r$ where

$q = q_1 \mid \cdots \mid q_n$

We assume that each process q_i performs its p_1, p_2 and v communications in a disciplined manner, namely a p_1 then a p_2 then a v then a p_1 etc. sequence of communications among the others it can do.

A semaphore is used to synchronise a number of agents; the process sem(z) is used to synchronise the communications of a number of processes. A more general technique for scheduling the communications of a number of processes, in which a semaphore is a special case, is given in Chapter 7.

The process model does not solve the fair scheduling problem; one process q_i may regain access to the semaphore infinitely often whilst another q_j is not allowed to progress at all. But, by the definition of sem(z), once q_j is in the "wait" state after a p_1 communication, any v communication performed by q_i may not be followed by a p_1 from q until q_j or some other waiting process has completed its P operation (by a p_2 communication).

2.4 A hardware example

Rather than modelling the expected behaviour of well understood computing agents, such as our previous examples, we shall model here the actual behaviour of some existing computing agent. Some complete, though possibly informal, description of the behaviour of this agent is necessary and it is from this that our process is produced.

The example we use to illustrate the suitability of processes for modelling "real" agents involving hardware is that of the card reader mentioned in Chapter 1. This card reader is constructed by "joining up" four component agents; three of which are hardware while

the fourth is a software agent.

The card reader considered is the DIGITAL CR 11 described in [Dig], which is compatible with the PDP 11 system philosophy of peripheral handling via interrogation of a status register. The four component units of this card reader are a buffer register, a status register, the reading mechanism itself and a driver program. This last component controls or drives the hardware (the other three components) and resides in the memory of, and is executed by, a PDP 11 processor connected to the card reader.

A description of each of these component agents is given below together with the processes which model them. The agent and its corresponding process will be given the same name. The four processes communicate according to the net given in figure 1.1, which is reproduced here

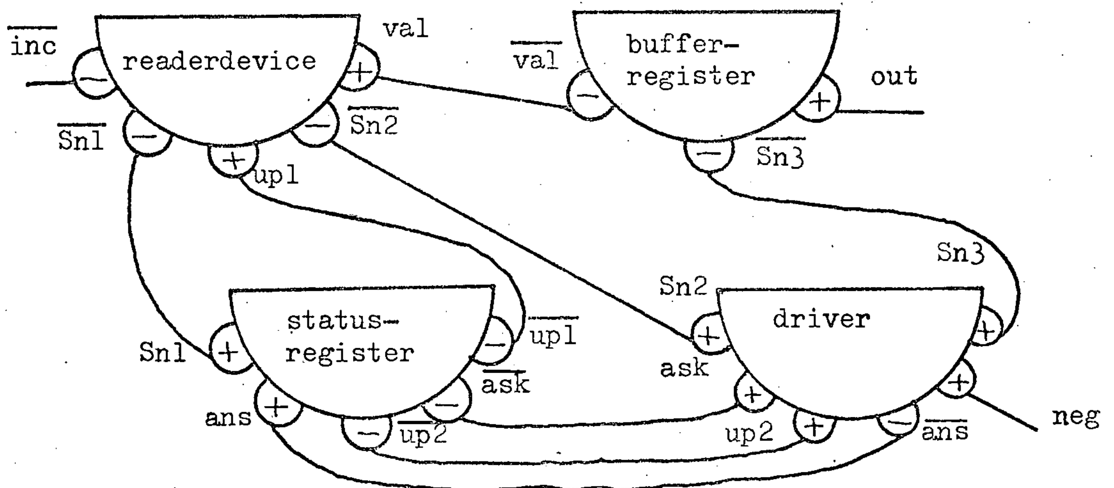


FIGURE 2.1

Domain pairs are assigned to the labels used by our processes as indicated by table 2.2 This specifies what values flow along the communication lines of the net in figure 2.1.

TABLE 2.2

LABEL (λ)	DOMAINS (U_λ, V_λ)
Sn1	1, 1
ans	T, 1
$\overline{\text{inc}}$	1, CARD
up1	N×T, 1
val	N, 1
out	N, 1
Sn2	1, 1
ask	N, 1
up2	N×T, 1
neg	M, 1
Sn3	1, 1

where N is the flat cpo of positive integers $N = \{\perp_N, 0, 1, 2 \ldots\}$; T is the flat cpo of booleans $T = \{\perp_T, 0, 1\}$; M is the flat cpo $M = \{\perp_M, -1\}$; and CARD = COLUMN^{80}, where COLUMN is some binary encoding of card columns and will not concern us.

(a) The status register

The status register consists of 16 bits, each of which may represent some condition of the card reader hardware. This register is updated by the reader device and driver agents and interrogated by the driver.

The card reader design description in terms of an interrogated status, though intuitively simple, may in fact be implemented by the device sending an appropriate interrupt to the driver (which resides

in the PDP 11 processor), rather than the device changing a status bit which may then be examined by the driver. We will not concern ourselves with whether a status register is just an abstract concept or not, but shall assume that it actually exists. Once again we use the fact that processes only model behaviour and not the intensional details of how this behaviour is arrived at.

Only a certain number of the bits belonging to the status register (which are numbered from zero to fifteen) are used by the driver and reader device. Which bits these are and what status they indicate appears in the following table.

Bit	Status indicated when set
15	error
14	a card has passed through the read station and another may be demanded
9	card being read
8	reader device off-line; if not set then device on-line and read commands may be accepted
6	if set when status register loaded allows the setting of bits 14 and 15 to cause a driver interrupt
0	if set when status register loaded, causes the driver to signal the reader device to deliver a card to the read station, and commence reading

TABLE 2.3

The process statusregister : P_S, where $S = \{ Sn1, ans, \overline{up2}, \overline{ask}, \overline{up1} \}$, is parameterised on an auxiliary store of 16 boolean values. The Sn1 line is used by this process to signal the driver process that reading is to commence, the ask and ans lines are used by the driver to interrogate the auxiliary store while lines up1 and up2 are used by the driver and reader device processes respectively to update the contents of the auxiliary store belonging to the status register process.

statusregister : P_S is defined by

statusregister = status(d), where status : $T^{16} \to P_S$ is defined by

$$status(b) = \{\!\!\{ \overline{ask}:\langle o, \lambda n. \{\!\!\{ ans:\langle b(n+1), K(status(b))\rangle \}\!\!\}\rangle,$$
$$\overline{up2}:\langle o, \lambda m\epsilon(N\times T).(m(1)=o \wedge m(2)=1) \to$$
$$\{\!\!\{ Sn1:\langle o, K(status(\langle 1, b(2),\ldots, b(8), o,\ldots o\rangle))\rangle \}\!\!\},$$
$$status(\langle o,\ldots, b(m(1)), m(2), b(m(1)+2), o,\ldots, o\rangle)\rangle,$$
$$\overline{up1}:\langle o, \lambda m\epsilon(N\times T).status(\langle o,\ldots, b(m(1)), m(2), b(m(1)+2),\ldots$$
$$\ldots b(16)\rangle)\rangle \}\!\!\}$$

and $d = \langle o,1,o,o,o,o,1,o,o,o,o,o,o,o,o,o \rangle$

(b) The buffer register

The buffer register is a 16 bit register which holds a positive integer encoding of a data card column. Whenever the reading mechanism reads a card column, this column will be converted into an integer by the reading device and placed in the buffer register.

The process bufferregister : $P_{\{\overline{val}, \overline{Sn3}, out\}}$ sequences the passing of an integer from the reader device process to the "outside world" with respect to a synchronising communication from the driver process on line Sn3.

As the behaviour of the buffer register acts sequentially no auxiliary store is needed by the process buffer-register to hold the value being buffered.

$$bufferregister = \{\!\!\{ \overline{val}:\langle o, \lambda n. \{\!\!\{ \overline{Sn3}:\langle o, K \{\!\!\{ out:\langle n, K(bufferregister)\rangle \}\!\!\}\rangle \}\!\!\}\rangle \}\!\!\}$$

(c) The reader device

The reading mechanism is described both by Wirth [Wir 2] and in the PDP 11 peripheral handbook [Dig]. The action it performs may be described as follows.

When the driver sets bit o of the status register to zero it also signals the reader device to commence reading. The device sets bit 9 and promptly inputs a card and starts to read it. Once a card is in motion the action of reading a column, converting it to an integer and placing this in a buffer register takes place column by column. When the last column has been read bit 14 is set. This bit will be checked by the driver and if found to be set, the driver requests the device to read another card.

The process readerdevice:P_R, where $R = \{\overline{inc}, \overline{Sn1}, upl, \overline{Sn2}, val\}$, which models this action is defined by

$$readerdevice = \{\!\{\overline{Sn1}:\langle o, K\{\!\{upl:\langle\langle 9,1\rangle, K\{\!\{\overline{inc}:\langle o, \lambda c \in CARD.\ countsend(1)\rangle\}\!\}\rangle\}\!\}\rangle\}\!\}$$

where countsend : $N \rightarrow P_R$ is defined by

$$countsend(j) = \{\!\{\overline{Sn2}:\langle o, K((j=81) \rightarrow \{\!\{upl:\langle\langle 14,1\rangle, K(readerdevice)\rangle\}\!\}, \{\!\{val:\langle change(c(j)), K(countsend(j+1))\rangle\}\!\})\rangle\}\!\}$$

The function change:COLUMN $\rightarrow$ N encodes a card column as an integer and is not defined.

(d) The driver

The card reader driver we model is that given by Wirth in [Wir 2], his paper on the use of the language MODULA. We do not attempt to give a process semantics for this driver program but use it as an algorithm from which to produce the corresponding process. Later in this thesis we show how processes can be used to give the denotational semantics of another multiprocessing language.

The driver is implemented in MODULA by the following program fragment where we have added statement numbers to aid its explanation.

```
1    process driver [230B];
2     const m = 81; (*block size*)
3      var crs [177160B]: bits;     (*status*)
4       crb [177164B]: integer;     (*buffer*)
5      procedure put (x: integer);
6       begin buf [inx] : =; inx : = (inx mod n) + 1;
7         inc (nf)
8    end put;
9     begin
10     loop dec (ne,m);
11      if ne < o  then wait (nonfull) end;
12     while not off (crs, [8,9]) do wait (crsig) end;
13      crs : = [0,6];   (*start card motion*)
14        loop doio;
15   when not off (crs, [14,15]) exit
16    put(crb)
17   end;
18     put(-1); crs[6] : = false; (*end of line mark*)
19    if nf  >=  0 then  send (nonempty) end
20   end
21    end driver;
```

Once activated the device reads a full 80 characters without halting. Therefore each value must be removed from the buffer register before the next arrives. Wirth uses a cyclic buffer to hold the values output by the buffer register and he prevents a card from being read until 81 locations (80 card locations plus a delimiter) are available in this buffer. Statements 2, 10 and 11 implement this cyclic buffer. We do not model this as it is not a card reader feature and we presume that there is always an available

destination for the values from the buffer register.

The status of the buffer register is interrogated before and after each card is read. If bit 8 is set (not ready) then this interrogation must be repeated until bits 8 and 9 are found to be unset. Statement 12 implements this.

We assume that our statusregister process is in the correct state before data transfer takes place. Hence we do not model statements 3 and 4 which initialise the status and buffer registers.

Statement 13 sets the interrupt enable bit (bit 6) and initiates reading by setting bit 0 to zero. Statement 14 initiates a loop where repeated input of a card column to the buffer register takes place via the procedure doio.

Statement 15 tests bits 14 and 15, while statement 16 causes the value in the buffer register to be output.

Statement 18 causes a delimiting value to be output after the end of a card, and bit 6 to be unset. This unsetting disables bit 14 and prevents another delimiter from being output until the next card has been read. Statement 19 concerns the cyclic buffer and is ignored.

We define a process driver:P_T where $T = \{Sn2, ask, up2, \overline{ans}, neg, Sn3\}$, which models the action described above. The ask and ans lines are used by this process to interrogate the statusregister process, while up2 is used to change its contents. Line Sn2 is used to start the readerdevice process reading, line Sn3 synchronises the output of the bufferregister and line neg outputs the delimiter value.

$$\begin{aligned} driver = &\llbracket ask{:}\langle 9, K\llbracket \overline{ans}{:}\langle o, \lambda t.(t=1) \rightarrow driver, \\ &\qquad \llbracket ask{:}\langle 8, K\llbracket \overline{ans}{:}\langle o, \lambda t.(t=1) \rightarrow driver, \\ &\qquad \llbracket up2{:}\langle\langle 6,1\rangle, K\llbracket up2{:}\langle\langle o,1\rangle, K(doio)\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket \end{aligned}$$

where doio:P_T is defined by

$$
\begin{aligned}
doio = \llbracket\, & Sn2:\langle o, K\llbracket ask:\langle 15, K\llbracket \overline{ans}:\langle o, \lambda t.(t=1) \rightarrow endcard, \\
& \llbracket ask:\langle 14, K\llbracket \overline{ans}:\langle o, \lambda t.(t=1) \rightarrow endcard, \\
& \llbracket Sn3:\langle o, K(doio)\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket\rangle\rrbracket
\end{aligned}
$$

$$
endcard = \llbracket neg:\langle -1, K\llbracket up2:\langle\langle 6, o\rangle, K(driver)\rangle\rrbracket\rangle\rrbracket .
$$

As we have modelled each component of the card reader by a process, we can then model the card reader itself by the process:

$$
cardreader \;=\; readerdevice \,\|\, bufferregister \,\|\, statusregister \,\|\, driver
$$

The $\|$ combinator is used as we wish the internal communication lines to be unavailable for any other process to communicate on. In Chapter 4 we prove that this process is equal to another process which models the expected (or desired) behaviour of a card reader. This proof then demonstrates the "correctness" of the hardware card reader, along with its driver component.

CHAPTER 3

THE DOMAIN

Here we define the powerdomain construction used in our model, which is an amended form of the powerdomain construction of Smyth [Smy]. This alteration enables us to include the empty set in $\mathcal{P}(D)$ which is needed in our treatment of deadlock.

Various reasons for preferring Smyth's construction to that of Plotkin [Plo 1] (which was the first powerdomain construction) are given.

Our set-forming notation $\{\!\{ - | - \}\!\}$ is defined in terms of sets, while various extensions of this basic set-forming notation which prove useful when defining process operations, are mentioned.

3.1 The weak powerdomain

The powerdomain $\mathcal{P}(D)$ used in our model is known as "weak" to distinguish it from the "strong" powerdomain of Plotkin. These powerdomains differ in the orderings used in them.

The powerdomain $\mathcal{P}(D)$ of any domain D contains only the finitely generable subsets of D. Just as the restriction to continuous functions enabled Scott [Sco 1] to show the existence of solutions to isomorphic domain equations involving $\rightarrow$, so the restriction to finitely generable subsets of the power set allows both Plotkin and Smyth to show the existence of solutions to domain equations involving $\mathcal{P}$. Finite generability is defined as follows, and this differs from that of Smyth in that it allows finite paths in the generation tree T.

<u>Definition</u> A set $X \subseteq D$ is <u>finitely generable</u> if (a) there exists an infinite finitely branching tree $\top$, which may contain finite as well as infinite paths whose nodes are labelled with elements of D in such a way that the labels of each path form a $\sqsubseteq$ chain in D, and (b) X is the set of lubs of chains which label the infinite paths of $\top$.

The powerdomain $\mathcal{P}(D)$ is constructed by:

(i) forming the set $\mathcal{F}(D)$ of finitely generable subsets of D;

(ii) defining a pre-order $\lesssim_0$ over $\mathcal{F}(D)$ by $X \lesssim_0 X'$ iff $\forall x' \in X'.$ $\exists x \in X. x \sqsubseteq x'$;

(iii) defining the equivalence $\simeq_0$ over $\mathcal{F}(D)$ by $X \simeq_0 X'$ iff $X \lesssim_0 X'$ and $X' \lesssim_0 X$.

We then have a domain

$$(\mathcal{F}_0(D), \sqsubseteq) \quad = \quad (\mathcal{F}(D)/\simeq_0, \lesssim/\simeq_0)$$

where $\mathcal{F}_0(D)$ is the quotient of $\mathcal{F}(D)$ under $\simeq_0$, and this is a domain of equivalence classes. We take a certain member of each equivalence class in $\mathcal{F}_0(D)$; the right-closed member.

<u>Definition</u> For $X \in \mathcal{F}_0(D)$, its <u>right-closure</u> $\mathcal{R}C(X)$ is defined by $\mathcal{R}C(X) = \{y \in D \mid \exists x \in X. x \sqsubseteq y\}$.

$\mathcal{F}_0(D)$ is then taken to be the set of right-closed finitely generable sets under the ordering $X \sqsubseteq X'$ iff $X \lesssim_0 X'$, for right-closed X and X'. $\mathcal{P}(D)$ will then be $\mathcal{F}_0(D)$.

The strong powerdomain $\mathcal{F}_M(D)$ of Plotkin is constructed with a "stronger" ordering $\lesssim_M$, the Milner ordering, which identifies less sets under the equivalence $\simeq_M$ than $\lesssim_0$ does under $\simeq_0$.

<u>Definition</u> The Milner ordering if defined by

$X \sqsubseteq_M X'$ iff $(\forall x \in X . \exists x' \in X' . x \sqsubseteq x')$ and $(\forall x' \in X' . \exists x \in X . x \sqsubseteq x')$

For $a, b \in D$ where $a \sqsubseteq b$;

$\{a\} \simeq_O \{a,b\}$ while $\{a\} \not\simeq_M \{a,b\}$

This "greater" identification is one of the disadvantages of using the weak powerdomain rather than the strong powerdomain.

One immediate property of the weak powerdomain (due to right-closure) is that any set in $\mathcal{P}(D)$ which contains $\perp_D$ is equivalent to the set $\{\perp_D\}$. As $\perp$ is usually used to denote non-termination, we do not therefore distinguish between a program which sometimes fails to terminate and one which never terminates. This identification may appear severe but it is also the position adopted by Dijkstra who believes that a program which sometimes fails to terminate is as unsuitable as one which never terminates.

3.2 Choice of domain

There are certain advantages to be had by using $\mathcal{F}_O(D)$ as our powerdomain rather than $\mathcal{F}_M(D)$.

Firstly, the empty set $\emptyset$ can easily be included in $\mathcal{F}_O(D)$ where it is "top", the most defined element in our domain. As processes are right-closed sets in $\mathcal{F}_O(D)$, then the more defined they become the smaller they become; that is they then contain less members. The most defined element is then intuitively the empty set, which may be thought of as modelling deadlock. This interpretation is mentioned in Chapter 6.

It appears that $\emptyset$ may be included in $\mathcal{F}_M(D)$ but in a less obvious manner than in $\mathcal{F}_O(D)$. $\emptyset_{\mathcal{F}_M(D)}$ may be placed to the "side"

of $\perp_{\mathcal{F}_M(D)}$ and a new $\perp$ is added which is dominated both by $\phi_{\mathcal{F}_M(D)}$ and $\perp_{\mathcal{F}_M(D)}$. This has not as yet been investigated.

As a second advantage of using $\mathcal{F}_o(D)$; $\sqsubseteq$ and the set-theoretic superset ordering $\supseteq$ are identical. This follows directly from their definitions as the sets being ordered are right-closed. This result is not only useful when carrying out proofs about processes but appears to be necessary in certain cases. In the proof of the associativity of the $|$ combinator (Chapter 5) we use that $p \sqsubseteq p' \Rightarrow p|q \sqsubseteq p'|q$; and this follows from the equivalence of $\sqsubseteq$ with $\supseteq$ and the monotonicity of $|$. As $\sqsubseteq_M$ is not equivalent to $\supseteq$ we are unable to prove that $|$ is associative in $\mathcal{F}_M(D)$. In fact $|$ is <u>not</u> associative in $\mathcal{F}_M(D)$ but there is a combinator which is. The associativity of $|$ appears to be a necessary feature of our model.

For these two reasons we choose $\mathcal{F}_o(D)$ as our powerdomain. As an added advantage we should note that when proving $p \sqsubseteq q$, where p and q are processes, it is required to show that (a) $\forall y \in q . \exists x \in p . x \sqsubseteq y$; while to prove that $p \sqsubseteq_M q$ we need show that (a) holds as well as

(b) $\forall x \in p . \exists y \in q . x \sqsubseteq y$.

Hence such proofs in $\mathcal{F}_M(D)$ are often twice as long as in $\mathcal{F}_o(D)$.

3.3 Continuity of the set-forming operation

We list certain properties of $\mathcal{P}(D)$, where X and Y are members of $\mathcal{P}(D)$.

(1) $X \sqsubseteq Y \Leftrightarrow X \supseteq Y$

(2) D is minimum and ϕ is maximum in $\mathcal{P}(D)$

(3) $\cup$ and $\cap$ are continuous where $X \cap Y = X \sqcup Y$ and $X \cup Y = X \sqcap Y$

(4) for S any directed set of sets, $S \in \mathcal{P}(D)$, then $\bigcup S = \bigcap S$

(5) function extension: if $f: D \to E$ is continuous then so also is $\hat{f}: \mathcal{P}(D) \to \mathcal{P}(E)$; where $\hat{f}(X) = \mathcal{R}C\{f(x) \mid x \in X\}$

(6) large union: $U: \mathcal{P}(\mathcal{P}(D)) \to \mathcal{P}(D)$ is continuous, where $\bigcup\{X, Y\} = X \cup Y$

(7) singleton set: $\{\!|-|\!\}: D \to \mathcal{P}(D)$ is continuous, where $\{\!|x|\!\} = \mathcal{R}C\{x\}$.

(8) restriction: $\setminus : \mathcal{P}(D) \times (D \to 2) \to \mathcal{P}(D)$ is continuous, where $2 = \{\bot, \top\}$ with $\bot \sqsubseteq \top$ and $x \setminus f = \mathcal{R}C\{x \in X \mid f(x) = \bot\}$.

The proof of (1) to (7) are due to Smyth [Smy] while that of (8) is due to Milner and appears in [Mil 1] .

The proofs of the following two corollaries, which define two basic set-forming constructs and state that they are continuous, also appear in [Mil 1] and are due to Milner.

Corollary 3.3.1

Let $X \in \mathcal{P}(D)$, $p \in D \to T$, $h \in D \to E$.

The set $\{\!|h(x) \mid p(x),\ x \in X|\!\} \in \mathcal{P}(E)$ then varies continuously in h, p and X when interpreted as $\mathcal{R}C\{h(x) \mid x \in X,\ p(x) \sqsubseteq \text{true}\}$,

$T = \{\bot_T, \text{true}, \text{false}\}$.

Corollary 3.3.2

Let $D = \sum_{\lambda \in L} D_\lambda$, $E = \sum_{\mu \in M} E_\mu$ where L and M are finite sets of labels. If $X \in \mathcal{P}(D)$ and $g_\nu \in D_\nu \to E_\nu$ for each $\nu \in N \subseteq L \cap M$ then the set

$$\{\!|\nu : g_\nu(u,f) \mid \nu \in N, \nu : \langle u,f \rangle \in X|\!\} \in \mathcal{P}(E)$$

varies continuously in each g_ν and in X, when interpreted as

$$\mathcal{R}C(\{\nu : g_\nu(u,f) \mid \nu \in N, \nu : \langle u,f \rangle \in X\} \cup \{\bot_E \mid \bot_D \in X\}) \ .$$

$\{\!|-|-|\!\}$ as used in these corollaries is frequently used in the definition of processes, often along with the functions $\cup$ and U. Our definition

of the combinator $|$ is such an example, and $|$ is therefore continuous by the above results. All process operations and functions will therefore be continuous.

$\{\!\{-|-\}\!\}$ may be treated as $\{-|-\}$ while remembering that it denotes a right-closure and is strict in the set parameter. Various extensions of the basic set-forming constructs given above follow together with extensions of set-theoretic $\{-|-\}$.

Our set-forming construct of Corollary 3.3.1 may be extended to $\{\!\{\phi(x_1,x_2)|p(x_1,x_2),x\epsilon X_1,x_2\epsilon X_2\}\!\}$ which is interpreted as $\mathcal{R}C\{\phi(x_1,x_2)|x_1\epsilon X_1,x_2\epsilon X_2,\ p(x_1,x_2)\equiv true\}$. The continuity of this follows in a similar manner to that of Corollary 3.3.1.

Set forming abbreviations whose interpretation is obvious are also used. For example

$$\{\!\{\alpha_i:\langle u,f\rangle|1\leqslant i\leqslant n\}\!\} \quad = \quad \{\!\{\alpha_1:\langle u,f\rangle,\ldots\alpha_n:\langle u,f\rangle\}\!\} \ .$$

The continuity of all process functions and operations allows us to use the computational induction technique of the next chapter on recursively defined processes.

CHAPTER 4

PROOFS IN THE PROCESS MODEL

One of the purposes of modelling concurrent computing agents by processes is to allow us to reason about these agents by carrying out proofs on their corresponding processes. Most of these processes will be defined recursively using the set-forming construct, often producing infinite sets. Hence the computational induction rule due to Scott (and described in Manna [Man]) and Park [Par] is used.

Using computational induction we can prove a number of theorems which will be useful in future proofs. As the production of proofs involves a large amount of intuition the techniques and theorems in this chapter are only a guide to carrying out process proofs. These techniques are quite general and instances of their use in examples are given.

4.1 Induction techniques

Computational induction is an induction method on the depth of recursion which is described in Manna [Man].

Consider the process p defined recursively by $p = \phi p$. Such processes have the least fixed point fix ϕ of the functional ϕ as their solution. If ϕ is continuous we also know that

$\text{fix}\,\phi = \bigsqcup_{i=0}^{\infty} \phi^{i}(\perp)$; the lub of the chain $\{\phi^{i}(\perp)\}$, where $\phi^{0}(x) = x$ and $\phi^{i+1}(x) = \phi(\phi^{i}(x))$.

That the functionals used in our process definition are continuous is due to the continuity results of the previous chapter together with the following theorem, due to Scott.

Theorem 4.1.1

Any functional ϕ defined by composition and λ abstraction of continuous functions and the function variable F, where $\phi = \lambda F.\phi'$, is continuous. Before giving the induction rule we require the following definition.

Definition A predicate P is admissible iff for a directed set X, and $\forall x \in X.Px$ holds, then $P(\bigsqcup_{i=0}^{\infty} X)$.

For $p = \phi p$ let $\phi^i(\perp) = p_i$. Our computation induction rule is as follows:

To prove that $P(\text{fix}\,\phi)$ holds for some admissible predicate P it is sufficient to prove that

(i) $P(\perp)$ holds, and

(ii) $\forall j.P(p_j) \Rightarrow P(p_{j+1})$ such that $j \geq 0$.

(i) and (ii) give us that $\forall i.P(p_i)$ holds and as the p_i form a directed set, and as P is admissible then $P(\bigsqcup_{i=0}^{\infty} p_i)$ holds. As ϕ is continuous $\text{fix}\,\phi = \bigsqcup_{i=0}^{\infty} \phi^i(\perp) = \bigsqcup_{i=0}^{\infty} p_i$, hence $P(\text{fix}\,\phi)$ also holds.

The admissible predicate P often involves the equality operator; proofs such as $p = q$ where p and q are recursively defined frequently occur. If p and q recurse at different rates then this is proved by showing that the two inequalities $p \sqsubseteq q$ and $q \sqsubseteq p$ hold. These may be proved by computation induction on the definition of p in the former inequality and on the definition of q in the latter.

Although computation induction is usually used to prove that inequalities hold another induction technique may prove useful. This is the recursion induction technique of McCarthy [McC] which states that

$$q = \phi(q) \Rightarrow \text{fix}\,\phi \sqsubseteq q.$$

Thus if p is defined by $p = \phi p$ then to prove that $p \sqsubseteq q$ all we need prove is that $q = \phi(q)$.

When proving inequalities using computation induction another technique may also have to be used. This is the technique where we argue elementwise that one process dominates the other.

4.2 The elementwise technique

When using our computation induction technique we are required to prove inequalities $p \sqsubseteq q$ for processes p and q. Since $\sqsubseteq$ and $\supseteq$ are identical and $p \supseteq q$ iff $\forall m.(m \in q \Rightarrow m \in p)$ we may argue elementwise. Hence to prove that $p \sqsubseteq q$ it is sufficient to prove $\forall m.(m \in q \Rightarrow m \in p)$.

In some cases when proving inequalities by using computation induction all that needs be done is to expand processes using their definitions and apply the induction hypothesis directly, without using the elementwise argument. This is often the case with processes defined using the singleton-set and union functions, and relies on the continuity of these functions.

The elementwise technique is necessary when processes are defined using the set-forming construct, by applying a process function to a suitable process argument.

Definition For any set D (possibly a domain) and powerdomain P_L (for some sort L), a function which is a member of the domain $D \rightarrow P_L$ is called a process function.

Members of the domain $P_M \rightarrow P_L$, for any sorts M and L, will therefore be process functions.

Supposing that we wish to prove $p \sqsubseteq q$ where q is defined using the set-forming construct. For some process function $G:L \to M$ and process $r:L$ then $q = G(r)$. If G is defined by

$$G(s) = \{\!\{ g(y) \mid pred(y), y \in s \}\!\}$$

our elementwise argument is as follows:

To prove that $p \sqsubseteq G(r)$ we prove that $\forall m. m \in G(r) \Rightarrow m \in p$.

By definition of our set-forming construct,

if $m \in \{\!\{ g(y) \mid pred(y), y \in r \}\!\}$ then

<u>either</u> $\{\!\{ g(y) \mid pred(y), y \in r \}\!\} = \perp_{P_M}$ and $m \in \perp_{P_M}$ by right-closure

<u>or</u> $\{\!\{ g(y) \mid pred(y), y \in r \}\!\} \neq \perp_{P_M}$ and

$m \sqsupseteq g(y)$ for some $y \in r$ such that $pred(y) \sqsubseteq true$, again by right-closure.

By considering these two cases we consider all possible ways that m is a member of $G(r)$. We then prove that $p \sqsubseteq G(r)$ by case analysis on these two cases; for the former we show that $p = \perp_{P_M}$ while the latter is proved by showing that such an m is also a member of process p. If p is defined recursively then we use the induction hypothesis to show this latter case.

If G is defined using our other basic set-forming construct then our elementwise argument goes as follows:

To prove that $p:M \sqsubseteq G(r)$ where $G:L \to M$ is defined by

$$G(s) = \{\!\{ \lambda:h(u,f) \mid \lambda \in M, \lambda:\langle u,f \rangle \in s \}\!\}$$

we note that $G(r) = \perp$ iff $r = \perp$, by our interpretation of this construct. Hence if $m \in G(r)$ then

<u>either</u> $r = \perp_{P_M}$

<u>or</u> $r \neq \perp_{P_M}$ and $m \sqsupseteq \lambda:h(u,f)$ for some $\lambda:\langle u,f \rangle \in r$ such that $\lambda \in M$.

Again if p is defined recursively the computation induction technique is used on the latter case to show that such an m is also a member of p. The former case is dealt with by proving that

for such a process r, $p = \bot_{P_M}$ as well. p will normally be a process function also parameterised on r and so $p = \bot_{P_M}$ again by our interpretation of the set-forming construct.

If we have a process function $H{:}L \mapsto M$ defined by

$$H(p) = U\{\!|\, h(x) \mid pred(x), x \in p \,|\!\}$$

where $h{:}D \to P_M$ and $P_L = \wp(D)$; then $m \in H(p) \Rightarrow m \in h(x)$ for some $x \in p$ such that $pred(x) \sqsubseteq true$. This follows from our interpretation of the set-forming construct and the definition of U.

When using the elementwise argument to prove that $p \sqsubseteq q$ for some processes p and q, if q is defined using a number of set-forming constructs together with the set functions $\cup$ and U then the techniques mentioned above must be repeated for each construct, as a member of q may be a member of any of the component constructs.

In the example which follows we perform a proof using computational induction but not our elementwise technique. Examples of the use of the elementwise technique appear in Chapter 5.

4.3 An example concerning queues

A process $q_{\bar{\beta}\alpha}(\varepsilon)$ which models the behaviour of an initially empty queue is defined by $q_{\bar{\beta}\alpha} : Z^* \to P_{\{\alpha, \bar{\beta}\}}$ where

$$q_{\bar{\beta}\alpha}(\varepsilon) = \{\!|\, \alpha : \langle o, \lambda z. q_{\bar{\beta}\alpha}(\varepsilon \curvearrowleft z) \rangle \,|\!\}$$

$$q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) = \{\!|\, \alpha : \langle o, \lambda z. q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1 \curvearrowleft z) \rangle, \bar{\beta} : \langle z_1, K(q_{\bar{\beta}\alpha}(s_1)) \rangle \,|\!\}$$

Z^* is not a domain but an unordered set of sequences of members of Z, with $\varepsilon \in Z^*$ the empty sequence.

A theorem concerning queues of unbounded capacity is that the "composition in series" of two queues behaves as a single queue. We then have the following theorem concerning processes modelling queues.

Theorem 4.3.1

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(s_1) = q_{\bar{\gamma}\alpha}(s_2 {}^\frown s_1)$$

where $^\frown$ is the usual concentration operation on sequences,and elements in the queues "move" from right to left.

Proof We require two lemmas;

Lemma 4.3.2

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) = q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel q_{\bar{\beta}\alpha}(s_1)$$

Proof

We prove that

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \sqsubseteq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel q_{\bar{\beta}\alpha}(s_1) \qquad (1)$$

and

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \sqsupseteq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel q_{\bar{\beta}\alpha}(s_1) \qquad (2)$$

By the definition of q and $\parallel$

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \geq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel q_{\bar{\beta}\alpha}(s_1)$$

and (1) follows immediately.

To show (2) we show that for all $i \geqslant o$

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \sqsupseteq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel_i q_{\bar{\beta}\alpha}(s_1) \qquad (3)$$

, (2) will then follow by computation induction.

Basis $q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel_o q_{\bar{\beta}\alpha}(s_1) = \bot \sqsubseteq q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1)$ as required .

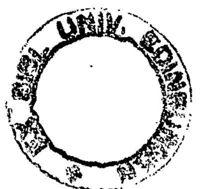
Induction step for some $i \geqslant o$ assume that

$$\forall s_1, s_2 . q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \sqsupseteq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel_i q_{\bar{\beta}\alpha}(s_1) \qquad (4)$$

and show that

$$\forall s_1, s_2 . q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(z_1 \curvearrowright s_1) \sqsupseteq q_{\bar{\gamma}\beta}(s_2 \curvearrowleft z_1) \parallel_{i+1} q_{\bar{\beta}\alpha}(s_1) \qquad (5)$$

We carry out case analysis on s_2.



<u>Case 1 ($s_2=\varepsilon$)</u>

$$q_{\bar\gamma\beta}(\varepsilon) \parallel q_{\bar\beta\alpha}(z_1 \frown s_1) = \{\alpha:\langle o, \lambda z. q_{\bar\gamma\beta}(\varepsilon) \parallel q_{\bar\beta\alpha}(z_1 \frown s_1 \frown z)\rangle\} \cup q_{\bar\gamma\beta}(\varepsilon \frown z_1) \parallel q_{\bar\beta\alpha}(s_1) \qquad (6)$$

by the definition of q and $\parallel$

$$\sqsupseteq \{\alpha:\langle o, \lambda z. q_{\bar\gamma\beta}(\varepsilon \frown z_1) \parallel_i q_{\bar\beta\alpha}(s_1 \frown z)\rangle\} \cup q_{\bar\gamma\beta}(\varepsilon \frown z_1) \parallel_{i+1} q_{\bar\beta\alpha}(s_1)$$

by (4) and that $\parallel \sqsupseteq \parallel_{i+1}$

$$= q_{\bar\gamma\beta}(\varepsilon \frown z_1) \parallel_{i+1} q_{\bar\beta\alpha}(s_1)$$

by the definition of q and $\parallel$, as required.

<u>Case 2 ($s_2 \neq \varepsilon$)</u>

Let $s_2 = z_3 \frown s_3$.

$$q_{\bar\gamma\beta}(z_3 \frown s_3) \parallel q_{\bar\beta\alpha}(z_1 \frown s_1) =$$

$$\{\alpha:\langle o, \lambda z. q_{\bar\gamma\beta}(z_3 \frown s_3) \parallel q_{\bar\beta\alpha}(z_1 \frown s_1 \frown z)\rangle, \bar\gamma:\langle z_3, K(q_{\bar\gamma\beta}(s_3)) \parallel q_{\bar\beta\alpha}(z_1 \frown s_1))\rangle\}$$
$$\cup\ q_{\bar\gamma\beta}(z_3 \frown s_3 \frown z_1) \parallel q_{\bar\beta\alpha}(s_1)$$

by the definition of q and $\parallel$

$$\sqsupseteq \{\alpha:\langle o, \lambda z. q_{\bar\gamma\beta}(z_3 \frown s_3 \frown z_1) \parallel_i q_{\bar\beta\alpha}(s_1 \frown z)\rangle, \bar\gamma:\langle z_3, K(q_{\bar\gamma\beta}(s_3 \frown z_1) \parallel_i q_{\bar\beta\alpha}(s_1))\rangle\}$$
$$\cup\ q_{\bar\gamma\beta}(z_3 \frown s_3 \frown z_1) \parallel_{i+1} q_{\bar\beta\alpha}(s_1)$$

by induction hypothesis (4) and that $\parallel \sqsupseteq \parallel_{i+1}$

$$= q_{\bar\gamma\beta}(z_3 \frown s_3 \frown z_1) \parallel_{i+1} q_{\bar\beta\alpha}(s_1)$$

by the definition of q and $\parallel$, hence (5) as required.

Hence Lemma 4.3.2 by computation induction .

<u>Lemma 4.3.3</u> $q_{\bar\gamma\beta}(s_2) \parallel q_{\bar\beta\alpha}(\varepsilon) = q_{\bar\gamma\alpha}(s_2)$

<u>Proof</u> we show

$$\forall s_2 . q_{\bar{\gamma}\beta}(s_2) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon) = (q_{\bar{\gamma}\alpha})_i(s_2) \qquad \text{for all } i \geq o \tag{7}$$

Basis result is immediate for $i=o$.

Induction step assume that for some $i \geq o$

$$\forall s_2 . q_{\bar{\gamma}\beta}(s_2) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon) = (q_{\bar{\gamma}\alpha})_i(s_2) \tag{8}$$

and we show that

$$\forall s_2 . q_{\bar{\gamma}\beta}(s_2) \parallel_{i+1} q_{\bar{\beta}\alpha}(\varepsilon) = (q_{\bar{\gamma}\alpha})_i(s_2) \tag{9}$$

by case analysis on s_2.

Case 1 ($s_2=\varepsilon$)

$$q_{\bar{\gamma}\beta}(\varepsilon) \parallel_{i+1} q_{\bar{\beta}\alpha}(\varepsilon) =$$

$$\{\!\!\{ \alpha : \langle o, \lambda z . q_{\bar{\gamma}\beta}(\varepsilon) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon^\frown z)\rangle \}\!\!\}$$

by definition of q and $\parallel$

$$= \{\!\!\{ \alpha : \langle o, \lambda z . q_{\bar{\gamma}\beta}(\varepsilon^\frown z) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon)\rangle \}\!\!\}$$

by Lemma 4.3.2

$$= \{\!\!\{ \alpha : \langle o, \lambda z . (q_{\bar{\gamma}\alpha})_i(s_2^\frown z)\rangle \}\!\!\}$$

by (8)

$= (q_{\bar{\gamma}\alpha})_{i+1}(s_2)$ by definition of q when $s_2=\varepsilon$, as required.

Case 2 ($s_2 \neq \varepsilon$) Let $s_2 = z_3{}^\frown s_3$

$$q_{\bar{\gamma}\beta}(z_3{}^\frown s_3) \parallel_{i+1} q_{\bar{\beta}\alpha}(\varepsilon) =$$

$$\{\!\!\{ \alpha : \langle o, \lambda z . q_{\bar{\gamma}\beta}(z_3{}^\frown s_3) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon^\frown z)\rangle, \bar{\gamma} : \langle z_3, K(q_{\bar{\gamma}\beta}(s_3) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon))\rangle \}\!\!\}$$

by definition of q and $\parallel$

$$= \{\!\!\{ \alpha : \langle o, \lambda z . q_{\bar{\gamma}\beta}(z_3{}^\frown s_3{}^\frown z) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon)\rangle, \bar{\gamma} : \langle z_3, K(q_{\bar{\gamma}\beta}(s_3) \parallel_i q_{\bar{\beta}\alpha}(\varepsilon))\rangle \}\!\!\}$$

by Lemma 4.3.2

$$= \{\!\!\{ \alpha : \langle o, \lambda z . (q_{\bar{\gamma}\alpha})_i(z_3{}^\frown s_3{}^\frown z)\rangle, \bar{\gamma} : \langle z_3, K((q_{\bar{\gamma}\alpha})_i(s_3))\rangle \}\!\!\}$$

by (8)

$= (q_{\bar{\gamma}\alpha})_{i+1}(z_3 \frown s_3)$ by definition of q.

Hence (9) and so also (7). Lemma then follows by computation induction .

Returning to the proof of our theorem, we have

$$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(s_1) = q_{\bar{\gamma}\beta}(s_2 \frown s_1) \parallel q_{\bar{\beta}\alpha}(\varepsilon)$$

by repeated use of Lemma 4.3.2 where

$s_1 = z_1 \frown z_2 \ldots \frown z_n \frown \varepsilon; n = \text{length}(s_1)$. Hence

$q_{\bar{\gamma}\beta}(s_2) \parallel q_{\bar{\beta}\alpha}(s_1) = q_{\bar{\gamma}\alpha}(s_2 \frown s_1)$ by Lemma 4.3.3,

as required .

4.4 Process function composition

We have seen in the preceding chapter how process proofs involving $\sqsubseteq$ may be performed using an elementwise argument, especially when these process are defined using the set-forming construct.

For process functions $F:L \to M$, $G:M \to N$ and $H:L \to N$ we may wish to prove the equality

$$G(F(p)) = H(p).$$

It would appear that we would always need to prove this by proving both inequalities using the elementwise argument. This is necessary since if $F(p) = \{\!\!\{ f(x) \mid \text{pred1}(x), x \in p \}\!\!\}$ and

$$G(q) = \{\!\!\{ g(y) \mid \text{pred2}(y), y \in q \}\!\!\}$$

m will be a member of $G(F(p))$ as follows;

<u>either</u> $G(F(p)) = \bot$

<u>or</u> $m \sqsupseteq g(y)$ where $y \in F(p)$ such that $\text{pred2}(y) \sqsubseteq \text{true}$.

But as y is a member of $F(p)$ then

<u>either</u> $F(p) = \bot$

<u>or</u> $y \sqsupseteq f(x)$ where $x \in p$ such that $\text{pred1}(x) \sqsubseteq \text{true}$.

Hence in proofs such as $G(F(p)) = H(p)$ we not only have to use the elementwise technique (which is more laborious than if we can produce a proof directly; such as proving that $I(p) = J(p)$ where I and J are recursive and recurse at the same rate), but we must consider three cases corresponding to how m can be a member of $G(F(p))$.

The following theorems allow us to compose process functions defined using the set-forming construct, so avoiding one "level" of the elementwise argument in certain proofs. This may also allow us to prove equalities directly, particularly if in the above $G \circ F$ recurses at the same rate as H.

We have three basic composition theorems.

Theorem 4.4.1 (composition)

for $F(p) = \{ f(x) \mid \phi_1(x), x \in p \}$

and $G(q) = \{ g(y) \mid \phi_2(y), y \in q \}$

then $G \circ F(p) = \{ g \circ f(x) \mid \phi_2(f(x)) \wedge \phi_1(x), x \in p \}$

Theorem 4.4.3 (composition)

for $F(p) = U\{ f(x) \mid \phi_1(x), x \in p \}$

and $G(q) = \{ g(y) \mid \phi_2(y), y \in q \}$

then $G \circ F(p) = U\{ G \circ f(x) \mid \phi_1(x), x \in p \}$

Theorem 4.4.6 (composition)

for $F(p) = U\{ f(x) \mid \phi_1(x), x \in p \}$

and $G(q) = \{ g(y) \mid \phi_2(y), y \in q \}$

then $F \circ G(q) = U\{ f \circ g(y) \mid \phi_2(y) \wedge \phi_1(g(y)), y \in q \}$

The proofs of these three theorems together with auxiliary lemmas are given in the appendix. The following lemma arises immediately from the interpretation of our set-forming construct and the usual set-theoretic properties.

Lemma 4.4.7 (composition)

for $F: S \to L$, $p: S$ [illegible]

then $F(p) = F(\{f(x) \mid \phi_1(x), x \in p\} \cup \{f'(x) \mid \phi_2(x), x \in p\}) = F(\{f(x) \mid \phi_1(x), x \in p\}) \cup F(\{f'(x) \mid \phi_2(x), x \in p\})$

The following corollary arises from the above composition theorems and lemma.

Corollary 4.4.8

if $F(p) = \{f(x) \mid \phi_1(x), x \in p\} \cup \{f'(x) \mid \phi_1'(x), x \in p\}$

and $G(q) = \{g(y) \mid \phi_2(y), y \in q\}$

then $GoF(p) = \{gof(x) \mid \phi_2(f(x)) \wedge \phi_1(x), x \in p\}$

$\qquad \cup \{gof'(x) \mid \phi_2(f'(x)) \wedge \phi_1'(x), x \in p\}$

The proof of this corollary is in the appendix.

Many corollaries such as this follow from the composition theorems above, where the process functions are defined using a number of set forming constructs, $\cup$ and U.

The composition theorems may be extended in a natural way to deal with process functions defined on more than one parameter. These theorems, their extensions and corollaries allow us to avoid using one level of the elementwise argument in proofs involving the composition of process functions. This proof technique is used in the following chapter to prove Rule (F5) of our process algebra. Here the use of a composition theorem removes the necessity of using the elementwise technique completely since a simple inductive proof is all that is needed. Hence these theorems do simplify proofs.

4.5 Proof techniques involving the $\|$ combinator

The combinator $\|$ appears frequently in process proofs particularly when we wish to prove that a composite agent is equivalent to a singular agent. Often the component agents will compute serially due to synchronisation, although they admit the possibility of concurrent computation. When proofs involving such composite agents take place the $\|$ combinator is used. The inherent seriality is modelled by only one communication being possible among the collection of processes, at some given instant. These techniques will be used by an example proof in the next section.

Two theorems prove useful in such cases. First we require some definitions.

Definition The function label $:P_L \to 2^L$ from processes to the subsets of the sort of these processes is defined by:
label (p) = $\{\alpha \mid \alpha : \langle u, f \rangle \in p\}$. This is not a continuous function as 2^L is not considered to be ordered in any way. For any process p of sort L, label (p) $\subseteq$ L.

Definition for p:L and q:M the internal labels of the pair of processes p and q are members of $L.M$.

Definition for p:L and q:M the external labels of the pair p and q are members of $L \cup M-(L.M)$.

This notion of internal and external labels can be extended to more than two processes.

Definition for $p_1:L_1,\ldots,p_n:L_n$ the internal labels of the processes $p_1,\ldots,p_n$ are those members of $\bigcup_{k,\ell} L_k.L_\ell$ where $1\leq k,\ell\leq n$ and k and ℓ are distinct.

Definition for $p_1:L_1,\ldots,p_n:L_n$ the external labels of the processes $p_1,\ldots,p_n$ are those members of the set $(\bigcup_i L_i)-(\bigcup_{k,\ell} L_k.L_\ell)$; where $1\leq i,k,\ell\leq n$ and k and ℓ are distinct.

Internal labels are internal to at least one pair of processes whilst external labels are external to all pairs of processes.

The following definition ensures that $\|$ is associative among a collection of processes. The proof of the associativity of $\|$ under restrictions on the sorts of its arguments appears in the appendix.

Definition for $p_1:L_1,\ldots,p_n:L_n$ their sorts satisfy the $\|$ restriction iff for any sorts L_i,L_j,L_k where $1\leq i,j,k\leq n$ and i,j,k are distinct

$L_i\cap(L_j.L_k) = \emptyset$.

These definitions allow us to state the following lemmas:

Lemma 4.5.1 for $p,q\neq\perp$ where p:L and q:M and

(a) $\text{label}(p)\cap\overline{\text{label}}(q) = \emptyset$;

(b) λ is the only external label of p,q which is a member of label(p) $\cup$ label(q);

(c) only one capability of p or q is labelled by λ then

$p\parallel q = \{\!|\lambda:\langle u,\lambda v.(p\parallel fv)\rangle|\!\}$ if $\lambda:\langle u,f\rangle\in q$ and $\lambda\in M$

or $p\parallel q = \{\!|\lambda:\langle u,\lambda v.(fv\parallel q)\rangle|\!\}$ if $\lambda:\langle u,f\rangle\in p$ and $\lambda\in L$

The proof of this lemma follows directly from the definition of $\|$.

<u>Lemma 4.5.2</u> for $p,q \neq \perp$ where $p:L$ and $q:M$ and

(a) all members of label $(p) \cup$ label(q) are internal labels of p,q;

(b) there is only one $\lambda \in$ label(p) and one $\mu \in$ label(q) such that $\lambda = \bar{\mu}$;

(c) only one capability $\lambda:\langle u,f\rangle \in p$ is labelled by λ and only one capability $\mu:\langle v,g\rangle \in q$ is labelled by μ ;

then $p \parallel q \;=\; fv \parallel gu$

The proof of this follows from the definition of $\parallel$.

The following two theorems are extensions of the above lemmas to deal with greater than two processes.

<u>Theorem 4.5.3 ($\parallel$ Theorem 1)</u>

for processes $p_i \neq \perp$ where $p_i:L_i$, $1 \leq i \leq n$ such that the L_i satisfy the $\parallel$ restriction and

(a) no labels $\lambda,\mu \in$ label$(p_1) \cup \ldots \cup$ label(p_n) such that $\lambda = \bar{\mu}$;

(b) $\exists$ only one $\lambda \in$ label$(p_1) \cup \ldots \cup$ label(p_n) such that λ is an external label of $p_1,\ldots,p_n$;

(c) $\exists$ only one member of $p_1 \ldots p_n$ labelled by λ, and $\lambda:\langle u,f\rangle \in p_k$ say, where $1 \leq k \leq n$;

then $\displaystyle\prod_{i=1}^{n} p_i = \{\lambda:\langle u, \lambda v.(p_1 \parallel \ldots \parallel p_{k-1} \parallel fv \parallel p_{k+1} \parallel \ldots \parallel p_n)\rangle\}$

<u>Proof</u> we may assume that $k=1$ by the commutivity of $\parallel$ and re-indexing. We then show that $\displaystyle\prod_{i=1}^{n} p_i = \{\lambda:\langle u, \lambda v.(fv \parallel \prod_{i=2}^{n} p_i)\rangle\}$

Let $\displaystyle\prod_{i=1}^{n} p_i = p_1 \parallel p$ were $p = (\prod_{i=2}^{n} p_i)$ by associativity of $\parallel$,

then label$(p) = \bigcup \{$label$(p_i) \mid 2 \leq i \leq n\}$ by the definition of label. By conditions (a), (b) and (c) we have that:

1) there are no labels $\lambda \in$ label(p_1) and $\mu \in$ label(p) such that $\lambda = \bar{\mu}$ and

2) $\exists$ only one external label $\lambda \in$ label$(p_1) \cup$ label(p) and this only

labels one capability of $p_1 \cup p$, namely $\lambda:\langle u,f\rangle \in p_1$.

Then $\mathop{\|}\limits_{i=1}^{n} p_i = \{\!\{\lambda:\langle u, \lambda v.(fv \,\|\, p)\rangle\}\!\}$

by Lemma 4.5.1

$= \{\!\{\lambda:\langle u, \lambda v.(fv \,\|\, (\mathop{\|}\limits_{i=2}^{n} p_i))\rangle\}\!\}$

$= \{\!\{\lambda:\langle u, \lambda v.(fv \,\|\, \mathop{\|}\limits_{i=2}^{n} p_i)\rangle\}\!\}$

by the associativity of $\|$, as required .

Theorem 4.5.4 ($\|$ Theorem 2)

for processes $p_i \neq \bot$ where $p_i:L_i, 1 \leq i \leq n$ such that the L_i satisfy $\|$ restriction and

(a) for all $\lambda \in \cup\{\text{label}(p_i) \mid 1 \leq i \leq n\}$, λ is an internal label of $p_1, \ldots, p_n$;

(b) there is only one λ and one μ members of $\cup\{\text{label}(p_i) \mid 1 \leq i \leq n\}$ such that $\lambda = \overline{\mu}$;

(c) for $k < \ell$, λ is only a member of L_k and μ is only a member of L_ℓ such that $\lambda:\langle u,f\rangle$ and $\mu:\langle v,g\rangle$ are the only so labelled capabilities of p_k and p_ℓ respectively;

then $\mathop{\|}\limits_{i=1}^{n} p_i = p_1 \| \cdots \| p_{k-1} \| fv \| p_{k+1} \cdots \| p_{\ell-1} \| gu \| p_{\ell+1} \| \cdots \| p_n$

Proof By associativity, commutivity and reindexing let $\lambda:\langle u,f\rangle \in p_1$ and $\mu:\langle v,g\rangle \in p_2$ such that (b) and (c) are satisfied. Thus $k=1$ and $\ell=2$ and what we are required to prove is that

$$\mathop{\|}\limits_{i=1}^{n} p_i = fv \,\|\, gu \,\|\, \mathop{\|}\limits_{i=3}^{n} p_i.$$

Let $\mathop{\|}\limits_{i=1}^{n} p_i = p_1 \| p_2 \| p$ where $p = \mathop{\|}\limits_{i=3}^{n} p_i$.

label (p) = label $(p_3) \cup \ldots \cup \text{label}(p_n)$ by (c). By (c) again all members of label (p) are external members of $p_3, \ldots, p_n$. By (a) and (b), $\|$, and the definition of internal ,

$$p_2 \| \cdot p = \{\!\!\{ \mu : \langle v, \lambda y.(gy \| p) \rangle \}\!\!\} \cup \{\!\!\{ \alpha_i : \langle \omega_i, \lambda y.(p_2 \| h_i y) \rangle \mid \alpha_i : \langle \omega_i, h_i \rangle \in p \}\!\!\}$$

where μ and the α_i are members of label(p_2)$\cup$label(p) which are external to $p_2, p_3, \ldots, p_n$.

By (a) and Lemma 4.5.2 we then have that

$$p_1 \| (p_2 \| p) = fv \| (gu \| p),$$

hence

$$\prod_{i=1}^{n} p_i = fv \| gu \| \prod_{i=3}^{n} p_i$$ as required, by the associativity of $\|$.

A proof using these two theorems is given in the next section.

4.6 The card reader theorem

In Chapter 2 we produced a process cardreader which models the behaviour of an existing cardreader by modelling its components.

We can also define a process which models the desired behaviour of a cardreader. This desired behaviour is what we would expect; a card is read and a sequence of values with a delimiter is produced before the next card is read, and so on. This behaviour (of a conceptual or abstract cardreader) is given by the process abs:$\{\overline{inc}, out, neg\}$ where

$$abs = \{\!\!\{ \overline{inc} : \langle o, \lambda c.\, send(c,1) \rangle \}\!\!\}$$ and

send : CARD $\times$ N $\rightarrow P_{\{\overline{inc}, out, neg\}}$ is defined by

$$send(c,j) = (j=81) \rightarrow \{\!\!\{ neg : \langle o, K(abs) \rangle \}\!\!\}, \{\!\!\{ out : \langle change(c(j)), K(send(c,j+1)) \rangle \}\!\!\}$$

, where change: COLUMN $\rightarrow$ N. The labels used here correspond to those used in cardreader, with the same domains.

To show that the software driver component of the cardreader (which is modelled by the driver process) controls the hardware components in a correct manner, we prove that

<u>Theorem 4.6.1 (cardreader)</u>

cardreader = abs. This theorem illustrates that even though our cardreader can carry out concurrent actions, due to synchronising communications it has serial behaviour. This seriality helps in the proof in that the $\|$ theorems of the previous section may be used.

The process cardreader: $P_{\{\overline{inc}, out, neg\}}$ is defined by:

$$cardreader = readerdevice \parallel bufferregister \parallel statusregister \parallel driver$$

where readerdevice: $P_{\{\overline{inc}, \overline{Sn1}, up1, \overline{Sn2}, val\}}$ is defined by

$$readerdevice = \{\!\!\{ \overline{Sn1}: \langle o, K\{\!\!\{ up1: \langle\langle 9,1\rangle, K\{\!\!\{ \overline{inc}: \langle o, \lambda c.\, countsend(1)\rangle \}\!\!\} \rangle \}\!\!\} \rangle \}\!\!\}$$

and

$$countsend(j) = \{\!\!\{ \overline{Sn2}: \langle o, K((j=81) \rightarrow \{\!\!\{ up1: \langle\langle 14,1\rangle, K(readerdevice)\rangle \}\!\!\},$$
$$\{\!\!\{ val: \langle change(c(j)), K(countsend(j+1))\rangle \}\!\!\}) \rangle \}\!\!\}$$

bufferregister : $P_{\{\overline{val}, \overline{Sn3}, out\}}$ is defined by

$$bufferregister = \{\!\!\{ \overline{val}: \langle o, \lambda n. \{\!\!\{ \overline{Sn3}: \langle o, K\{\!\!\{ out: \langle u, K(bufferregister)\rangle \}\!\!\} \rangle \}\!\!\} \rangle \}\!\!\}$$

statusregister: $P_{\{Sn1, ans, \overline{up2}, ask, \overline{up1}\}}$ is defined by

statusregister = status(d) where

$$status(b) = \{\!\!\{ \overline{ask}: \langle o, \lambda n. \{\!\!\{ ans: \langle b(n+1), K(status(b))\rangle \}\!\!\} \rangle,$$
$$\overline{up2}: \langle o, \lambda m \in (N \times T).\, (m(1)=o \wedge m(2)=1) \rightarrow$$
$$\{\!\!\{ Sn1: \langle o, K(status(\langle 1, b(2), \ldots, b(8), o, \ldots o\rangle))\rangle \}\!\!\},$$
$$status(\langle o, \ldots, b(m(1)), m(2), b(m(1)+2), o \ldots o\rangle)\rangle,$$
$$up1: \langle o, \lambda m \in (N \times T).\, status(\langle o, \ldots, b(m(1)), m(2), b(m(1)+1), \ldots$$
$$\ldots b(16)\rangle)\rangle \}\!\!\}$$

and $d = \langle o,1,o,o,o,o,1,o,o,o,o,o,o,o,o,o\rangle$

The process driver: $P\{Sn2, ask, up2, \overline{ans}, reg, Sn3\}$ is defined here slightly differently from that in Chapter 2 to avoid writing large expressions in the proof.

$$\begin{aligned}
\text{driver} &= \{\!\!\{\, ask{:}\langle 9, K(\text{driver})\rangle \,\}\!\!\} \quad \text{where} \\
\text{driver-1} &= \{\!\!\{\, \overline{ans}{:}\langle o, \lambda t.(t{=}1) \rightarrow \text{driver}, \text{driver-2}\rangle \,\}\!\!\} \\
\text{driver-2} &= \{\!\!\{\, ask{:}\langle 8, K(\text{driver-3})\rangle \,\}\!\!\} \\
\text{driver-3} &= \{\!\!\{\, \overline{ans}{:}\langle o, \lambda t.(t{=}1) \rightarrow \text{driver}, \text{driver-4}\rangle \,\}\!\!\} \\
\text{driver-4} &= \{\!\!\{\, up2{:}\langle\langle 6,1\rangle, K(\text{driver-5})\rangle \,\}\!\!\} \\
\text{driver-5} &= \{\!\!\{\, up2{:}\langle\langle o,1\rangle, K(\text{doio})\rangle \,\}\!\!\} \\
\text{doio} &= \{\!\!\{\, Sn2{:}\langle o, K(\text{doio-1})\rangle \,\}\!\!\} \\
\text{doio-1} &= \{\!\!\{\, ask{:}\langle 15, K(\text{doio-2})\rangle \,\}\!\!\} \\
\text{doio-2} &= \{\!\!\{\, \overline{ans}{:}\langle o, \lambda t.(t{=}1) \rightarrow \text{endcard}, \text{doio-3}\rangle \,\}\!\!\} \\
\text{doio-3} &= \{\!\!\{\, ask{:}\langle 14, K(\text{doio-4})\rangle \,\}\!\!\} \\
\text{doio-4} &= \{\!\!\{\, \overline{ans}{:}\langle o, \lambda t.(t{=}1) \rightarrow \text{endcard}, \text{doio-5}\rangle \,\}\!\!\} \\
\text{doio-5} &= \{\!\!\{\, Sn3{:}\langle o, K(\text{doio})\rangle \,\}\!\!\} \\
\text{endcard} &= \{\!\!\{\, \varepsilon_5{:}\langle -1, K(\text{endcard-1})\rangle \,\}\!\!\} \\
\text{endcard-1} &= \{\!\!\{\, \varepsilon_3{:}\langle\langle 6,o\rangle, K(\text{driver})\rangle \,\}\!\!\}
\end{aligned}$$

This process lends itself to being defined in this manner since it is deterministic.

<u>Proof of Theorem 4.6.1</u>

As the sorts of processes readerdevice, bufferregister, statusregister and driver satisfy the $\|$ restriction we may use $\|$ theorems freely.

$$\begin{aligned}
&\text{readerdevice} \,\|\, \text{bufferregister} \,\|\, \text{statusregister} \,\|\, \text{driver} \\
&= \text{readerdevice} \,\|\, \text{bufferregister} \,\|\, \{\!\!\{\, ans{:}\langle b(10), K(\text{status } b)\rangle \,\}\!\!\} \,\|\, \text{driver-1} \\
&= \text{readerdevice} \,\|\, \text{bufferregister} \,\|\, \text{status}(b) \,\|\, \text{driver-2}
\end{aligned}$$

= readerdevice ‖ bufferregister ‖ ⦃ans:⟨b(9),K(status(b))⟩⦄ ‖ driver-3

= readerdevice ‖ bufferregister ‖ status(b) ‖ driver-4

= readerdevice ‖ bufferregister ‖ status(b) ‖ driver-5

= readerdevice ‖ bufferregister ‖ ⦃Sn1:⟨o,K(status(b'))⟩⦄ ‖ doio

where b' = ⟨1100001000000000⟩

= ⦃up1:⟨⟨9,T⟩,K⦃$\overline{\text{inc}}$:⟨o,λc.countsend(1)⟩⦄⟩⦄ ‖ bufferregister ‖ ...

...status(b') ‖ doio

= ⦃$\overline{\text{inc}}$:⟨o,λc.countsend(1)⟩⦄ ‖ bufferregister ‖ status(b'') ‖ doio

where b'' = ⟨1100001001000000⟩

by repeated use of ‖ Theorem 2

= ⦃$\overline{\text{inc}}$:⟨o,λc.countsend(1)⟩⦄ ‖ bufferregister ‖ status(b'') ‖ doio

by ‖ Theorem 1

Since abs = ⦃$\overline{\text{inc}}$:⟨o,λc.send(c,1)⟩⦄ to prove that cardreader = abs it is enough to show that countsend(1) ‖ bufferregister ‖ status(b'')... ..‖doio = send(c,1). (1)

Since countsend and send recurse in a similar manner we may prove (1) by computation induction on the definitions of countsend and send.

We shall prove that

$\forall$i.countsend_j(i) ‖ bufferregister ‖ status(b'') ‖ doio = send_j(c,i) (2)

for all $j \geq o$, and (1) then follows by computation induction.

<u>Basis</u> $\forall$i. $\text{send}_o(c,i) = \bot$ and $\text{countsend}_o(i) = \bot$.

Since ‖ is strict by definition,

countsend_o(i) ‖ bufferregister ‖ status(b'') ‖ doio = $\bot$,

as required.

Induction step assume (2) for some $k \geq o$. We then prove (2) for $k+1$ by case analysis on i. Since $i \in N$ and N is a set (not a domain) we have the cases $i=81$ and $1 \leq i < 81$.

Case i=81

$\text{Countsend}_{k+1}(81) \| \text{bufferregister} \| \text{status}(b'') \| \text{doio}$

$= \text{readerdevice}_{k+1} \| \text{bufferregister} \| \text{status}(b''') \| \text{endcard}$

by $\|$ Theorem 2 six times, where $b''' = \langle 1100001001000010 \rangle$

$= \{\text{neg}: \langle -1, K(\text{readerdevice}_{k+1} \| \text{bufferregister} \| \text{status}(b''') \| \ldots$

$\ldots \text{endcard}-1) \rangle \}$ by $\|$ Theorem 1.

Since $\text{send}_{k+1}(c,81) = \{\text{neg}: \langle -1, K\text{abs}_{k+1} \rangle \}$; to prove (2) for k+1 it is sufficient to prove that

$$\text{readerdevice}_{k+1} \| \text{bufferregister} \| \text{status}(b''') \| \text{endcard}-1 = \text{abs}_{k+1}.$$

Now

$\text{readerdevice}_{k+1} \| \text{bufferregister} \| \text{status}(b''') \| \text{endcard}-1$

$= \text{readerdevice}_{k+1} \| \text{bufferregister} \| \text{status}(b^{1V}) \| \text{driver}$

by $\|$Theorem 2, where $b^{1V} = \langle 010 \ldots\ldots 0 \rangle$

$= \text{readerdevice}_{k+1} \| \text{bufferregister} \| \{\text{Snl}: \langle o, K.\text{status}(b') \rangle \} \| \text{doio}$

by $\|$ Theorem 2 six times

$= \{\overline{\text{inc}}: \langle o, \lambda c.\text{countsend}_k(1) \rangle \} \| \text{bufferregister} \| \text{status}(b'') \| \text{doio}$

by $\|$ Theorem 2 twice

$= \{\overline{\text{inc}}: \langle o, \lambda c.(\text{countsend}_k(1) \| \text{bufferregister} \| \text{status}(b'') \| \text{doio}) \rangle \}$

by $\|$ Theorem 1

$= \{\overline{\text{inc}}: \langle o, \lambda c.\text{send}_k(c,1) \rangle \}$

by induction hypothesis

$= \text{abs}_{k+1}$, as required.

Case $1 \leq i < 81$

$\text{countsend}_{k+1}(i) \parallel \text{bufferregister} \parallel \text{status}(b'') \parallel \text{doio}$

$= \text{countsend}_k(i+1) \parallel \{\text{out}:\langle \text{change}(c(i)), K\,\text{bufferregister}\rangle\} \parallel \text{status}(b'')\ldots$

$\ldots \parallel \text{doio}$ by $\parallel$ Theorem 2 seven times

$= \{\text{out}:\langle \text{change}(c(i)), K(\text{countsend}_k(i+1) \parallel \text{bufferregister} \parallel \text{status}(b'')\ldots$

$\ldots \parallel \text{doio})\rangle\}$ by $\parallel$ Theorem 1

$= \{\text{out}:\langle \text{change}(c(i)), K(\text{send}_k(c, i+1))\rangle\}$

by induction hypothesis

$= \text{send}_{k+1}(c, i)$ where $1 \leq i < 81$, as required.

Since we have proved both case $i=81$ and case $1 \leq i < 81$ we have proved the cardreader theorem.

This theorem utilises the $\parallel$ theorem due to the seriality imposed by the driver process. But similar proofs can also take place when we wish to show that two non-determinate processes are equal. These processes will then have a finite, equal number of members and the theorems will be used to prove that their renewals are equivalent.

In this example we avoid the need to use an elementwise argument since the two processes to be proved equal contain the same finite number (one) of members and the two processes were defined by process functions recursing at the same rate.

CHAPTER 5

PROCESSES AS A FLOW ALGEBRA

In this chapter we prove that certain laws, such as the associativity of $|$, hold in the process model. These laws, known as the laws of flow, are rather intuitive and are used frequently in process proofs.

We define a category of algebras known as flow algebras. These algebras have the property that the laws of flow hold in them; thus the algebra of processes is a flow algebra.

The category of flow algebras contains a distinguished member known as the algebra of nets (or of flowgraphs). This algebra involves a formalisation of the net concept used in Chapter 1, and is defined in Milne and Milner [Mil 1]. We shall not consider nets further (since we are concerned with processes) but note that for a similar category of flow algebras Milner [Mil 4] demonstrates that net algebras are free in this category. He also mentions that the algebra of nets in our category of flow algebras is initial in its category.

We have the following definition for initiality:

<u>Definition</u> An algebra $\mathcal{S}$ is initial in a category $\mathcal{C}$ of algebras iff for every A in $\mathcal{C}$ there exists a unique homomorphism $h_A:\mathcal{S}\rightarrow A$.

Following the work of Goguen, Thatcher, Wagner and Wright [Gog] we interpret the algebra of nets as a syntactic algebra with our process algebra specifying one possible semantics for it. Since

the net algebra is initial, there exists a unique homomorphism from this to an algebra of processes. This is a semantic function assigning a meaning in terms of processes to each net. A net is a syntactic construct in that two flow expressions denote the same net iff they can be proved equivalent using only the laws of flow.

5.1 Flow algebras

A (Σ,Γ)-flow algebra is defined by the following, where Σ is the alphabet of names used in Chapter 1, with the set of labels $\Lambda = \Sigma \cup \bar{\Sigma}$.

Definition B is a (Σ,Γ)-flow algebra if it has

(a) a phylum B_L for each sort L;

(b) nullary operations $c^B\lambda_1 \ldots \lambda_k : \{\lambda_1, \ldots, \lambda_k\}$ for each $c \in \Gamma$ and labels $\lambda_1, \ldots, \lambda_k$;

(c) each $c \in \Gamma$ has a sign, where sign $c \in \{+,-\}^*$;

(d) binary operations $|^B : B_L x B_M \rightarrow B_{L \cup M}$, for each pair of sorts L, M;

(e) unary operations $\backslash^B\alpha : B_L \rightarrow B_{L-\{\alpha,\bar{\alpha}\}}$, for each sort L and $\alpha \in \Sigma$;

(f) and the following laws of flow hold, for $x_i : L_i$.

(F1) $x_1 |^B x_2 = x_2 |^B x_1$

(F2) $x_1 |^B (x_2 |^B x_3) = (x_1 |^B x_2) |^B x_3$

(F3) $x_1 \backslash^B \alpha = x_1$ where $\{\alpha,\bar{\alpha}\} \cap L_1 = \emptyset$

(F4) $x_1 \backslash^B \alpha \backslash^B \beta = x_1 \backslash^B \beta \backslash^B \alpha$

(F5) $(x_1 |^B x_2) \backslash^B \alpha = (x_1 \backslash^B \alpha) |^B (x_2 \backslash^B \alpha)$ where $\{\alpha,\bar{\alpha}\} \cap L_1 \cap \bar{L}_2 = \emptyset$

(F6) $E \backslash^B \alpha = E[\beta/\alpha] \backslash^B \beta$ where $\beta, \bar{\beta}$ do not occur in E, some arbitrary flow expression.

Flow expressions are constructed from the operators $c^B\lambda_1 \ldots \lambda_k$, $|^B$ and $\backslash^B\alpha$, for arbitrary algebra B. $[\beta/\alpha]$ is the meta-syntactic substitution operation and is not considered as an operation of our algebra.

The laws of flow may be motivated by considering the informal net concept of Chapter 1. The commutivity and associativity laws (F1) and (F2) follow since we do not wish to distinguish how a net is constructed using $|$ since the resulting nets will be undistinguishable. This corresponds to building systems of hardware agents where we do not care in what order communication wires are soldered between components. Law (F3) is motivated by considering an agent with no α or $\bar{\alpha}$ line. When this is removed no change therefore occurs. Law (F4) also follows from considering nets whilst Law (F5) says that connecting then removing is the same as removing then connecting two components not connected by the lines to be removed. Law (F6) allows us to relabel agents with a new label providing it does not already occur.

These laws are therefore desirable in our process model as it purports to model concurrent agents satisfying these laws.

A somewhat similar flow algebra may have substitution as an operation, together with extra laws involving this operation replacing law (F6).

5.2 Processes are a flow algebra

The phyla P_L of the (Σ,Γ)-process algebra P, the unary operations $\backslash^P\alpha$ and binary operations $|^P$ over respective phyla were defined in Chapter 1.

The nullary operations $c^P\lambda_1\ldots\lambda_k$ of P are defined using substitution, which is not a flow algebra operation. Given a command c with sign +- (for example) we choose distinct names α, β say and specify $c^P\alpha\bar{\beta} = p$, for some arbitrary process $p:\{\alpha,\bar{\beta}\}$. Each nullary operation $c^P\lambda_1\lambda_2$ can then be given by using the substitution operation to replace the λ's by labels.

The semantic substitution operation $\{\beta/\alpha\}:P_L \to P_{L\{\beta/\alpha\}}$, where sort $L\{\beta/\alpha\}$ is the replacement of α by β and $\bar{\alpha}$ by $\bar{\beta}$ in sort L, as defined by:

<u>Definition</u> for p:L where $\beta,\bar{\beta}\notin L$,

$$
\begin{aligned}
p\{\beta/\alpha\} = & \{\!\{\beta:\langle u, \lambda v.(fv\{\beta/\alpha\})\rangle \mid \alpha:\langle u,f\rangle \in p\}\!\} \\
& \cup \{\!\{\bar{\beta}:\langle u, \lambda v.(fv\{\beta/\alpha\})\rangle \mid \bar{\alpha}:\langle u,f\rangle \in p\}\!\} \\
& \cup \{\!\{\lambda:\langle u, \lambda v.(fv\{\beta/\alpha\})\rangle \mid \lambda \notin \{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle \in p\}\!\}
\end{aligned}
$$

We now prove that the laws of flow hold in our process algebra, for arbitrary processes $p_i:L_i$. The superfix p on operations will be omitted.

These proofs are rather routine; the proof of law(F2)(associativity) is perhaps the most interesting.

<u>Law (F1)</u> $p_1|p_2 = p_2|p_1$

<u>Proof</u> Let $p_1|p_2 = p_2|_R p_1$ for all p_1,p_2 where $|:L_1 \times L_2 \to L_1 \cup L_2$ and $|_R:L_2 \times L_1 \to L_1 \cup L_2$.
Note that $|_R$ is <u>not</u> the R-th truncation of $|$.

$Y\phi = |$ where $\phi:(L_1 \times L_2 \to L_1 \cup L_2) \to (L_1 \times L_2 \to L_1 \cup L_2)$

$$\phi = \lambda \not| . \lambda(p_1, p_2). [$$

$$\{ \lambda : \langle u_1, \lambda v. f_1 v \not| p_2 \rangle | \lambda \in L_1, \lambda : \langle u_1, f_1 \rangle \in p_1 \}$$

$$\cup \{ \mu : \langle u_2, \lambda v. p_1 \not| f_2 v \rangle | \mu \in L_2, \mu : \langle u_2, f_2 \rangle \in p_2 \}$$

$$\cup \bigcup \{ f_1 u_2 \not| f_2 u_1 | \lambda \in L_1 \wedge \bar{\lambda} \in L_2, \lambda : \langle u_1, f_1 \rangle \in p_1, \bar{\lambda} : \langle u_2, f_2 \rangle \in p_2 \}]$$

$|_R$ is defined recursively by

$$|_R = \lambda(q_2, q_1). [$$

$$\{ \rho : \langle s_1, \lambda v. g_1 v |_R q_1 \rangle | \rho \in L_2, \rho : \langle s_1, g_1 \rangle \in q_2 \}$$

$$\{ \sigma : \langle s_2, \lambda v. q_2 |_R g_2 v \rangle | \sigma \in L_1, \sigma : \langle s_2, g_2 \rangle \in q_1 \}$$

$$\{ g_1 s_2 |_R g_2 s_1 | \rho \in L_2 \wedge \bar{\rho} \in L_1, \rho : \langle s_1, g_1 \rangle \in q_2, \bar{\rho} : \langle s_2, g_2 \rangle \in q_1 \}]$$

Now $\phi |_R = |_R$ by the above two definitions and the renaming of variables, hence $|_R$ is a fixed point of ϕ. Since $|$ is the least fixed point of ,

$$| \sqsubseteq |_R \text{ (recursion induction techniques).}$$

The inequality $|_R \sqsubseteq |$ follows in a similar manner, hence $| \equiv |_R$ and

$$p_1 | p_2 = p_2 | p_1 \text{ as required .}$$

We require two lemmas for the proof of Law (F2).

<u>Lemma 5.2.1 (strictness of $|$)</u> $\perp | p = p | \perp = \perp$

<u>Proof</u> We prove that $p_1 = \perp \Rightarrow p_1 | p_2 = \perp$. The lemma then follows by Law (F1).

Let $P_{L_1} = \mathcal{P}(D_{L_1})$ where $p_1 : L_1$.

By definition of $|$,

$$\{ \lambda : \langle u_1, \lambda v. f_1 v | p_2 \rangle | \lambda \in L_1, \lambda : \langle u_1, f_1 \rangle \in p_1 \} \subseteq p_1 | p_2.$$

Since $p_1 = \perp_{P_{L_1}}$ then by our interpretation of the above set-forming

construct, $\perp_{P_{L_1 \cup L_2}} \subseteq p_1|p_2$. Since $\perp_{P_{L_1 \cup L_2}}$ is the "greatest" member of $P_{L_1 \cup L_2}$, $\perp_{P_{L_1 \cup L_2}} = p_1|p_2$ as required

Lemma 5.2.2

$$p_1|p_2 = \perp \Rightarrow p_1|(p_2|p_3) = \perp$$

Proof We show that for all p_1, p_2, p_3 and k where $k \geqslant o$

$$p_1|p_2 = \perp \Rightarrow p_1|(p_2 |_k p_3) = \perp .$$

Basis $p_1|(p_2 |_o p_3) = p_1|\perp = \perp$

by Lemma 5.2.1, as required.

Induction step Assume that $\forall p_1, p_2, p_3 . p_1|p_2 = \perp \Rightarrow p_1|(p_2|_k p_3) = \perp$ for some $k \geqslant o$. (1)

We prove that $\forall p_1, p_2, p_3 . p_1|p_2 = \perp \Rightarrow p_1|(p_2|_{k+1} p_3) = \perp$ (2)

by case analysis. $p_1|p_2 = \perp$ under three cases by the definition of $|$.

Case 1 $p_1 = \perp$, hence by Lemma 5.2.1,

$p_1|(p_2|_{k+1} p_3) = \perp$ as required.

Case 2 $p_2 = \perp$, hence by Lemma 5.2.1,

$p_2|_{k+1} p_3 = \perp$ and

$p_1|(p_2|_{k+1} p_3) = \perp$ as required, by Lemma 5.2.1.

Case 3 $\lambda : \langle u_1, f_1 \rangle \in p_1$ and $\bar{\lambda} : \langle u_2, f_2 \rangle \in p_2$ (3)

such that $f_1 u_2 | f_2 u_1 = \perp$ (4)

By (4) and induction hypothesis

$$f_1u_2|(f_2u_1|_kp_3) = \bot \tag{5}$$

By (3) and the definition of $|$

$$\bar{\lambda}:\langle u_2, \lambda v. f_2v|_kp_3\rangle \in p_2|_{k+1}p_3 \tag{6}$$

Hence by (3), (6) and the definition of $|$,

$$f_1u_2|(f_2u_1|_kp_3) \subseteq p_1|(p_2|_{k+1}p_3)$$

and by (5) and right closure,

$$p_1|(p_2|_{k+1}p_3) = \bot \text{ as required .}$$

<u>Law (F2)</u> $p|(q|r) = (p|q)|r$

We show that $p|(q|r) \sqsubseteq (p|q)|r$ (1)

and $p|(q|r) \sqsupseteq (p|q)|r$ (2)

To prove (1), show $\forall i,j,p,q,r$

$$p|_i(q|_jr) \sqsubseteq (p|q)|r \tag{3}$$

by induction on the sum of i and j. Then (2) follows from (1) by Law (F1).

<u>Basis</u> We prove (3) when i=0, or j=0. In fact it follows immediately using $|_o = \bot$ and Lemma 5.2.1.

<u>Induction step</u>

To prove (3) it is enough to show that

$$m \in (p|q)|r \Rightarrow m \in p|_i(q|_jr) \tag{4}$$

when $i,j > 0$, under the inductive assumption

$$\forall p,q,r.\ p|_k(q|_hr) \sqsubseteq (p|q)|r \text{ with } k + h < i + j \tag{5}$$

That is, assume (5) and show (4) by cases.

Now m can arise as a member of $(p|q)|r$ in three ways:

Case 1

$$m \in \{\mu:\langle v, \lambda x.(gx|r)\rangle \mid \mu:\langle v,g\rangle \in p|q\} \tag{6}$$

Then either $p|q = \perp$

and so by Lemma 5.2.2, $p|(q|r) = \perp$,

hence $p|_i(q|_j r) = \perp \ni m$

or $p|q \neq \perp$, in which case

$$m \ni \mu:\langle v, \lambda x.(gx|r)\rangle \tag{7}$$

where $\mu:\langle v,g\rangle \in p|q$. This later can arise in $p|q$ in three ways:

Case 1.1 $\exists \mu:\langle u,f\rangle \in p$ (8)

such that

$$\mu:\langle v,g\rangle \ni \mu:\langle u, \lambda x.(fx|q)\rangle. \tag{9}$$

From (8) we have

$$\mu:\langle u, \lambda x.(fx|_{i-1}(q|_j r))\rangle \in p|_i(q|_j r).$$

So by induction hypothesis (5) and right closure,

$$\mu:\langle u, \lambda x.((fx|q)|r)\rangle \in p|_i(q|_j r)$$

and by right closure again, using (7) and (9)

$m \in p|_i(q|_j r)$ as required.

Case 1.2 $\mu:\langle u,f\rangle \in q$

such that

$$\mu:\langle v,g\rangle \ni \mu:\langle u, \lambda x.(p|fx)\rangle$$

The proof in this case is similar to that for 1.1 and is omitted.

Case 1.3 There exists some $\lambda:\langle u,f\rangle \in p$ (10)

and some $\bar{\lambda}:\langle u',f'\rangle \in q$ (11)

such that $\mu:\langle v,g\rangle \in fu'|f'u$ (12)

From (11), $\bar{\lambda}:\langle u',\lambda x.(f'x|_{j-1}r)\rangle \in q|_j r$

So using (10),

$$fu'|_{i-1}(f'u|_{i-1}r) \subseteq p|_i(q|_j r) \quad (13)$$

But $(fu'|f'u)|r \subseteq fu'|_{i-1}(f'u|_{j-1}r)$ (14)

by induction hypothesis (5), using that $\subseteq$ equals $\sqsupseteq$

and by (12)

$$\mu:\langle v,\lambda x.(gx)|r\rangle \in (fu'|f'u)|r$$

and hence by (7) and right closure

$$m \in (fu'|f'u)|r \quad (15)$$

and by (13), (14) and (15)

$m \in p|_i(q|_j r)$ as required.

Case 2

$$m \in \{\mu:\langle v,\lambda x.((p|q)|gx)\rangle \mid \mu:\langle v,g\rangle \in r\} \quad (16)$$

Then either $r = \perp$

and so by Lemma 5.2.1, $p|(q|r) = \perp$.

hence $p|_i(q|_j r) = \perp \ni m$

or $r \neq \perp$, in which case

$$m \sqsupseteq \mu:\langle v,\lambda x.((p|q)|gx)\rangle \quad (17)$$

where $\mu:\langle v,g\rangle \in r$. Now

$$\mu:\langle v,\lambda x.(q|_{j-1}gx)\rangle \in q|_j r$$

by definition of $|$, and so

$$\mu:\langle v,\lambda x.(p|_{i-1}(q|_{j-1}gx))\rangle \in p|_i(q|_j r) \quad (18)$$

again using the definition of $|$.

Using induction hypothesis (5) and monotonicity

$$\mu:\langle v,\lambda x.((p|q)|gx)\rangle \sqsupseteq \mu:\langle v,\lambda x.(p|_{i-1}(q|_{j-1}gx))\rangle \quad (19)$$

and so by (17), (18), (19) and right closure

$m \in p|_i(q|_j r)$ as required.

Case 3 $m \in gv' | g'v$ (20)

where $\mu: \langle v, g \rangle \in p | q$ (21)

and $\bar{\mu}: \langle v', g' \rangle \in r$ (22)

Now $\mu: \langle v, g \rangle \in p | q$ can arise in three ways:

Case 3.1 $\exists \mu: \langle u, f \rangle \in p$

such that

$$\mu: \langle v, g \rangle \ni \mu: \langle u, \lambda x. (fx | q) \rangle$$

The proof of this case is again similar to others and is omitted.

Case 3.2 $\exists \mu: \langle u, f \rangle \in q$

such that

$$\mu: \langle v, g \rangle \ni \mu: \langle u, \lambda x. (p | fx) \rangle$$

Similar, and proof omitted.

Case 3.3 There exist some $\lambda: \langle u, f \rangle \in p$ (23)

and some $\bar{\lambda}: \langle u', f' \rangle \in q$ (24)

such that $\mu: \langle v, g \rangle \in fu' | f'u$. (25)

Now by (24)

$$\bar{\lambda}: \langle u', \lambda x. (f'x |_{j-1} r) \rangle \in q |_j r \quad (26)$$

and by (23) and (26)

$$fu' |_{i-1} (f'u |_{j-1} r) \subseteq p |_i (q |_j r) \quad (27)$$

and by induction hypothesis (5)

$$(fu' | f'u) | r \subseteq fu' |_{i-1} (f'u |_{j-1} r). \quad (28)$$

By (22) and (25)

$$gv' | g'v \subseteq (fu' | f'u) | r \quad (29)$$

and by (27), (28) and (29)

$$gv' | g'v \subseteq p |_i (q |_j r) \quad (30)$$

and by (20) and (30) $m \in p\,|_i(q\,|_j r)$ as required.

So (3) is proved, and Law (F2) is verified.

<u>Law (F3)</u> $p\backslash\alpha = p$ where $\alpha, \bar{\alpha} \notin L, p:L$

<u>Proof</u> We prove that

$$(p\backslash\alpha) \sqsubseteq p \quad \alpha, \bar{\alpha} \notin L \tag{1}$$

$$\text{and } (p\backslash\alpha) \sqsupseteq p \quad \alpha, \bar{\alpha} \notin L \tag{2}$$

We prove that $\forall p.(p\backslash_i\alpha) \sqsubseteq p$ for all $i \geq o$ (3)

such that $\alpha, \bar{\alpha} \notin L$. (1) then follows by computation induction.

<u>Basis</u> $(p\backslash_o\alpha) = \bot \sqsubseteq p$ as required.

<u>Induction step</u> Assume (3) for some $i \geq o$, and show for $i+1$.

$(p\backslash_{i+1}\alpha) = \{\!\!\{ \lambda:\langle u, \lambda v.(fv)\backslash_i\alpha\rangle \mid \lambda \in L-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle \in p \}\!\!\}$

by definition of $\backslash\alpha$

$= \{\!\!\{ \lambda:\langle u, \lambda v.(fv)\backslash_i\alpha\rangle \mid \lambda \in L, \lambda:\langle u,f\rangle \in p \}\!\!\}$

since $\alpha, \bar{\alpha} \notin L$

$\sqsubseteq \{\!\!\{ \lambda:\langle u, \lambda v.fv\rangle \mid \lambda \in L, \lambda:\langle u,f\rangle \in p \}\!\!\}$

by induction hypothesis and monotonicity

$= \{\!\!\{ \lambda:\langle u,f\rangle \mid \lambda \in L, \lambda:\langle u,f\rangle \in p \}\!\!\}$

since $\lambda v.fv = f$

$= p$, by definition of $\{\!\!\{-|-\}\!\!\}$, as required.

(2) may be proved by computation induction on the definition of the identity function $proc: L \to L$, where

$$proc(p) = \{\!\!\{ \lambda:\langle u, \lambda v.proc(fv)\rangle \mid \lambda \in L, \lambda:\langle u,f\rangle \in p \}\!\!\}$$

We prove $\forall p.(p\backslash\alpha) \sqsupseteq proc_i(p)$ for all $i \geq o$ (4)

where $\alpha, \bar{\alpha} \notin L$. (2) then follows by computation induction and that $p = proc(p)$.

<u>Basis</u> $proc_o(p) = \bot \sqsubseteq (p\backslash\alpha)$ as required

<u>Induction step</u> Assume (4) for some $i \geq o$ and show for $i+1$.

$proc_{i+1}(p) = \{\!|\lambda : \langle u, \lambda v.proc_i(fv)\rangle \mid \lambda \in L, \lambda : \langle u, f\rangle \in p |\!\}$

by definition of proc

$\sqsubseteq \{\!|\lambda : \langle u, \lambda v.(fv\backslash\alpha)\rangle \mid \lambda \in L, \lambda : \langle u, f\rangle \in p |\!\}$

by induction hypothesis and monotonicity

$= \{\!|\lambda : \langle u, \lambda v.(fv\backslash\alpha)\rangle \mid \lambda \in L - \{\alpha, \bar{\alpha}\}, \lambda : \langle u, f\rangle \in p |\!\}$

since $\alpha, \bar{\alpha} \notin L$

$= p\backslash\alpha$, by definition of $\backslash\alpha$, as required

Law (F3) follows from (1) and (2) .

<u>Law (F4)</u> $p\backslash\alpha\backslash\beta = p\backslash\beta\backslash\alpha$

<u>Proof</u> we show that

$\forall p. p\backslash\alpha\backslash_i\beta = p\backslash\beta\backslash_i\alpha$, for all $i \geq o$ (1)

(F4) follows by computation induction.

<u>Basis</u> $p\backslash\alpha\backslash_o\beta = \bot = p\backslash\beta\backslash_o\alpha$ as required

<u>Induction step</u> Assume (1) for some $i \geq o$ and show for $i+1$.

$p\backslash\beta\backslash_{i+1}\alpha = \{\!|\lambda : \langle u, \lambda v.(fv)\backslash\beta\backslash_i\alpha\rangle \mid \lambda \in L - \{\beta, \bar{\beta}, \alpha, \bar{\alpha}\}, \lambda : \langle u, f\rangle \in p |\!\}$

by first composition theorem and the definition of

$\backslash\beta$ and $\backslash\alpha$

$= \{\!|\lambda : \langle u, \lambda v.(fv)\backslash\alpha\backslash_i\beta\rangle \mid \lambda \in L - \{\beta, \bar{\beta}, \alpha, \bar{\alpha}\}, \lambda : \langle u, f\rangle \in p$

as required

$= p\backslash\alpha\backslash_{i+1}\beta$, by first composition theorem and the

definitions of $\backslash\alpha$ and $\backslash\beta$.

Law (F5) $(p_1|p_2)\backslash\alpha = (p_1\backslash\alpha)|(p_2\backslash\alpha)$
where $\{\alpha,\bar{\alpha}\}\cap L_1\cap\bar{L}_2 = \emptyset$

Proof we show that

$$\forall p_1,p_2.(p_1|_ip_2)\backslash\alpha = (p_1\backslash\alpha)|_i(p_2\backslash\alpha)\ \alpha,\bar{\alpha}\notin L_1\cap\bar{L}_2 \qquad (1)$$

for all $i\geq o$. (F5) follows by computation induction.

Basis $(p_1|_op_2)\backslash\alpha = \bot = (p_1\backslash\alpha)|_o(p_2\backslash\alpha)$ as required

Induction step Assume (1) for some $i\geq o$ and show for $i+1$.

$$(p_1|_{i+1}p_2)\backslash\alpha = \{\!\!\{\lambda:\langle u_1,\lambda v.f_1v|_ip_2\rangle \mid \lambda\in L_1, \lambda:\langle u_1,f_1\rangle\in p_1\}\!\!\}\backslash\alpha$$
$$\cup\{\!\!\{\lambda:\langle u_2,\lambda v.p_1|_if_2v\rangle \mid \lambda\in L_2, \lambda:\langle u_2,f_2\rangle\in p_2\}\!\!\}\backslash\alpha$$
$$\cup\bigcup\{\!\!\{f_1u_2|_if_2u_1 \mid \lambda\in L_1\wedge\bar{\lambda}\in L_2, \lambda:\langle u_1,f_1\rangle\in p_1,$$
$$\bar{\lambda}:\langle u_2,f_2\rangle\in p_2\}\!\!\}$$

by definition of $|$ and composition Lemma 4.4.7.

$$=\{\!\!\{\lambda:\langle u_1,\lambda v.(f_1v|_ip_2)\backslash\alpha\rangle \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}\wedge\lambda\in L_1, \lambda:\langle u_1,f_1\rangle\in p_1\}\!\!\}$$
$$\cup\{\!\!\{\lambda:\langle u_2,\lambda v.(p_1|_if_2v)\backslash\alpha\rangle \mid \lambda\in L_2-\{\alpha,\bar{\alpha}\}\wedge\lambda\in L_2, \lambda:\langle u_2,f_2\rangle\in p_2\}\!\!\}$$
$$\cup\bigcup\{\!\!\{(f_1u_2\ _if_2u_1)\backslash\alpha \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}\wedge\bar{\lambda}\ L_2-\{\alpha,\bar{\alpha}\},$$
$$\lambda:\langle u_1,f_1\rangle\in p_1, \bar{\lambda}:\langle u_2,f_2\rangle\in p_2\}\!\!\}$$

by composition theorems since $\alpha,\bar{\alpha}\notin L_1\cap\bar{L}_2$

$$=\{\!\!\{\lambda:\langle u_1,\lambda v.(f_1v\backslash\alpha)|_i(p_2\backslash\alpha)\rangle \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}, \lambda:\langle u_1,f_1\rangle\in p_1\}\!\!\}$$
$$\cup\{\!\!\{\lambda:\langle u_2,\lambda v.(p_1\backslash\alpha)|_i(f_2u\backslash\alpha)\rangle \mid \lambda\in L_2-\{\alpha,\bar{\alpha}\}, \lambda:\langle u_2,f_2\rangle\in p_2\}\!\!\}$$
$$\cup\bigcup\{\!\!\{(f_1u_2\backslash\alpha)|_i(f_2u_1\backslash\alpha) \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}\wedge\bar{\lambda}\in L_2-\{\alpha,\bar{\alpha}\},$$
$$\lambda:\langle u_1,f_1\rangle\in p_1, \bar{\lambda}:\langle u_2,f_2\rangle\in p_2\}\!\!\}$$

by induction hypothesis

$$=\{\!\!\{\lambda:\langle u_1,\lambda v.f_1'v|_i(p_2\backslash\alpha)\rangle \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}, \lambda:\langle u_1,f_1'\rangle\in p_1\backslash\alpha\}\!\!\}$$
$$\cup\{\!\!\{\lambda:\langle u_2,\lambda v.(p_1\backslash\alpha)|_if_2'v\rangle \mid \lambda\in L_2-\{\alpha,\bar{\alpha}\}, \lambda:\langle u_2,f_2'\rangle\in p_2\backslash\alpha\}\!\!\}$$
$$\cup\bigcup\{\!\!\{(f_1'u_2')|_i(f_2'u_1') \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}\wedge\bar{\lambda}\in L_2-\{\alpha,\bar{\alpha}\},$$
$$\lambda:\langle u_1',f_1'\rangle\in p_1\backslash\alpha, \bar{\lambda}:\langle u_2',f_2'\rangle\in p_2\backslash\alpha\}\!\!\}$$

by the definition of $\backslash\alpha$

$= (p_1 \backslash \alpha) |_{i+1} (p_2 \backslash \alpha)$ by definition of $|$, as required .

Law (F6) $E\backslash\alpha = E[\beta/\alpha]\backslash\beta$ where $\beta, \bar{\beta}$ do not occur in E

Proof We prove that the meta-syntactic substitution operation $[\beta/\alpha]$ over flow expressions is correctly interpreted by the semantic substitution operation $\{\beta/\alpha\}$ over processes. This proof proceeds in four parts;

(a) lemmas concerning $\{\beta/\alpha\}$,

(b) the $\{\ \}$ theorem that $p\backslash\alpha = (p\{\beta/\alpha\})\backslash\beta$,

(c) that $(E[\beta/\alpha])^P = E^P\{\beta/\alpha\}$,

(d) $(E\backslash\alpha)^P = (E[\beta/\alpha]\backslash\beta)^P$

(d) is in fact law (F6) for our process algebra P.

(a) Lemma 5.2.3 $(p_1|p_2)\{\beta/\alpha\} = p_1\{\beta/\alpha\} \mid p_2\{\beta/\alpha\}$ where $\beta, \bar{\beta} \notin L_1 \cup L_2$

Proof We show that $\forall p_1, p_2 . (p_1 |_i p_2)\{\beta/\alpha\} = p_1\{\beta/\alpha\} |_i p_2\{\beta/\alpha\}$ (1) for all $i \geq o$, where $\beta, \bar{\beta} \notin L_1 \cup L_2$. The lemma then follows by computation induction.

Basis $(p_1 |_o p_2)\{\beta/\alpha\} = \perp = p_1\{\beta/\alpha\} |_o p_2\{\beta/\alpha\}$ as required.

Induction step Assume (1) for some $i \geq o$ and show for $i+1$.

$(p_1 \;{}_{i+1} p_2)\{\beta/\alpha\} =$

$$\{\!\!\{ \lambda:\langle u, \lambda v.(p_1 \,|_i\, fv)\{\beta/\alpha\}\rangle \mid \lambda\in L_2-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \{\!\!\{ \lambda:\langle u, \lambda v.(fv \,|_i\, p_2)\{\beta/\alpha\}\rangle \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \beta:\langle u, \lambda v.(fv \,|_i\, p_2)\{\beta/\alpha\}\rangle \mid \alpha:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \bar{\beta}:\langle u, \lambda v.(fv \,|_i\, p_2)\{\beta/\alpha\}\rangle \mid \bar{\alpha}:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \beta:\langle u, \lambda v.(p_1 \,|_i\, fv)\{\beta/\alpha\}\rangle \mid \alpha:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \{\!\!\{ \bar{\beta}:\langle u, \lambda v.(p_1 \,|_i\, fv)\{\beta/\alpha\}\rangle \mid \bar{\alpha}:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ (f_1u_2 \,|_i\, f_2u_1)\{\beta/\alpha\} \mid \lambda\in L_1 \wedge \bar{\lambda}\in L_2, \lambda:\langle u_1,f_1\rangle\in p_1, \bar{\lambda}:\langle u_2,f_2\rangle\in p_2 \}\!\!\} \quad (2)$$

by definition of $|$, $\{\beta/\alpha\}$ and composition theorems.

Now $p_1\{\beta/\alpha\} \,|_{i+1}\, p_2\{\beta/\alpha\} =$

$$\{\!\!\{ \beta:\langle u, \lambda v.fv\{\beta/\alpha\} \,|_i\, p_2\{\beta/\alpha\}\rangle \mid \alpha:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \bar{\beta}:\langle u, \lambda v.fv\{\beta/\alpha\} \,|_i\, p_2\{\beta/\alpha\}\rangle \mid \bar{\alpha}:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \lambda:\langle u, \lambda v.fv\{\beta/\alpha\} \,|_i\, p_2\{\beta/\alpha\}\rangle \mid \lambda\in L_1-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p_1 \}\!\!\}$$
$$\cup \{\!\!\{ \beta:\langle u, \lambda v.p_1\{\beta/\alpha\} \,|_i\, fv\{\beta/\alpha\}\rangle \mid \alpha:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \{\!\!\{ \bar{\beta}:\langle u, \lambda v.p_1\{\beta/\alpha\} \,|_i\, fv\{\beta/\alpha\}\rangle \mid \bar{\alpha}:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \{\!\!\{ \lambda:\langle u, \lambda v.p_1\{\beta/\alpha\} \,|_i\, fv\{\beta/\alpha\}\rangle \mid \lambda\in L_2-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p_2 \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ (f_1u_2)\{\beta/\alpha\} \,|_i\, (f_2u_1)\{\beta/\alpha\} \mid \lambda\in L_1 \wedge \bar{\lambda}\in L_2, \bar{\lambda}:\langle u_2,f_2\rangle\in p_2, \lambda:\langle u_1,f_1\rangle\in p_1 \}\!\!\}$$

by definition of $\{\beta/\alpha\}$, $|$ and composition theorems as $\beta,\bar{\beta}\notin L_1\cup L_2$

= (2) by induction hypothesis, as required

<u>Lemma 5.2.4</u> $p\{\beta/\alpha\} = p$ where $\alpha,\bar{\alpha}\notin L$ and $p:L$

<u>Proof</u> we show that

$$\forall p.p\{\beta/\alpha\}_i = proc_i(p) \text{ for all } i\geq 0 \quad (1)$$

and the lemma follow by computation induction.

Basis $p\{\beta/\alpha\}_o = \bot = proc_o(p)$ as required

Induction step Assume (1) for some $i \geq o$ and show for $i+1$.

$$p\{\beta/\alpha\}_{i+1} = \{\!\{ \lambda:\langle u, \lambda v.(fv)\{\beta/\alpha\}_i\rangle \mid \lambda \in L-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle \in p \}\!\}$$

by definition of $\{\beta/\alpha\}$ since $\alpha,\bar{\alpha} \notin L$

$$= \{\!\{ \lambda:\langle u, \lambda v.proc_i(fv)\rangle \mid \lambda \in L, \lambda:\langle u,f\rangle \in p \}\!\}$$

since $L = L-\{\alpha,\bar{\alpha}\}$ and induction hypothesis

$= proc_{i+1}(p)$ as required, by definition of proc.

Lemma 5.2.5 $(p\backslash\gamma)\{\beta/\alpha\} = p\{\beta/\alpha\}\backslash\gamma$ where $\gamma \neq \alpha,\beta$

Proof we show that

$$(p\backslash_i\gamma)\{\beta/\alpha\} = p\{\beta/\alpha\}\backslash_i\gamma \text{ for all } i \geq o, \qquad (1)$$

for $\gamma \neq \alpha,\beta$. The lemma follows by induction hypothesis.

Basis $(p\backslash_o\gamma)\{\beta/\alpha\} = \bot = p\{\beta/\alpha\}\backslash_o\gamma$ $\gamma \neq \alpha,\beta$ as required

Induction step Assume that (1) holds for some $i \geq o$ and show for $i+1$.

$$\begin{aligned}(p\backslash_{i+1}\gamma)\{\beta/\alpha\} = {} & \{\!\{ \beta:\langle u, \lambda v.(fv\backslash_i\gamma)\{\beta/\alpha\}\rangle \mid \alpha:\langle u,f\rangle \in p \}\!\} \\ & \cup \{\!\{ \bar{\beta}:\langle u, \lambda v.(fv\backslash_i\gamma)\{\beta/\alpha\}\rangle \mid \bar{\alpha}:\langle u,f\rangle \in p \}\!\} \\ & \cup \{\!\{ \lambda:\langle u, \lambda v.(fv\backslash_i\gamma)\{\beta/\alpha\}\rangle \mid \lambda \in L-\{\gamma,\bar{\gamma},\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle \in p \}\!\} \qquad (2)\end{aligned}$$

by first composition theorem and definition of $\backslash\gamma$ and $\{\beta/\alpha\}$, since $\gamma \neq \alpha,\beta$.

$$\begin{aligned}\text{Now } p\{\beta/\alpha\}\backslash_{i+1}\gamma = {} & \{\!\{ \beta:\langle u, \lambda v.(fv)\{\beta/\alpha\}\backslash_i\gamma\rangle \mid \alpha:\langle u,f\rangle \in p \}\!\} \\ & \cup \{\!\{ \bar{\beta}:\langle u, \lambda v.(fv)\{\beta/\alpha\}\backslash_i\gamma\rangle \mid \bar{\alpha}:\langle u,f\rangle \in p \}\!\} \\ & \cup \{\!\{ \lambda:\langle u, \lambda v.(fv)\{\beta/\alpha\}\backslash_i\gamma\rangle \mid \lambda \in L-\{\alpha,\bar{\alpha},\gamma,\bar{\gamma}\}, \lambda:\langle u,f\rangle \in p \}\!\}\end{aligned}$$

by first composition theorem and the definition of $\backslash\gamma$ and $\{\beta/\alpha\}$, since $\gamma \neq \alpha,\beta$

= (2) as required, by induction hypothesis.

(b) <u>Theorem 5.2.6</u> $p\{\beta/\alpha\}\backslash\beta = p\backslash\alpha \quad \beta,\bar{\beta}\notin L$

<u>Proof</u> we show that

$\forall p. p\{\beta/\alpha\}\backslash_i\beta = p\backslash_i\alpha$ for all $i\geq o$ (1)

such that $\beta,\bar{\beta}\notin L$. Theorem follows by computation induction.

<u>Basis</u> $p\{\beta/\alpha\}\backslash_o\beta = \perp = p\backslash_o\alpha$

<u>Induction step</u> Assume that (1) holds for some $i\geq o$ and show for $i+1$.

$p\{\beta/\alpha\}\backslash_{i+1}\beta = \{\!\{ \lambda:\langle u,\lambda v.(fv)\{\beta/\alpha\}\backslash_i\beta\rangle \mid \lambda\in L-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p \}\!\}$

by the first composition theorem and the definition of $\{\beta/\alpha\}$ and $\backslash\beta$.

$= \{\!\{ \lambda:\langle u,\lambda v.(fv)\backslash_i\alpha\rangle \mid \lambda\in L-\{\alpha,\bar{\alpha}\}, \lambda:\langle u,f\rangle\in p \}\!\}$

by induction hypothesis, since $\beta,\bar{\beta}\notin L$.

$= p\backslash_{i+1}\alpha$ by definition of $\backslash\alpha$, as required .

(c) We now show by induction on the structure of E that

$(E[\beta/\alpha])^P = E^P\{\beta/\alpha\}$ where superfix P indicates the interpretation as processes.

<u>Basis</u> This is assumed due to consistency among the nullary operations $c_i\ \lambda_1\ldots\lambda_n$. Thus for $E = c_i\lambda_1\ldots\lambda_n$

$$((c_i\lambda_1\ldots\lambda_n)[\beta/\alpha])^P = p\{\beta/\alpha\}$$

and p is of sort $\{\lambda_1,\ldots,\lambda_n\}$ and is the process $(c_i\lambda_1\ldots\lambda_n)^P$.

<u>Induction step</u> either $E = E_1|E_2$ with $\beta,\bar{\beta}\notin E_1$ and E_2, or $E = E_1\backslash\gamma$ with $\beta,\bar{\beta}\notin E_1$ or $\gamma = \beta$.

<u>Case 1</u> $E = E_1 | E_2$ where $\beta, \bar{\beta} \notin E_1$ and E_2

$$
\begin{aligned}
(E[\beta/\alpha])^P &= ((E_1 | E_2)[\beta/\alpha])^P \\
&= (E_1[\beta/\alpha] \,|\, E_2[\beta/\alpha])^P \text{ by } [\beta/\alpha] \\
&= (E_1[\beta/\alpha])^P \,|^P\, (E_2[\beta/\alpha])^P \text{ where } |^P \text{ is a process operation} \\
&= (E_1^P\{\beta/\alpha\}) \,|^P\, (E_2^P\{\beta/\alpha\}) \text{ by induction hypothesis} \\
&= (E_1^P \,|^P\, E_2^P) \qquad \text{by Lemma 5.2.3} \\
&= E^P\{\beta/\alpha\} \text{ by our interpretation, as required.}
\end{aligned}
$$

<u>Case 2</u> $E = E_1 \backslash \alpha$ where $\beta, \bar{\beta} \notin E_1$. Either $\gamma \neq \beta$ and $\gamma = \alpha$ or $\gamma \neq \alpha, \beta$ or $\gamma = \beta$.

<u>Case 2.1</u> $\gamma \neq \beta$ and $\gamma = \alpha$

$$
\begin{aligned}
((E_1 \backslash \gamma)[\beta/\alpha])^P &= (E_1 \backslash \gamma)^P \\
&= E_1^P \backslash^P \gamma \text{ where } \backslash^P \gamma \text{ is now a process operation} \\
&= (E_1^P \backslash^P \gamma)\{\beta/\alpha\} \text{ by Lemma 5.2.4} \\
&= (E_1 \backslash \gamma)^P\{\beta/\alpha\} \text{ by } P\text{, as required}
\end{aligned}
$$

<u>Case 2.2</u> $\gamma \neq \alpha, \beta$

$$
\begin{aligned}
((E_1 \backslash \gamma)[\beta/\alpha])^P &= (E_1[\beta/\alpha] \backslash \gamma)^P \\
&= (E_1[\beta/\alpha])^P \backslash^P \gamma \quad \text{by } P \\
&= (E_1^P\{\beta/\alpha\}) \backslash^P \gamma \quad \text{by induction hypothesis} \\
&= (E_1^P \backslash^P \gamma)\{\beta/\alpha\} \text{ by Lemma 5.2.5} \\
&\quad \text{since } \gamma \neq \alpha \text{ and } \beta, \bar{\beta} \notin E_1 \\
&= (E_1 \backslash \gamma)^P\{\beta/\alpha\} \text{ as required}
\end{aligned}
$$

<u>Case 2.3</u> $\gamma = \beta$

$$
((E_1 \backslash \beta)[\beta/\alpha])^P = (E_1[\beta/\alpha])^P \text{ since } \beta, \bar{\beta} \notin E_1
$$

$= E_1^{\,P}\{\beta/\alpha\}$ by induction hypothesis

$= (E_1^{\,P}\backslash^P\beta)\{\beta/\alpha\}$ by Law (F3) since $\beta,\overline{\beta}\notin E_1$

$= (E_1\backslash\beta)^P\{\beta/\alpha\}$ as required.

Hence $(E[\beta/\alpha])^P = E^P\{\beta/\alpha\}$ by structural induction.

(d) $(E\backslash\alpha)^P = E^P\backslash\alpha$ where $\backslash\alpha$ is a process operation

$= (E^P\{\beta/\alpha\})\backslash\beta$ by (b)

$= (E[\beta/\alpha])^P\backslash\beta$ by (c)

$= (E[\beta/\alpha]\backslash\beta)^P$ as required.

Hence $E\backslash\alpha = E[\beta/\alpha]\backslash\beta$; Law (F6) when syntactic $[\beta/\alpha]$ is interpreted as $\{\beta/\alpha\}$.

Since we have all six laws in our process algebra, processes are a flow algebra.

CHAPTER 6

TERMINATING PROCESSES

Until now we have only considered processes which model agents that compute for an infinite amount of time. These processes are defined recursively and model such agents as operating systems, which should "operate" forever if written correctly and run on fault-proof hardware.

Some computing agents thus have infinite behaviour, but some others are intrinsically finite; such as terminating programs. Such termination must be capable of representation in our process model.

We introduce the notion of computation trees in this chapter to assist in the explanation of terminating processes.

6.1 Representing termination

We shall use a special label τ to indicate the final communication to be made by a process. A capability labelled by τ can be thought of as a communication with some external environment which will not be modelled. Thus all finite processes will have τ as a member of their sort, whilst $\bar{\tau}$ will never occur in the sort of a process. As τ is used only to indicate a final communication it will not communicate any significant values. A capability labelled by τ will therefore be a synchroniser.

Considering a process which only wishes to communicate once on the α line and then terminate, we have the process defined by

$$\llbracket \alpha : \langle u, \lambda v. \llbracket \tau : \langle o, K\phi \rangle \rrbracket \rangle \rrbracket$$

The renewal of this process is the set containing only the terminating capability :

$\Upsilon:\langle o,f\rangle$, where $f = K\emptyset$.

Since termination implies the inability to communicate further, it is natural to adopt $\emptyset$ as the renewal of a terminating capability. $\Upsilon:\langle o,f\rangle$ may be written as Υ.

We have the following definition for a process which always terminates.

Definition

A process $p:L$ is finite iff $\Upsilon \in L$ and $\forall m \in p$ either $m = \Upsilon$ or $m = \alpha:\langle u,f\rangle, \alpha \neq \Upsilon$ and fv is finite for every v.

We also have a definition for a process which sometimes terminates:

Definition

A process $p:L$ is terminable iff $\Upsilon \in L$ and $\exists m \in p$. either $m = \Upsilon$ or $m = \alpha:\langle u,f\rangle, \alpha \neq \Upsilon$ and fv is terminable for v.

These two definitions do not take into account the termination caused by a capability $\alpha:\langle u,f\rangle$ for which $\alpha \neq \Upsilon$ and $fv = \emptyset$ for some v, or if $p = \emptyset$. If $\emptyset$ never occurs except "within" Υ then

finite $p \Rightarrow$ terminable p and

terminable $p \not\Rightarrow$ finite p, directly from the definitions.

6.2 Computation trees

A commonly used method for informally describing non-deterministic computation is to consider this computation as being represented by a tree. A number of branches emanating from a given node will correspond to the number of choices the computation can make at this point. A

given computation sequence will be represented by one path through such a tree, the path being infinite if the corresponding computation sequence is infinite. Processes shall be represented by trees as follows:

Definition

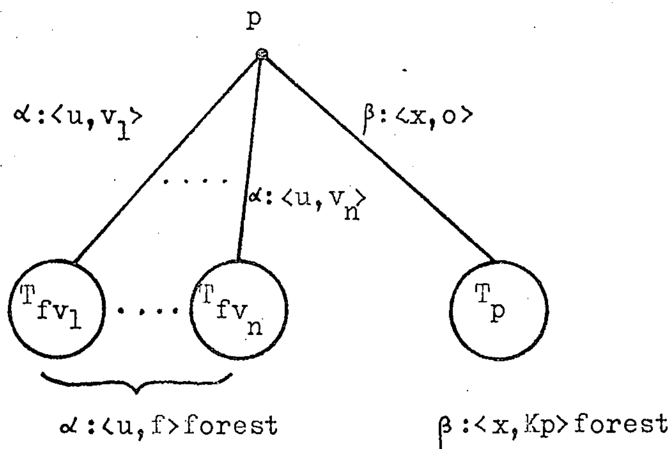
The computation tree T_p which corresponds to process p:L is defined by

1) the root of T_p is named by process p,
2) each subtree of T_p belongs to one forest of subtrees where
   2.1) each forest corresponds to one capability of p
   2.2) each tree T_{fv} in the forest which corresponds to capability α:<u,f>$\in$p, will itself correspond to renewal fv, for each $v \in V_\alpha$.
3) the arc from the root of T_p to the root of each subtree T_{fv} is named by the triple α:<u,v>, when T_{fv} is in that forest corresponding to α:<u,f>$\in$p, and $v \in V_\alpha$.

This definition allows for an infinite number of arcs out of any node. The first component of an arc name is the arc label.

As an example, let the process p be defined by

p:{α,β} = {α:<u,f>,β:<x,Kp>}, and the corresponding computation tree will be T_p =



where $T_{fv_1}, \ldots, T_{fv_n}$ will be a possibly infinite number of subtrees, one for each $v_i \in V_\alpha$. As $V_\beta = 1$ then our forest corresponding to capability $\beta:\langle x, Kp\rangle$ will consist only of the single tree T_p. The subtrees $T_{fv_1}, \ldots, T_{fv_n}$ may have all the paths emanating from them being of finite length, but since T_p is defined recursively we shall have at least one infinite path in this example. In the above tree T_p, we use the notation (T_m) to represent subtree T_m.

For a finite process each path of its corresponding computation tree will end with an arc named by $\tau:\langle o,o\rangle$, leading to the node named by ϕ which has no arcs emanating from it.

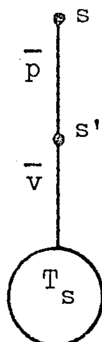
The tree corresponding to a terminable process will have at least one path which ends with an arc named by $\tau:\langle o,o\rangle$ leading to the node named by ϕ with out-degree zero.

For pure synchronisation process s:L (where $U_\lambda = V_\lambda = \{o\}$ for each $\lambda \in L$), then it is sufficient to name the arcs of the computation tree T_s by the labels alone. Each forest corresponding to a capability of s will then consist of only one tree. For example consider the process $s:\{\overline{p}, \overline{v}\}$ which models a binary semaphore and is defined by

$$s = \{\!\{ \overline{p}:\langle o, Ks'\rangle \}\!\}$$

where $s' = \{\!\{ \overline{v}:\langle o, Ks\rangle \}\!\}$. All the capabilities of s and its renewals are synchronisers. The computation tree corresponding to s is then:

T_s =



Trees corresponding to pure synchronisation processes will be known as synchronisation trees.

Computation trees may form a flow algebra with a homomorphism from the algebra of processes to the algebra of such trees. This tree algebra will not be considered but computation trees will be used when discussing sequences of communications and also in our scheduling technique.

We can consider a computation tree as producing a set of words which are those sequences of arc labels which we meet as we progress down any path of the tree from the root. These possibly infinite words constitute the language of a tree, and we may denote the language of tree T by $\mathcal{L}(T)$.

The language of a tree T_p will consist only of words of finite length if p is a finite process; and will contain some words of finite length if p is a terminable process. We may then define an infinite process by:

Definition

A process p is infinite iff $\mathcal{L}(T_p)$ is a set of words of infinite length. We may note that infinite p $\not\Leftrightarrow$ ($\neg$finite p)$\wedge$($\neg$terminable p). This follows from section 6.5.

6.3 Combinators for terminating processes

The $|$ and $\|$ combinators are inadequate when used with finite or terminable processes. When two agents compute concurrently we require that if one agent terminates before the other all remaining computation is due to the later. Computation of the two agents consequently ceases only when both agents have finished computing.

We define new $|$ and $\|$ combinators which produce terminating

capabilities only when both processes contain terminating capabilities. Should one of the processes be infinite the resulting process is infinite and no terminating capabilities remain as members or members of renewals, although Υ will be a member of the resulting sort.

Definition

$\not| : L_1 \times L_2 \to L_1 \cup L_2$, $\Upsilon_i, \overline{\Upsilon}_i \notin L_1 \cup L_2, \forall_i \geq o$ is defined by

$p \not| q = (p\{\Upsilon_i/\Upsilon\} | q\{\Upsilon_j/\Upsilon\} | J_{ij}) \backslash \Upsilon_i \backslash \Upsilon_j$, for any $i,j \geq o$ such that $i \neq j$.

The "join" process J_{ij} is defined by

$J_{ij} = \{\!\{ \overline{\Upsilon}_i : \langle o, K\{\!\{\overline{\Upsilon}_j : \langle o, K\{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle,$
$\overline{\Upsilon}_j : \langle o, K\{\!\{\overline{\Upsilon}_i : \langle o, K\{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle \}\!\}$

is the substitution operation defined in the previous chapter.

The join process essentially accepts renamed terminating communications labelled by $\overline{\Upsilon}_i$ and $\overline{\Upsilon}_j$ and produces a termination capability only when both the processes p and q have communicated along the Υ_i and Υ_j lines. These communication lines are then removed.

We define $\not\|$ analogously with the definition of $\|$.

Definition

$\not\| : L_1 \times L_2 \to L_1 \cup L_2 - (L_1 \circ L_2)$ is defined by

$p \not\| q = (p \not| q) \backslash \alpha_1 \ldots \backslash \alpha_n$ where $L_1 \circ L_2 = \{\alpha_1, \overline{\alpha}_1, \ldots \alpha_n, \overline{\alpha}_n\}$

We have the following lemma, due to this definition:

Lemma 6.3.1

$p \not\| q = ((p\{\Upsilon_i/\Upsilon\} \| q\{\Upsilon_j/\Upsilon\}) | J_{ij}) \backslash \Upsilon_i \backslash \Upsilon_j$

for any $i,j \geq o$ such that $i \neq j$

Proof

$p \nparallel q \;=\; (p \nmid q)\backslash\alpha_1 \dots \backslash\alpha_n$ where $L_1 . L_2 = \{\alpha_1, \bar{\alpha}_1, \dots \alpha_n, \bar{\alpha}_n\}$

by definition of $\nparallel$

$= \left[(p\{\tau_i/\tau\} \mid q\{\tau_j/\tau\} \mid J_{ij})\backslash\tau_i\backslash\tau_j\right]\backslash\alpha_1 \dots \backslash\alpha_n$

by definition of $\nmid$

$= \left[(p\{\tau_i/\tau\} \mid q\{\tau_j/\tau\} \mid J_{ij})\backslash\alpha_1 \dots \backslash\alpha_n\right]\backslash\tau_i\backslash\tau_j$

by Law (F4)

$= \left[(p\{\tau_i/\tau\} \mid q\{\tau_j/\tau\})\backslash\alpha_1 \dots \backslash\alpha_n \mid J_{ij}\right]\backslash\tau_i\backslash\tau_j$

by Laws (F3) and (F5)

$= \left[(p\{\tau_i/\tau\} \parallel q\{\tau_j/\tau\}) \mid J_{ij}\right]\backslash\tau_i\backslash\tau_j$

by definition of $\parallel$.

A property of $\nmid$ and $\nparallel$ is that they behave as $\mid$ and $\parallel$ in the absence of terminating capabilities .

Lemma 6.3.2

For $\mid : L_1 \times L_2 \rightarrow L_1 \cup L_2$ and $\nmid : L_1 \times L_2 \rightarrow L_1 \cup L_2$ and $\tau \notin L_1 \cup L_2$ then $\nmid \equiv \mid$

Proof We show that $\forall p : L_1, q : L_2 . p \nmid q = p \mid q$.

$p \nmid q \;=\; (p\{\tau_1/\tau\} \mid q\{\tau_2/\tau\} \mid J_{12})\backslash\tau_1\backslash\tau_2$

by definition of $\nmid$

$= (p \mid q \mid J_{12})\backslash\tau_1\backslash\tau_2$

by Lemma 5.2.4 and $\tau \notin L_1 \cup L_2$

$= ((p \mid q)\backslash\tau_1\backslash\tau_2) \mid (J_{12})\backslash\tau_1\backslash\tau_2$

by Law (F5) as $\tau_1, \bar{\tau}_1 \notin L_1$ and $\tau_2, \bar{\tau}_2 \notin L_2$ by definition of $\nmid$

$= (p \mid q) \mid (J_{12})\backslash\tau_1\backslash\tau_2$

by Law (F3)

$= p \mid q$ by definition of $\backslash\alpha$ and J_{12} .

Lemma 6.3.3

For $\| : L_1 \times L_2 \rightarrow L_1 \cup L_2 - (L_1 \cdot L_2)$ and $\not\| : L_1 \times L_2 \rightarrow L_1 \cup L_2 - (L_1 \cdot L_2)$ and if $\tau \notin L_1 \cup L_2$ then $\| \equiv \not\|$

Proof We show that $\forall p:L_1, q:L_2 \cdot p \| q = p \not\| q$

$$
\begin{aligned}
p \not\| q &= (p \not| q) \backslash \alpha_1 \ldots \backslash \alpha_n \quad \text{where } L_1 \cdot L_2 = \{\alpha_1, \bar{\alpha}_1, \ldots, \alpha_n, \bar{\alpha}_n\} \\
&\quad \text{by definition of } \not\| \\
&= (p|q) \backslash \alpha_1 \ldots \backslash \alpha_n \\
&\quad \text{by Lemma 6.3.2 as } \tau \notin L_1 \cup L_2 \\
&= p \| q \text{ by definition of } \| .
\end{aligned}
$$

6.4 Properties of termination combinators

To enable $\not|$ and $\not\|$ to be used freely, certain properties are necessary.

Theorem 6.4.1 (associativity of $\not|$)

$$p \not| (q \not| r) = (p \not| q) \not| r$$

The following lemmas are necessary in the proof of this theorem.

Lemma 6.4.2

$$
\begin{aligned}
&(p\{\tau_m/\tau\} | q\{\tau_n/\tau\} | J_{mn}) \backslash \tau_m \backslash \tau_n \\
&= (p\{\tau_i/\tau\} | q\{\tau_j/\tau\} | J_{ij}) \backslash \tau_i \backslash \tau_j \\
&\quad \forall i,j,m,n \text{ where } i \neq j \text{ and } m \neq n.
\end{aligned}
$$

Proof Immediate from Law (F6), the definition of $\{\alpha/\beta\}$ and the definition of the join process.

Lemma 6.4.3

$$(J_{12}\{\tau_3/\tau\}|J_{34})\backslash\tau_3 = (J_{24}\{\tau_3/\tau\}|J_{13})\backslash\tau_3$$

Proof $(J_{12}\{\tau_3/\tau\}|J_{34})\backslash\tau_3 =$

$$\{\!\{\ \bar{\tau}_2:\langle o,K\{\!\{\bar{\tau}_4:\langle o,K\{\!\{\bar{\tau}_1:\langle o,K\ \{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle,$$
$$\bar{\tau}_1:\langle o,K\{\!\{\bar{\tau}_2:\langle o,K\{\!\{\bar{\tau}_4:\langle o,K\ \{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle,$$
$$\bar{\tau}_2:\langle o,K\{\!\{\bar{\tau}_1:\langle o,K\{\!\{\bar{\tau}_4:\langle o,K\ \{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle,$$
$$\bar{\tau}_4:\langle o,K\ \{\!\{\bar{\tau}_1:\langle o,K\{\!\{\bar{\tau}_2:\langle o,K\{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle,$$
$$\bar{\tau}_4:\langle o,K\{\!\{\bar{\tau}_2:\langle o,K\{\!\{\bar{\tau}_1:\langle o,K\ \{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle,$$
$$\bar{\tau}_1:\langle o,K\{\!\{\bar{\tau}_4:\langle o,K\{\!\{\bar{\tau}_2:\langle o,K\ \{\!\{\Upsilon\}\!\}\rangle\}\!\}\rangle\}\!\}\rangle\}\!\}$$

by definition of $|$, $\backslash\tau_3$ and the join process

$= (J_{24}\{\tau_3/\tau\}|J_{13})\backslash\tau_3$ again by $|$,$\backslash\tau_3$ and the join process

Proof of theorem 6.4.1

$p\nmid(q\nmid r) =$

$$(p\{\tau_5/\tau\}|(q\{\tau_7/\tau\}|r\{\tau_8/\tau\}|J_{78})\backslash\tau_7\backslash\tau_8)\{\tau_6/\tau\}|J_{56})\backslash\tau_5\backslash\tau_6$$

by the definition of $\nmid$ using Lemma 6.4.2

$= s\backslash\tau_5\backslash\tau_6\backslash\tau_7\backslash\tau_8$

where $s = (p\{\tau_5/\tau\}|q\{\tau_7/\tau\}|r\{\tau_8/\tau\}|J_{78}\{\tau_6/\tau\}|J_{56}$

by Laws (F1), (F2), (F3), (F4), (F5).

$= (s\{\tau_1/\tau_5\}\{\tau_2/\tau_7\}\{\tau_3/\tau_6\}\{\tau_4/\tau_8\})\backslash\tau_1\backslash\tau_2\backslash\tau_3\backslash\tau_4$

by Laws (F4) and (F6)

$= (p\{\tau_1/\tau\}|q\{\tau_2/\tau\}|r\{\tau_4/\tau\}|J_{24}\{\tau_3/\tau\}|J_{13})\backslash\tau_1\backslash\tau_2\backslash\tau_3\backslash\tau_4$ (*)

by definition of $\{\alpha/\beta\}$ and Lemma 5.2.3

Now

$(p \nmid q) \nmid r =$

$$(p\{\tau_1/\tau\}|q\{\tau_2/\tau\}|J_{12})\backslash\tau_1\backslash\tau_2)\{\tau_3/\tau\}|r\{\tau_4/\tau\}|J_{34})\backslash\tau_3\backslash\tau_4$$

by definition of $\nmid$ and Lemma 6.4.2

$$= (p\{\tau_1/\tau\}|q\{\tau_2/\tau\}|r\{\tau_4/\tau\}|J_{12}\{\tau_3/\tau\}|J_{34})\backslash\tau_1\backslash\tau_2\backslash\tau_3\backslash\tau_4$$

by Laws (F1), (F2), (F3), (F4), (F5)

$$= ((p\{\tau_1/\tau\}|q\{\tau_2/\tau\}|r\{\tau_4/\tau\})\backslash\tau_3|(J_{12}\{\tau_3/\tau\}|J_{34})\backslash\tau_3)\backslash\tau_1\backslash\tau_2\backslash\tau_4$$

by Law (F5)

$$= ((p\{\tau_1/\tau\}|q\{\tau_2/\tau\}|r\{\tau_4/\tau\})\backslash\tau_3|(J_{24}\{\tau_3/\tau\}|J_{13})\backslash\tau_3)\backslash\tau_1\backslash\tau_2\backslash\tau_4$$

by Lemma 6.4.3

$= (*)$ by Law (F5)

$= p \nmid (q \nmid r)$

<u>Theorem 6.4.4 (commutivity of $\nmid$)</u>

$p \nmid q = q \nmid p$

<u>Proof</u> Immediate from the definition of $\nmid$ and Law (F1).

<u>Theorem 6.4.5 (associativity of $\nparallel$)</u> For $p:L_1, q:L_2, r:L_3$

$p \nparallel (q \nparallel r) = (p \nparallel q) \nparallel r$

where $(L_1 \cap L_2 \cap \overline{L}_3) \cup (L_1 \cap \overline{L}_2 \cap L_3) \cup (\overline{L}_1 \cap L_2 \cap L_3) = \emptyset$

<u>Theorem 6.4.6 (commutivity of $\nparallel$)</u>

$p \nparallel q = q \nparallel p$

These theorems follow immediately from the definition of $\nparallel$ and the commutivity and associativity of $\nmid$, in a manner identical to that used to prove the commutivity and associativity of $\|$ in the appendix.

We do not examine properties of these combinators further but we expect certain properties such as

$p \not\| q$ is finite $\Longleftrightarrow$ p is finite <u>and</u> q is finite to hold.

6.5 Termination and deadlock

In our definition of finite and terminable processes we do not consider the termination which is caused by such capabilities as $\alpha:\langle u,f\rangle$, where $\alpha \neq \mho$ and $fv = \emptyset$, for some v. But capabilities similar to this do appear in our model, usually due to the $\|$ and $\not\|$ combinators.

If a process p (or its renewals) contains such a capability then $\mathcal{L}(T_p)$ will contain a finite word whose right-most letter is α. By our definitions in section 6.1, p will neither be finite nor terminable. We say that p may deadlock.

Definition

p <u>may deadlock</u> iff $\mathcal{L}(T_p)$ contains a finite word whose right-most letter is not $\mho$. We now have that

$(\neg\text{infinite } p) \Longleftrightarrow (\text{finite } p) \vee (\text{terminable } p) \vee (p \text{ may deadlock})$.

To see how processes which may deadlock arise, consider the following example.

Suppose $p:\{\alpha,\bar{\beta}\}$ and $q:\{\beta,\bar{\alpha},\gamma\}$ where

$p = \{\!\{\beta:\langle u,f\rangle\}\!\}$

$q = \{\!\{\alpha:\langle v,g\rangle\}\!\}$

then $p \| q = \emptyset$, by definition of $\|$.

Intuitively, p and q can only communicate on their interconnecting communication lines; but p wishes to communicate on the β line and q on the α line. These two processes <u>try</u> to communicate with each other but fail due to the inherent synchronisation of our $|$ and $\|$ combinators. The outcome of the attempted communication between p and q is then the empty set; this is another form of termination. This termination differs from the explicit termination explained above, and is due to some required communication failing to take place. This may be thought of as corresponding to <u>deadlock</u>, as used in the operating systems sense. If a number of communicating processes terminate in this fashion, then the agents which they model can be said to deadlock, and the processes themselves may be considered to have deadlocked.

Deadlock in the operating systems sense can occur when we have two agents M and N trying to access some shared resource R. If M waits for N to access the resource first and N waits for M to access the resource first, deadlock occurs. Let M, N and R be modelled by the processes m, n and r respectively. As an example let these processes be defined as follows:

$$m: \{\alpha_1, \beta_1, \gamma_1\} = \{\!\{\beta_1 : \langle u, \lambda v. m\rangle\}\!\}$$

$$n: \{\beta_2, \gamma_2, \delta\} = \{\!\{\beta_2 : \langle x, \lambda y. n\rangle\}\!\}$$

$$r: \{\bar{\beta}_1, \bar{\gamma}_1, \bar{\beta}_2\gamma_2\} = \{\!\{\bar{\gamma}_2 : \langle u', \lambda v'. \{\!\{\bar{\beta}_1 : \langle v, \lambda v. r\rangle\}\!\}\rangle, \bar{\gamma}_1 : \langle x', \lambda y'. \{\!\{\bar{\beta}_2 : \langle y, \lambda x. r\rangle\}\!\}\rangle\}\!\}$$

Now $m \| n \| r = m \| (n \| r)$ by associativity of $\|$, as the sorts involved satisfy the restriction that makes $\|$ associative. As $n \| r : \{\bar{\beta}_1, \bar{\gamma}_1, \delta\} = \{\!\{\bar{\gamma}_1 : \langle x', \lambda y'. \emptyset\rangle\}\!\}$ then $m \| n \| r = \emptyset$; and our processes have deadlocked.

If $p \| q = \emptyset$, then p and q may not immediately deadlock but may

communicate first. For instance, if $p:\{\alpha,\beta\}$ and $q:\{\bar{\alpha},\bar{\beta}\}$ are defined by:

$$p = \{\!\{\alpha:\langle u, \lambda v.a\rangle\}\!\}$$
$$q = \{\!\{\bar{\alpha}:\langle u', \lambda v'.b\rangle\}\!\}$$

then $p \| q = a \| b$ by definition of $\|$ and $a \| b = \phi$.

Unfortunately the Smyth ordering which we use in our model prevents us from modelling deadlock properly. Suppose that we have two processes $p:\{\gamma,\alpha,\beta\}$ and $q:\{\bar{\alpha},\bar{\beta}\}$ defined by:

$$p = \{\!\{\gamma:\langle u, \lambda v.\{\!\{\alpha:\langle u, \lambda v.p\rangle\}\!\}\rangle, \beta:\langle x, \lambda y.p\rangle\}\!\}$$
$$q = \{\!\{\bar{\alpha}:\langle u, \lambda u.q\rangle\}\!\}$$

and $p \| q \ = \ \{\!\{\gamma:\langle u, \lambda v.\{\!\{\alpha:\langle u, \lambda v.p\rangle\}\!\} \| \{\!\{\bar{\alpha}:\langle v, \lambda u.q\rangle\}\!\}\rangle\}\!\}$

by the definition of $\|$

$= \ \{\!\{\gamma:\langle u, \lambda v.p \| q\rangle\}\!\}$

again by the definition of $\|$.

But the agents modelled by q and q may become deadlocked since

$$\{\!\{\beta:\langle x, \lambda y.p\rangle\}\!\} \| \{\!\{\bar{\alpha}:\langle v, \lambda u.q\rangle\}\!\} = \phi$$

where $\{\!\{\beta:\langle x, \lambda y.p\rangle\}\!\} \subseteq p$. But for any process m, $m \equiv m \cup \phi$, where $\equiv$ is the equivalence induced by the Smyth ordering due to ϕ being treated as "top" in our cpo. This identification loses us the information that $p \| q$ may become deadlocked, hence $p \| q$ does <u>not</u> faithfully model the behaviour of those two agents which may become deadlocked. This deficiency of our model means that we cannot tell when $p \| q$ <u>may</u> deadlock, although we can tell when $p \| q$ <u>always</u> deadlocks; namely $p \| q = \phi$.

This shortcoming of the Smyth ordering in failing to let us model deadlock properly is a major fault of our model. It may therefore be necessary to use the Milner ordering in our model if we wish deadlock to be dealt with correctly, although ϕ does not "fit into" this ordering as the maximum element as it does with

Smyth's. But there are advantages to be had with the Smyth ordering and these have been mentioned in Chapter 3.

The property that for any process p, then $p \cup \phi = p$, which causes the deficiency of our model mentioned above, is found to be useful in producing a scheduling technique, which is described in Chapter 7.

CHAPTER 7

SCHEDULING PROCESSES

Scheduling the actions of a number of concurrent computing agents according to some algorithm is a commonly occurring feature in the treatment of concurrency.

If processes are to model concurrent computation adequately we must be able to represent this scheduling of actions; but for processes to be a good model of concurrent computation one requirement is that scheduling must be dealt with in a uniform and intuitive manner.

As we model concurrent computation by synchronised communication, to prevent unrestricted concurrency among a system of process we are required to control the order in which these processes communicate. The implicit synchronisation present during communication (due to our $|$ and $\|$ combinators) is not always adequate for this purpose. In our treatment of scheduling among a system of processes we utilise a special process known as a scheduler. A scheduler and its renewals contain only pure synchronisation capabilities which we call synchronisers. The scheduler specifies in what order communication takes place both among the processes in the system and between these processes and the outside world.

7.1 The scheduling technique

Given a number of processes $p_1,\ldots,p_n$ we may wish them to communicate in some well-defined manner specified by a scheduling algorithm. Special synchronisers known as scheduling synchronisers are added to $p_1,\ldots,p_n$ and are used to communicate with our

scheduler (which models the scheduling algorithm) via the $\|$ or $\nparallel$ combinators. Schedulers and their renewals consist only of scheduling synchronisers. Scheduling synchronisers are identified by being labelled by scheduling labels, which are distinct from the usual ones.

Our scheduling technique for $p_1,\ldots,p_n$ given a scheduler $s:\overline{S}$ is as follows:

(a) add scheduling synchronisers labelled by members of S to $p_1,\ldots,p_n$ (according to some rules which are given later) to get processes $q_1,\ldots,q_n$;

(b) let $q_1,\ldots,q_n$ communicate among themselves via $|$, $\|$, $\nmid$ or $\nparallel$, just as $p_1,\ldots,p_n$ would do if unrestricted concurrency were to be allowed, to get the process SCHEDULE;

(c) let SCHEDULE communicate with scheduler s by using the $\|$ combinator if $\tau \notin$ sort of SCHEDULE, or $\nparallel$ otherwise.

The scheduling synchronisers added by (a) are used for one of two reasons; either (i) they are used by s to control entry to the renewal of that capability now labelled by a scheduler label - this is guarding, or (ii) they are used to indicate to s when certain communications, with renewals labelled by a scheduler label, have taken place - this is monitoring.

Assume that $p_1,\ldots,p_n$ communicate among themselves using $|$, $\|$, $\nmid$ or $\nparallel$ to produce the process UNRESTRICT:L which allows unrestricted concurrency. The process SCHEDULE will then be of sort $L \cup S$ and $\|$ (or $\nparallel$ if $\tau \in L$) is used to let SCHEDULE communicate with s along these scheduling lines to get the process

$$\text{RESTRICT}:L \quad = \quad \text{SCHEDULE} \| S$$

The scheduling synchronisers added in (a) are thus removed by (c) giving us a process of the same sort as UNRESTRICT.

This scheduling technique therefore filters-out a number of the capabilities (and capabilities of renewals) of the process UNRESTRICT which results from the unrestricted concurrency among the system of process $p_1,\ldots,p_n$. We have then that

$$\mathcal{L}(T_{RESTRICT}) \subseteq \mathrm{TERM}\left(\mathcal{L}(T_{UNRESTRICT})\right)$$, where TERM is defined below.

$\mathcal{L}(T_{SCHEDULE})$ will differ from $\mathcal{L}(T_{UNRESTRICT})$ only in that it will have certain scheduling labels distributed among the other labels that go to make up the words in the language. If we consider a set of words identical to $\mathcal{L}(T_{SCHEDULE})$ but with all non-scheduling labels removed, then scheduler s can be said to permit only certain words in this set. We have the following definitions:

Definition

A scheduler $s:\bar{S}$ permits an infinite word ω iff $\bar{\omega}\in\mathcal{L}(T_s)$, where $\overline{a\omega_1} = \bar{a}\bar{\omega}_1$.

Using ε for the empty word we have the following definition for finite words.

Definition

A scheduler $s:\bar{S}$ permits a finite word σ iff $\bar{\sigma}\in\mathrm{TERM}(\mathcal{L}(T_s))$, where $\overline{a\sigma_1} = \bar{a}\bar{\sigma}_1$, $\bar{\varepsilon} = \varepsilon$ and TERM is defined by

$$\mathrm{TERM}(m) = \{a, ab, abc, \ldots \mid abcd\ldots \in m\}$$

Thus a scheduler permits an infinite word if it has an identical, but complemented word in the language of its computation tree. A scheduler permits a finite word σ if $\bar{\sigma}$ is an initial segment of a word in the language of the tree.

The following definition on words removes labels belonging to a certain set.

Definition

$$\mathrm{REMOVE}_B(a\omega) = \mathrm{REMOVE}_B(\omega) \text{ if } a \in B$$
$$= a\mathrm{REMOVE}_B(\omega) \text{ if } a \notin B$$
$$\mathrm{REMOVE}_B(a\sigma) = \mathrm{REMOVE}_B(\sigma) \text{ if } a \in B$$
$$= a\mathrm{REMOVE}_B(\sigma) \text{ if } a \notin B$$
$$\mathrm{REMOVE}_B(\varepsilon) = \varepsilon$$

Our scheduling technique is such that:

$$\mathcal{L}(T_{RESTRICT}) = \{\mathrm{REMOVE}_S(\phi) \mid \phi \in \mathcal{L}(T_{SCHEDULE}) \wedge s:\overline{S} \text{ permits } \mathrm{REMOVE}_{L-S}(\phi)\}$$

where UNRESTRICT is of sort L.

As a scheduler is used to specify that set of words to be permitted we may also think of it as representing the behaviour of some automaton which accepts the same set. The well-understood notion of accepting automaton can then be used to produce our schedulers.

7.2 Requirements for a scheduler

Greif [Gre 2] views scheduling of a number of concurrent computing agents in a somewhat similar manner to that which we adopt. She considers only partial orders on events; while we control only certain communications of our processes. She also mentions the requirement for the histories of events to determine the next events; while we require that the communication between the scheduler and the system of processes is dependent on previous

communication with the scheduler.

Greif also indicates the need for a method of scheduling where the "correctness" of the scheduler is immediately apparent with respect to some informal scheduling description, due to the model being "consistent" with reality.

This we would argue, is one of the desirable features of the process model; that our scheduler can be abstracted out from the rest of the model as a process and so be produced from some less-formal scheduling algorithm. The question of correctness (in some sense) of a scheduler process will not therefore arise; the scheduler will be taken to define the scheduling properties directly.

Analogously, we may consider the context-free language defined by production $S \rightarrow \varepsilon \mid oSo \mid 1S1$. This production defines the set of palindromes of even length over $\{o,1\}$, and is probably as good a definition of palindrome as the less formal definition "a string of even length unchanged by reversal".

Two necessary properties of schedulers appear to be that current scheduling depends on previous scheduling, and that some method of counting is necessary to control a number of similar processes.

This first requirement is easily seen to be necessary as our scheduler will control the orders of communication among a number of processes, some communication being dependent on previous communication. Our process model has been designed with this "order of events" in mind.

The second requirement is needed when we have several occurrences of the same process among those which we wish to schedule. For instance, we may require a scheduler to control a number of identical processes accessing some shared resource process.

We must therefore produce schedulers parameterised on some local store which can be used to hold a count of the number of processes accessing the resource. The semaphore example of Chapter 2 is an example of this for a single integer count. But integer counters may not be adequate for certain schedulers and other data structures such as stacks and queues may need to be used. The choice of structure would be made to allow for the best mathematical presentation of our scheduler and not for its implementation.

We now show how accepting automata are used when producing schedulers.

7.3 Acceptors

Given a number of processes which we wish to schedule, together with some scheduling algorithm or rule which defines the order in which certain communications involving these processes may take place, we can define an automaton, known as an acceptor. An acceptor "accepts" those possibly infinite words which form "correct" sequences of communication labels according to the scheduling rules. The behaviour of this acceptor can be represented by a scheduler.

An acceptor consists of an infinite input tape, a finite state control and an auxiliary memory. An input head reads the input tape from left to right and the memory can be any type of data structure and will be initialised before the acceptance procedure begins. We assume a finite memory alphabet and that at any point in time we can finitely describe the contents and structure of memory, though this memory may be arbitrarly large.

The type of memory determines the name of the acceptor, for instance a pushdown stack acceptor. The finite state control dictates the behaviour of our acceptor and so determines our scheduler. The control is a finite set of states together with a function of how these states are transformed for given tape values and contents of memory. It is adequate for our purposes to have our automata deterministic and so our state-transformation function will be single-valued. Note that this function δ will be partial, i.e. it is not defined for certain arguments.

Definition

An acceptor automaton (or acceptor) is a 6-tuple $A = (Q,\Sigma,M,\delta,q_o,m_o)$ where Q is a finite set of states with q_o the initial state; Σ is a finite input or tape alphabet; M is some memory set with m_o the initial memory; δ is a partial function from $Q \times M \times \Sigma$ to $Q \times M$.

Definition

A configuration of an acceptor A is a triple (q,m,ω) in $Q \times M \times \Sigma^{\infty}$ for infinite input tape, or (q,m,σ) in $Q \times M \times \Sigma^*$ for finite length input tape.

Definition

A move of acceptor A is represented by the symbol $\vdash$ between configurations and is defined by:

$(q,m,a\omega) \vdash (q',m',\omega)$ where $\delta(q,m,a) = (q',m')$ for $a \in \Sigma$ and $\omega \in \Sigma^{\infty}$ and

$(q,m,a\sigma) \vdash (q',m',\sigma)$ where $\delta(q,m,a) = (q',m')$ for $a \in \Sigma$ and $\sigma \in \Sigma^*$.

For a configuration $(q,m,a\omega)$ or $(q,m,a\sigma)$ δ may be undefined for argument (q,m,a) as it is a partial function. No move is then possible and our acceptor halts. The words $a\omega$ or $a\sigma$ are not accepted in this case.

We have the following definitions for acceptance:

Definition (infinite)

For an acceptor A, A accepts ω from q,m iff there is an infinite sequence of moves

$$(q,m,\omega) \vdash (q_1,m_1,\omega_1) \vdash \ldots \vdash (q_n,m_n,\omega_n) \vdash \ldots$$

Definition (finite)

For acceptor A, A accepts σ iff there is a finite sequence of moves $(q,m,\sigma\sigma') \vdash \ldots \vdash (q_n,m_n,\sigma')$ where $n = |\sigma|$, and σ' is some finite or infinite word, possibly ε.

In the definitions of acceptance of finite and infinite words q and m need not necessarily be initial. These definitions cause finite left-most portions of words to be accepted even although none of the words themselves are accepted.

A scheduler may now be thought of as representing the behaviour of an acceptor.

The local memory used by an acceptor will be the same memory as the corresponding scheduler is parameterised on. For an acceptor A we may define a function beh_A which gives us a process specifying the behaviour of our acceptor; this process is a scheduler. Acceptors can therefore be used when producing schedules.

Definition

For an acceptor $A = (Q,\Sigma,M,\delta,q_0,m_0)$ we may specify its behaviour

by the process $beh_A(q_o)(m_o):\bar{\Sigma}$, where beh_A is defined recursively by

$$beh_A qm = \{\!\{ \bar{\alpha}_i : \langle o, K(beh_A(q_i)(m_i))\rangle \mid 1 \leq i \leq n \}\!\}$$

$$\text{where } \delta(q,m,\alpha_i) = (q_i,m_i) \quad .$$

As we can see, the process $beh_A qm$ is parameterised on memory m, as required. The words accepted by A have their labels complemented when they label synchronisers of $beh_A q_o m_o$ and its renewals. This is due to the definition of $\|$ and $\#$ which allow a scheduler to permit certain words.

We have the following theorem

Theorem 7.3.1

For an acceptor $A = (Q,\Sigma,M,\delta,q_o,m_o)$, A accepts δ iff $beh_A q_o m_o$ permits δ, where δ is a word of finite length.

Proof we shall prove that

$$A \text{ accepts } \sigma \Rightarrow beh_A q_o m_o \text{ permits } \sigma, \text{ for all finite } \sigma \qquad (1)$$

$$\text{and } beh_A q_o m_o \text{ permits } \sigma \Rightarrow A \text{ accepts } \sigma, \text{ for all finite } \sigma \ . \qquad (2)$$

Part 1 We assume that A accepts σ for some finite σ (3)

thus by the definition of accepts, there is a finite sequence of moves

$$(q_o,m_o,\sigma\omega) \vdash \ldots \vdash (q_n,m_n,\omega)$$

for all ω , with $|\sigma| = n$.

By our definition of move, we have that

$$\left.\begin{aligned} \delta(q_o,m_o,a_1) &= (q_1,m_1) \\ &\vdots \\ \delta(q_{n-1},m_{n-1},a_n) &= (q_n,m_n) \end{aligned}\right) \qquad (4)$$

for $\sigma = a_1 \ldots a_n$.

Thus by definition of beh_A,

$$\left.\begin{array}{l} \bar{a}_1 : \langle o, K(beh_A q_1 m_1)\rangle \in beh_A q_o m_o \\ \quad\vdots \qquad\qquad \vdots \\ \bar{a}_n : \langle o, K(beh_A q_n m_n)\rangle \in beh_A q_{n-1} m_{n-1} \end{array}\right) \qquad (5)$$

Hence from (5), and our definitions of $\mathcal{L}$, TERM and computation tree,

$$\bar{a}_1 \ldots \bar{a}_n \in TERM(\mathcal{L}(T_{beh_A q_o m_o}))$$

and hence,

$beh_A q_o m_o$ permits σ, as required.

Part 2 Assume that

$$beh_A q_o m_o \text{ permits } \sigma, \text{ for some finite } \sigma \qquad (6)$$

by our definition of permits,

$$\bar{\sigma} \in TERM(\mathcal{L}(T_{beh_A q_o m_o})) \qquad (7)$$

and so for $\sigma = a_1 a_2 \ldots a_n$,

by the definition of TERM, $\mathcal{L}$ and computation trees,

$$\left.\begin{array}{l} \bar{a}_n : \langle o, K(beh_A q_n m_n)\rangle \in beh_A q_{n-1} m_{n-1} \\ \quad\vdots \qquad\qquad \vdots \\ \bar{a}_1 : \langle o, K(beh_A q_1 m_1)\rangle \in beh_A q_o m_o \end{array}\right) \qquad (8)$$

and so from (8), by the definition of beh_A

$$\left.\begin{array}{l} \delta(q_{n-1}, m_{n-1}, a_n) = (q_n, m_n) \\ \quad\vdots \qquad\qquad \vdots \\ \delta(q_o, m_o, a_1) = (q_1, m_1) \end{array}\right) \qquad (9)$$

and from (9) and the definition of $\vdash$

$$(q_o, m_o, a_1 \ldots a_n) \vdash \ldots\ldots \vdash (q_{n-1}, m_{n-1}, a_n)$$

and for all ω

$$(q_o, m_o, \sigma\omega) \vdash \ldots\ldots \vdash (q_{n-1}, m_{n-1}, a_n\omega) \vdash (q_n, m_n, \omega),$$

hence A accepts σ, as required.

Thus from parts 1 and 2, we have theorem 7.5.1 as required.

The next theorem is similar to 7.5.1, but for infinite words.

Theorem 7.3.2

For an acceptor $A = (Q,\Sigma,M,\delta,q_o,m_o)$,

A accepts ω iff $beh_A q_o m_o$ permits ω,

for all infinite words ω .

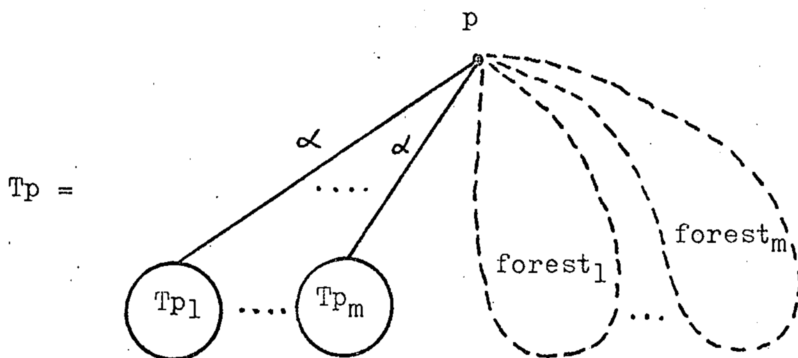
The proof of this theorem resembles that of theorem 7.3.1 and is omitted.

In the above we see how the well-understood concept of finite automaton and the function beh_A are used to produce a scheduler which permits those sequences of scheduling labels which the acceptor accepts. Schedulers may also be produced directly; this is in fact the usual method.

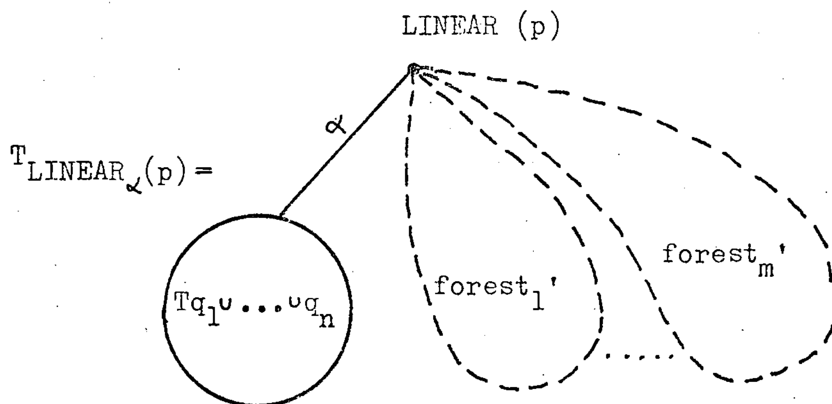
7.4 Linearisation of schedulers

Here we define the linearisation of a process which is composed only of synchronisers. We then prove that scheduling a process p using any scheduler s has the same effect as scheduling it using the linearisation of s. This result is used in the proof of Theorem 9.3.1 which appears in Chapter 9, but is interesting in its own right.

We define a function $LINEAR_\alpha$ which takes a process $p:L$, where $\alpha \in L$ labels synchronisers in p and its renewals, and linearises it with respect to α . If p has the following computation tree



where $forest_1, \ldots, forest_m$ correspond to all capabilities in p not labelled by α ; then $T_{LINEAR_\alpha(p)}$ will have tree



where $q_i = LINEAR_\alpha(p_i)$ and each subtree T_r in $forest_i$ is replaced by subtree $T_{LINEAR\ (r)}$ in $forest_i'$.

Definition

$LINEAR_\alpha : L \to L$, with $\alpha \in L$, $\alpha \neq \tau$ and $U_\alpha = V_\alpha = 1$, is defined by

$$LINEAR_\alpha(p) = \{ \alpha : \langle o, K(LIN_\alpha p)\rangle \mid \alpha : \langle o, K(p')\rangle \in p \}$$
$$\cup \{ \lambda : \langle u, \lambda v. LINEAR_\alpha(fv)\rangle \mid \lambda : \langle u, f\rangle \in p, \lambda \neq \alpha \}$$

where $LIN_\alpha : L \to L$ is defined by

$$LIN_\alpha(p) = \bigcup \{ LINEAR_\alpha(p') \mid \alpha : \langle o, K(p')\rangle \in p \}$$

Thus $LINEAR_\alpha(p)$ will contain only one synchroniser (if any) labelled by α. Similarly with its renewals. If $\alpha \notin L$ then $LINEAR_\alpha(p) = p$.

Definition A process $p:L$, with $\alpha \in L$ where $U_\alpha = V_\alpha = 1$, is linear with respect to α iff $p = LINEAR_\alpha(p)$.

Theorem 7.4.1 (commutivity)

$$LINEAR_{\alpha_1} \circ LINEAR_{\alpha_2} = LINEAR_{\alpha_2} \circ LINEAR_{\alpha_1}$$

The proof of this theorem is in the appendix.

Definition A process $p:L$, with $\alpha_1, \ldots, \alpha_n \in L$ where $U_{\alpha_1} = V_{\alpha_1} = \ldots = U_{\alpha_n} = V_{\alpha_n} = 1$, is linear with respect to $\{\alpha_1, \ldots, \alpha_n\}$ iff $p = (LINEAR_{\alpha_1} \circ \ldots \circ LINEAR_{\alpha_n})(p)$.

This definition relies on Theorem 7.4.1.

The corollary following theorem 7.4.2 illustrates that a scheduler and its linearisation with respect to its sort are equally adequate when used by our scheduling technique.

Theorem 7.4.2 For processes p:L and q:M, where $\alpha \in M, \bar{\alpha} \in L$ and $U_\alpha = V_\alpha = 1$,

$$p \| q = p \| \text{LINEAR}_\alpha(q)$$

The proof of this theorem appears in the appendix.

Corollary 7.4.3 For p:L and s:S, where s is a scheduler, $S = \{\bar{\alpha}_1', \ldots \bar{\alpha}_n'\}$ and $\bar{S} \subseteq L$, then

$$p \| s = p \| (\text{LINEAR}_{\bar{\alpha}_1'} \circ \ldots \circ \text{LINEAR}_{\bar{\alpha}_n'}(s))$$

This follows from repeated use of Theorem 7.4.2.

Theorem 7.4.2 and Corollary 7.4.3 hold due to the fact that

$$\mathcal{L}(T_p) = \mathcal{L}(T_{\text{LINEAR}_\alpha(p)})$$

This result is immediate from the definition of $\mathcal{L}$, T and LINEAR .

The following chapter illustrates the use of our scheduling technique.

CHAPTER 8

PRODUCING SCHEDULERS AND USING THE SCHEDULING TECHNIQUE

As an example we consider how to produce a scheduler which satisfies the second readers/writers problem of Courtois, Heymans and Parnas [Cou]. This example also illustrates the use of an acceptor in producing the scheduler and the technique by which scheduling lines are added to processes to enable schedules to control them. Often schedulers will be produced directly; that is, without using an acceptor and the function beh. The dining philosophers problem is used as an example of this later in this chapter.

8.1 The second readers/writers problem

We have a number of processes $p_1,\dots,p_n$ which wish to share some resource process in a well-defined manner. These processes are either reader processes or writer processes. The definition of these processes does not concern us except that we note that they are finite and are amended by the scheduling labels rd, rf, wt_1, wt_2, wf. The labels rd, wt_1, wt_2 are guard labels; rd guarding reader processes and wt_1 and wt_2 guarding writer processes, rf and wf monitor the finish of reader and writer processes respectively. All the reader processes will use the same scheduling labels and similarly all the writers will use the labels wt_1, wt_2 and wf. Let us call the reader and writer processes which have these scheduling lines added the read and write processes. Two guards control entry to the write processes since we are required to schedule two separate features of writer processes;

their write requests and the start of the actual writing. Read processes only require one guard.

Our scheduler must satisfy the following problem description; for reader, writer and resource agents.

"Writers must have exclusive access to the resource while readers may share the resource among themselves. Once a writer has announced that he is ready to write any reader which then wishes to read must wait until the writer has written, even if the writer is also waiting, possibly for another read to finish".

This problem is identical to the second readers/writers problem of Courtois et al., except that we cannot force write requests to take priority over read requests unless these are subsequent read requests. The computing agents mentioned in this problem specification will be modelled by processes and we require a scheduler which causes the reader and writer process to access the resource process in a manner consistent with this specification.

The priority part of Courtois et al's problem, that is write request priority over simultaneous read request, is left out as it is doubtful whether a priority mechanism can be introduced to the process model. Communications, or events generally, which are not forced to occur in some definite order by a scheduler may occur in any order, as we do not represent time in the model. As priorities depend on time we do not consider modelling this but model all orderings for all priorities. The problem as specified above will not therefore solve the fairness problem since this depends on write requests taking precedence over those read requests which have occurred previously.

We now define an acceptor RW which accepts correct (with respect to the problem specification above) infinite words of

scheduling labels rd, rf, wt_1, wt_2, wf.

$$RW = \langle \{q_o, q_1, q_2, q_3\}, \{rd, rf, wt_1, wt_2, wf\}, N, \delta, q_o, o \rangle$$

where N is the domain of natural number and our memory is a counter which holds such a number, initially zero.

The partial function δ is defined as follows

$$
\begin{aligned}
\delta(q_o, o, wt_1) &= (q_3, o) \\
\delta(q_o, o, rd) &= (q_1, 1) \\
\delta(q_1, i, rd) &= (q_1, i+1) \\
\delta(q_1, i, rf) &= (q_1, i-1) \quad \text{where } i>1 \\
\delta(q_1, i, wt_1) &= (q_2, i) \\
\delta(q_1, i, rf) &= (q_o, o) \quad \text{where } i=1 \\
\delta(q_2, i, rf) &= (q_2, i-1) \quad \text{where } i>o \\
\delta(q_2, i, wt_2) &= (q_3, o) \quad \text{where } i=o \\
\delta(q_3, o, wf) &= (q_o, o)
\end{aligned}
$$

Our scheduler which controls a number of reader and writer processes is then given by $beh_{RW} q_o o$;

$$
\begin{aligned}
beh_{RW}(q_o)o &= \{\!| \overline{wt}_1 : \langle o, K(beh_{RW}(q_2)o)\rangle, \overline{rd} : \langle o, K(beh_{RW}(q_1)1)\rangle |\!\} \\
beh_{RW}(q_1)i &= \{\!| \overline{rd} : \langle o, K(beh_{RW}(q_1)i+1)\rangle, \overline{wt}_1 : \langle o, K(beh_{RW}(q_2)i)\rangle |\!\} \\
beh_{RW}(q_1)j &= \{\!| \overline{rf} : \langle o, K(beh_{RW}(q_1)j-1)\rangle |\!\} \quad \text{for } j>1 \\
beh_{RW}(q_1)1 &= \{\!| \overline{rf} : \langle o, K(beh_{RW}(q_o)o)\rangle |\!\} \\
beh_{RW}(q_2)o &= \{\!| \overline{wt}_2 : \langle o, K(beh_{RW}(q_3)o)\rangle |\!\} \\
beh_{RW}(q_2)i &= \{\!| \overline{rf} : \langle o, K(beh_{RW}(q_2)i-1)\rangle |\!\} \quad \text{for } i>o \\
beh_{RW}(q_3)o &= \{\!| \overline{wf} : \langle o, K(beh_{RW}(q_o)o)\rangle |\!\}
\end{aligned}
$$

$beh_{RW}(q_o)i$ and $beh_{RW}(q_3)i$ are not defined for i=o since such "cases" do not arise and are thus irrelevant.

There is much redundancy in the definition of scheduler $beh_{RW}(q_o)o$. The ⦃ ⦄ brackets, and the ⟨o,K....⟩ part of each capability can easily be removed since all capabilities in schedulers are synchronisers. This suggests a language for writing schedulers involving only labels and colons, and our function beh could be redefined accordingly. For instance, the last clause above could be written as

$$beh_{RW}(q_3)o \quad = \quad \overline{wf}::beh_{RW}(q_o)o$$

We shall not in fact use this shorthand notation but note that it may be particularly useful for specifying scheduling rules as well as producing schedulers via acceptors.

8.2 Adding scheduling lines to processes

Processes are altered by the addition of synchronisations labelled by scheduling labels, which allow a scheduler to exercise control on them in some required manner. These synchronisations are used by a scheduler for two different reasons; as guards which control when their renewal capabilities may communicate or as monitors which indicate to the scheduler when certain capabilities, which have this monitor synchroniser as the only member of their renewal; have communicated. In the previous example rd, wt_1, wt_2 label guard synchronisations while rf, wf label monitor synchronisations. Guard synchronisations are added as follows.

G1: If p is a process whose communication we wish to control by scheduler $s:\overline{S}$ then we change p to the singleton process p' by making p the renewal of a synchroniser labelled by $\alpha \in S$.

thus $p' = \{\!\!\{\alpha:\langle o,K.p\rangle\}\!\!\}$.

G2: Guards may also be added to control when individual capabilities communicate. If we wish to control a capability $\beta:\langle u,f\rangle$ which occurs in a process q or its renewals, then we replace $\beta:\langle u,f\rangle$ by the synchronisation $\gamma:\langle o,K\{\!\!\{\beta:\langle u,f\rangle\}\!\!\}\rangle$ where $\gamma \in S$. Unfortunately this results in a great number of new labels being produced to label synchronisers. To avoid this and to enable scheduling labels to be distinguished from other labels we adopt the convention of making $S = R'$ where R is a set of ordinary labels and $R' = \{\alpha' | \alpha \in R\}$. We can then add synchronisers in a consistent manner. When adding a guard on $\beta:\langle u,f\rangle$ we may now replace this capability by the synchroniser $\beta':\langle o,K\{\!\!\{\beta:\langle u,f\rangle\}\!\!\}\rangle$.

M1: Monitor synchronisations may be added to finite processes to indicate to the scheduler when they have finished all communication other than on the τ line (which indicates termination and is never actually used in communication). Here we replace every $\tau:\langle o,K\phi\rangle$ in the finite process and its renewals, by $\gamma':\langle o,K\{\!\!\{\tau:\langle o,K\phi\rangle\}\!\!\}\rangle$ where $\gamma \neq \tau$.

M2: Monitors may also be added to indicate to the scheduler when a certain capability has communicated. To effect this we replace the renewal m of $\gamma:\langle u,\lambda v.m\rangle$ by $\{\!\!\{\gamma':\langle o,Km\rangle\}\!\!\}$, and γ' indicates to the scheduler when this γ has communicated.

Usually we wish to know when all such γ communications have taken place and so every renewal of a capability labelled by γ will be replaced as above. Similarly with guards, we will wish to control all communications on the β line. Thus we replace all capabilities labelled by β say with synchronisers labelled by β' as indicated above.

We give the following functions which add synchronisations as described above.

Definition G1

$\mathrm{FRONT}_{\alpha'} : L \rightarrow L \cup \{\alpha'\}$ is defined by

$$\mathrm{FRONT}_{\alpha'}(p) = \{\!\{ \alpha' : \langle o, Kp \rangle \}\!\}$$

Definition G2

$\mathrm{BEFORE}_{\alpha',\beta} : L \rightarrow L \cup \{\alpha'\}$ is defined recursively by

$$\mathrm{BEFORE}_{\alpha',\beta}(p) = \{\!\{ \lambda : \langle u, \lambda v.\mathrm{BEFORE}_{\alpha',\beta}(fv) \rangle \mid \lambda : \langle u, f \rangle \in p, \lambda \neq \beta \}\!\}$$
$$\cup \{\!\{ \alpha' : \langle o, K\{\!\{ \beta : \langle u, \lambda v.\mathrm{BEFORE}_{\alpha',\beta}(fv) \rangle \}\!\} \rangle \mid \beta : \langle u, f \rangle \in p \}\!\}$$

Definition M1

$\mathrm{BACK}_{\alpha'} : L \rightarrow L \cup \{\alpha'\}$, where $\tau \in L$, is defined recursively by

$$\mathrm{BACK}_{\alpha'}(p) = \{\!\{ \lambda : \langle u, \lambda v.\mathrm{BACK}_{\alpha'}(fv) \rangle \mid \lambda : \langle u, f \rangle \in p, \lambda \neq \tau \}\!\}$$
$$\cup \{\!\{ \alpha' : \langle o, K\{\!\{ \Upsilon \}\!\} \rangle \mid \Upsilon \in p \}\!\}$$

where $\Upsilon = \tau : \langle o, K\emptyset \rangle$.

Definition M2

$\mathrm{AFTER}_{\alpha,\beta'} : L \rightarrow L \cup \{\beta'\}$ is defined by

$$\mathrm{AFTER}_{\alpha,\beta'}(p) = \{\!\{ \lambda : \langle u, \lambda v.\mathrm{AFTER}_{\alpha,\beta'}(fv) \rangle \mid \lambda : \langle u, f \rangle \in p, \lambda \neq \alpha \}\!\}$$
$$\cup \{\!\{ \alpha : \langle u, \lambda v.\{\!\{ \beta' : \langle o, K.\mathrm{AFTER}_{\alpha,\beta'}(fv) \rangle \}\!\} \rangle \mid \alpha : \langle u, f \rangle \in p \}\!\}$$

These definitions prove useful when adding synchronisers in our scheduling technique. Often we will use $\mathrm{BEFORE}_{\alpha',\alpha}$ and $\mathrm{AFTER}_{\beta,\beta'}$ for some labels α and β, and this helps identify certain scheduling labels (i.e. α' and β') with what they are controlling (α and β).

To illustrate the use of the functions in our scheduling technique we return to the second readers/writers example of 8.1.

8.3 Synchronisers in the second readers/writers problem

In this example we have a number of reader and writer processes accessing a resource. Let this resource be an unbounded integer store. When a reader accesses the store it sends the location of the value to be read and the store returns the integer found at this location. When a writer accesses the store it sends both the value to be stored and the location for it, to the store. We can define a process store(s) as follows:

$$\text{store}: N^{\infty} \rightarrow P_{\{str, \bar{r}, \bar{\omega}\}} \text{ is defined by}$$

$$\text{store}(s) = \{\!\!\{ \bar{\omega}: \langle o, \lambda(n, locn).\text{store}(s \text{ with}(locn/n))\rangle,$$
$$\bar{r}: \langle o, \lambda locn. \{\!\!\{ str: \langle s(locn), K(\text{store}(s))\rangle \}\!\!\} \rangle \}\!\!\}$$

where N is the natural numbers and

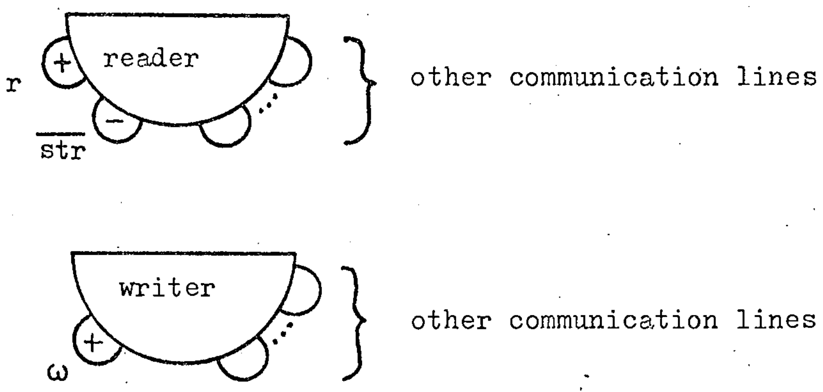
$str \mapsto N, 1;\ \ \bar{r} \mapsto 1,\ LOCN;\ \ \bar{\omega} \mapsto 1,\ (N \times LOCN)$ and $LOCN = N$.

s(i) is the contents at the ith location of the store while s with(i/n) is identical to s except that the content of location i is now the integer n. This notation is used to distinguish it from the more usual [-/-] notation which we use only to relabel processes.

Reader processes communicate with the store via the str lines, while writer processes use the ω line. Reader and writer processes may carry out other actions but as we are only scheduling their access to store(s), for some initial s, we need only consider their capabilities labelled by ω, r and $\overline{str}$.

Reader and writer processes will be finite, as we schedule some of them to start communicating when others have finished; this does not make sense with infinite reader and writer processes. The combinator $\nparallel$ will therefore be used, rather than $\|$.

We have the nets



We wish to add scheduling lines to guard "starting up" of reader processes and to monitor their termination, while we wish to guard writer processes in such a way that we can control a "request to write" separately from the "starting up" of the writer. This is necessary as our scheduler must satisfy the requirements of the problem specification in 8.1. We also wish to monitor the termination of writers.

As we are wishing to schedule a number of reader processes; reader-1,...,reader-n and a number of writer processes; writer-1,.. ..,writer-m we can use the functions FRONT and BACK to add scheduling lines to produce the processes read-1,...,read-n and write-1,...,write-m.

$$\text{read-i} = \text{FRONT}_{rd'} \circ \text{BACK}_{rf'}(\text{reader-i})$$

and

$$\text{write-i} = \text{FRONT}_{wt_1'} \circ \text{FRONT}_{wt_2'} \circ \text{BACK}(\text{writer-i})$$

As we have used the convention of adding "dashes" to scheduling lines we must change our acceptor RW to allow for this.

RW' is identical to RW but with rd', rf', wt_1', wt_2', wf' replacing rd, rf, wt_1, wt_2, wf respectively.

The process

$$((\mathop{\|}_{i=1}^{n} \text{read-i}) \| (\mathop{\|}_{i=1}^{m} \text{write-i})) \| \text{beh}_{RW'}(q_o)\text{o}$$

then satisfies the specification of the non-priority second readers/ writers problem, where

$$\mathop{\|}_{i=1}^{n} p_i = p_1 \| \ldots \| p_n$$ and we allow $\|$ between reader and writer process as we can assume that they will not inter-communicate, but will all be finite.

$$(\mathop{\|}_{i=1}^{n} \text{reader-i}) \| (\mathop{\|}_{i=1}^{m} \text{writer-i})$$ is the resulting process when no scheduling takes place on our system of reader and writer processes and unrestricted concurrency is allowed.

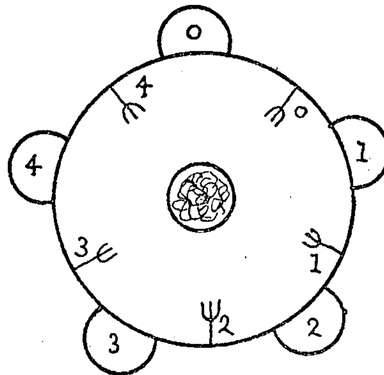
A further example of the use of our scheduling technique in modelling computing agents containing scheduling requirements follows in the next section.

8.4 The dining philosophers problem

This problem, which involves the scheduling of a number of agents, is due to Dijkstra [Dij 2]. It can be described by:

"Five philosophers spend their lives thinking and eating. They share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In

the centre of the table there is a large bowl of spaghetti and the table is laid with five forks as follows:



On feeling hungry a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he is finished, he puts down both forks and leaves the room. The room should keep a count of the number of philosophers in it".

We follow Hoare's [Hoa 3] solution to this problem by modelling the philosophers, the forks and the room. The processes which model these agents then interact via our $\|$ combinator.

Once again we abstract out the scheduling properties into a scheduler which communicates with the philosopher, fork and room processes along some scheduling lines. Thus we do not have our scheduler distributed among all the processes as Hoare does, and the correctness of our scheduler will hopefully be more apparent.

Our processes are as follows :

$philo_i : \{think, \overline{enter}, \overline{f}_i, \overline{f}_{(i+1)mod5}, eat, \overline{g}_i, \overline{g}_{(i+1)mod5}, \overline{leave}\}$

is defined recursively by

$$philo_i = \{\!\!\{ think:\langle u, K\{\!\!\{\overline{enter}:\langle o, K\{\!\!\{\overline{f}_i:\langle o, K\{\!\!\{ \overline{f}_{(i+1)mod5}:\langle o, K\{\!\!\{eat:\langle \omega, \lambda x.\{\!\!\{\overline{g}_i:\langle o, K\{\!\!\{ \overline{g}_{(i+1)mod5}:\langle o, K\{\!\!\{\overline{leave}:\langle o, K philo_i\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}\rangle\}\!\!\}$$

since philosophers sequentially think, enter the room, pick up their left-hand and their right-hand forks, eat, replace left-hand then right-hand forks then leave the room. Philosophers will repeat this sequence of actions forever.

$fork_i : \{f_i, g_i\}$ is defined by

$fork_i = \{\!\!\{f_i:\langle o, K\{\!\!\{g_i:\langle o, K fork_i\rangle\}\!\!\}\rangle\}\!\!\}$, since forks are repeatedly picked up and placed down again. The behaviour of the room is only that of counting those philosophers entering and leaving. Thus we can define $room(n) : \{enter, leave\}$ by

$$room(n) = \{\!\!\{enter:\langle o, K(room(n+1))\rangle,\ leave:\langle o, K(room(n-1))\rangle\}\!\!\}$$

As these processes only wish to communicate among themselves and never terminate we can use the $\|$ combinator to prevent their communication lines from being used by other processes. We then have the process

$$dine = \Big(\mathop{\|}_{i=o}^{4} (philo_i \| fork_i)\Big) \| room(o),$$

which models the behaviour of the spaghetti eating system, where $\mathop{\|}_{i=1}^{n} p_i = p_1 \| \ldots \| p_n$.

This unrestricted concurrency unfortunately gives the case where all philosophers enter the room, all pick up their left-hand forks then all try to pick up their right-hand forks. They fail to carry out this action and so starve to death. We design a

a scheduler to prevent this possibility from occurring.

One solution is to allow a philosopher to pick up his left-hand fork if the right-hand fork is also available. Our scheduler will therefore control the availability of forks. Synchronisers are added to fork processes to produce sfork processes.

$sfork_i : \{f_i', g_i', f_i, g_i\}$ is defined by

$$sfork_i = BEFORE_{f_i', f_i} \circ AFTER_{g_i, g_i'}(fork_i).$$

We now can define a process sdine which is similar to dine but with certain scheduling lines added.

$$sdine = (\big\Vert_{i=0}^{4} (philo_i \,\Vert\, sfork_i)) \,\Vert\, room(0)$$

We wish to define schedulers $s:S$ where $S = \{\overline{f_i}', \overline{g_i}' \mid 0 \leq i \leq 4\}$ so that <u>not all</u> philosophers starve, and so that no philosopher starves. We do not use an acceptor but define s directly.

$$s = Sc(empty) \text{ where } Sc: TRUTH^5 \rightarrow P_S, \text{ and } empty = \langle F,F,F,F,F \rangle.$$

To ensure that not all philosophers starve, Sc is defined by

$$Sc(list) = \bigcup_{i=0}^{4} \Big[(list_i = 0 \wedge list_{(i+1)mod5} = 0) \rightarrow \{\overline{f_i}' : \langle 0, K(\{\overline{f}'_{(i+1)mod5} : \langle 0, K(Sc(list\ with(i/T)((i+1)mod5/T))))\rangle\}\rangle\}$$

$$\cup (list_i = 1 \wedge list_{(i+1)mod5} = 1) \rightarrow \{\overline{g_i}' : \langle 0, K(\{\overline{g}'_{(i+1)mod5} : \langle 0, K(Sc(list\ with(i/F)((i+1)mod5/F))))\rangle\}\rangle\} \Big]$$

sdine $\Vert$ s is then the process which models the spaghetti eating problem without all philosophers starving. This is due to the scheduler only allowing forks to be picked up in pairs.

Unfortunately, due to the unfairness of our $|$ and $\|$ combinators, sdine $\|$ s does allow some philosophers to constantly eat and think while preventing some others from eating. A scheduler to prevent any philosopher starving may be defined by changing S to:

$$S = \text{allow}_0 \text{ where}$$

$$\text{allow}_i = \{\!\!\{ \overline{f_i}' : \langle o, K \{\!\!\{ \overline{f}_{(i+1)\text{mod}5} : \langle o, K \{\!\!\{ \overline{g_i}' : \langle o, K \{\!\!\{ \overline{g}_{(i+1)\text{mod}5} : \langle o, K(\text{allow}_{(i+1)\text{mod}5}) \rangle \}\!\!\} \rangle \}\!\!\} \rangle \}\!\!\} \rangle \}\!\!\}$$

We see from this example that arbitrary schedulers can be imposed upon the same system of processes, controlling these processes according to different rules.

CHAPTER 9

MULTI-LEVEL SCHEDULING

9.1 Multiple schedulers

The scheduling techniques of Chapter 7 depends on certain added synchronisations and the scheduler itself "disappearing" due to the use of the $\|$ or $\nparallel$ combinator. The resulting process, the "filtered" process, may still contain synchronisations labelled by scheduling labels but these labels will differ from those removed by the $\|$ or $\nparallel$ combinators. These remaining scheduling synchronisations may then be used by another scheduler to carry out further scheduling. This is known as repeated scheduling.

But our scheduling technique assumes that no scheduling synchronisations (either those added to a process or belonging to a scheduler, remain after filtering, to ensure that the sort of our filtered process is the same as that of the original process. For this to happen with repeated scheduling, not only must the union of the sorts of our schedulers be the complemented set of the added scheduling labels, but the sorts of the schedulers must be disjoint.

For schedulers $s_1:\overline{S_1'}$ and $s_2:\overline{S_2'}$ and a process $p:A\cup S_1'\cup S_2'$ which has scheduling synchronisations already added to it and where A contains no scheduling labels, then the filtered process $(p\|s_1)\|s_2$ will only be of sort A if $S_1\cap S_2=\phi$. This is due to the definition of $\|$. In order to allow p to communicate with s_2 along all required lines and to prevent some of those lines from being prematurely removed by communication with s_1 the following may appear to have the required effect:

$$(p|s_1)|s_2 \quad \text{or} \quad (p|s_1)\|s_2.$$

But neither of these will be of the required sort A.

$p \| (s_1 | s_2)$ will be of the correct sort but this is not <u>repeated</u> scheduling. This assumes that $(s_1 | s_2)$ is some new scheduler capturing the effect of <u>both</u> s_1 and s_2. $(s_1 | s_2)$ does not in fact do this but a combinator & is presented below where $(s_1 \& s_2)$ is a new scheduler capturing the scheduling properties of both s_1 and s_2.

For us to be able to filter a process using a number of non-disjoint schedulers (actually schedulers with non-disjoint sorts) two different techniques may be employed; that of producing a single scheduler which satisfies the scheduling rules of two schedulers via a suitable combinator, and that of adding further synchronisers to the process being scheduled to allow repeated scheduling. These two techniques will be shown to be equivalent.

9.2 The & combinator

For two schedulers $s_1 : \overline{S_1}$ and $s_2 : \overline{S_2}$, we wish a combinator which produces a new scheduler $s_3 : \overline{S_1 \cup S_2}$ that combines the scheduling properties of both s_1 and s_2. If $S_1 \cap S_2 = \emptyset$ then our combinator is required to interleave the communications of s_1, s_2 and their renewals in all possible ways. As s_1 and s_2 do not inter-communicate nor terminate the $\|$ or $|$ combinators suffice.

If $S_1 \cap S_2 \neq \emptyset$ we may have the case that $S_1 = \{\alpha', \beta'\}$ and $S_2 = \{\gamma', \beta'\}$. As the synchroniser $\overline{\alpha}' : \langle o, K \{ \overline{\beta}' : \langle \ldots . \rangle \} \rangle$ in scheduler s_1 implies that a communication on the α' line precedes that on the β' line, and as $\gamma' \notin S_1$ then s_1 allows a communication on the γ' line to occur in any order with respect to the α' and β' communications, provided the α' precedes the β' communication.

If $\overline{\alpha}' : \langle o, K \{ \overline{\beta}' : \langle \ldots . \rangle \} \rangle \in s_1$ and

$\overline{\gamma}' : \langle o, K \{ \overline{\beta}' : \langle \ldots . \rangle \} \rangle \in s_2$ then our

scheduler s_3 must be such that α, β and γ communications can occur in any order provided (i) α precedes a β communication and (ii) γ precedes a β communication. If an α precedes a γ precedes a β communication or a γ precedes an α precedes a β communication then both (i) and (ii) are satisfied. Thus $\overline{\alpha'}:\langle o,K\{\overline{\gamma'}:\langle o,K\{\overline{\beta'}:\langle\ldots\rangle\}\rangle\}\rangle$ and $\overline{\gamma'}:\langle o,K\{\overline{\alpha'}:\langle o,K\{\overline{\beta'}:\langle\ldots\rangle\}\rangle\}\rangle$ must be members of s_3. These requirements in producing s_3 are met by the $\&$ combinator.

The $\&$ combinator between schedulers which we define below is similar to that used by Plotkin in [Plo 2]. That this combinator "correctly" produces a scheduler containing the scheduling properties of its component schedulers follows from Theorem 9.3.1.

Definition

$\& : L_1 \times L_2 \rightarrow L_1 \cup L_2$ is defined for pure synchronising processes only, by

$$\begin{aligned} p \,\&\, q \;=\; & \{\lambda:\langle o, K(p_1 \& q)\rangle \mid \lambda:\langle o, Kp_1\rangle \in p, \lambda \in L_1 - L_2\} \\ \cup\; & \{\lambda:\langle o, K(p \& q_1)\rangle \mid \lambda:\langle o, Kq_1\rangle \in q, \lambda \in L_2 - L_1\} \\ \cup\; & \{\lambda:\langle o, K(p_1 \& q_1)\rangle \mid \lambda:\langle o, Kp_1\rangle \in p, \lambda:\langle o, Kq_1\rangle \in q, \lambda \in L_1 \cap L_2\} \end{aligned}$$

For $s_1:\overline{S_1}$ and $s_2:\overline{S_2}$ then if $S_1 \cap S_2 = \emptyset$, $s_1 \& s_2 = s_1 \| s_2 = s_1 | s_2$. This follows directly from the definitions of these combinators.

If p and q are incompatible in the sense that no process can satisfy both the scheduling rules of p and of q, the combinator gives us the empty set. For example suppose that

$$p = \{\alpha:\langle o, K\{\beta:\langle o, K\emptyset\rangle\}\rangle\}$$

$$q = \{\beta:\langle o, K\{\alpha:\langle o, K\emptyset\rangle\}\rangle\}$$

then p indicates that an α communication should precede a β communication while q indicates the opposite, thus no process can satisfy both and $p \& q = \emptyset$.

Our previous scheduling technique of Chapter 7 will be known as one-level scheduling, since we will filter out various communications from a process only once. The & combinator allows us to filter a process with more than one scheduler using the one-level technique, and prevents the problems of repeated scheduling which were mentioned in Section 9.1.

But we may still wish to filter out various communications from a process on more than one occasion by using a number of non-disjoint schedulers. A technique for implementing this will be known as multi-level scheduling.

9.3 Multi-level scheduling

We introduce a new form of synchroniser known as a marking synchroniser. These will be added to processes just as our scheduling synchronisers were with one-level scheduling. Marking synchronisers "keep the place" in processes where we wish to add scheduling synchronisers. These "places" are not removed when we apply || or ∦ to this amended process and our scheduler, so enabling us to add further scheduling synchronisers and carry out another level of scheduling. After all scheduling has been completed all marking lines are removed by a marker process.

Marking labels are distinguished by a superfixed asterisk *, just as a scheduling label is distinguished by a superfixed dash '. Marking synchronisers are added to processes using the BEFORE, AFTER, FRONT and END functions, or else in an ad hoc manner. They are added in exactly the same places, to carry out the same functions, as scheduling synchronisers in the one-level technique. We define a function SCHED which adds scheduling synchronisers "after" marking

synchronisers in much the same way as the function AFTER.

Definition

$SCHED_B : (L \cup A^*) \rightarrow (L \cup A^* \cup B')$

where L contains no marking or scheduling labels is defined by

$$SCHED_B(p) = \{\alpha : \langle u, \lambda v. SCHED_B(fv)\rangle \mid \alpha : \langle u, f\rangle \in p, \alpha \notin B^*\}$$
$$\cup \{\alpha^* : \langle o, K\{\alpha' : \langle o, K(SCHED_B(p'))\rangle\}\rangle \mid \alpha^* : \langle o, Kp'\rangle \in p, \alpha^* \in B^*\}$$

For our purposes the scheduling synchronisers could equally well have been added "before" the corresponding marking synchroniser.

We now define a function FILTER which utilises SCHED when filtering a process.

Definition

$FILTER : \overline{B'} \rightarrow ((L \cup A^*) \rightarrow (L \cup A^*))$

where $B \subseteq A$ and L contains no marking or scheduling labels, is defined by

$$FILTER\ s\ p = s \parallel SCHED_B(p) \quad \text{if } \tau \notin L$$
$$= s \not\parallel SCHED_B(p) \quad \text{if } \tau \in L$$

Note that scheduler s is of sort $\overline{B'}$.

Using the FILTER function we can effect multi-level scheduling by

$(FILTER\ s_2 \circ FILTER\ s_1)q \parallel marker_{A^*}$

where $q : L \cup A^*$ and $marker_{A^*} : \overline{A^*}$ is defined by $marker_{A^*} = \{\alpha : \langle o, Kmarker_{A^*}\rangle \mid \alpha \in A^*\}$. Similarly with $\not\parallel$. This process just "permits everything" and removes all marking synchronisers from $(FILTER\ s_2 \circ FILTER\ s_1)q$ via $\parallel$ or $\not\parallel$.

The following theorem justifies our definition of $\&$.

Theorem 9.3.1

$$\text{FILTER } s_2 \circ \text{FILTER } s_1 \;=\; \text{FILTER } (s_2 \,\&\, s_1)$$

The proof of this theorem appears in the appendix. This theorem essentially states that multi-level scheduling is equivalent to one-level scheduling using the & combinator.

The following lemma allows us to prove a useful corollary to theorem 9.3.1.

Lemma 9.3.2 (commutivity of &)

For pure synchronising processes p and q

$$p \,\&\, q \;=\; q \,\&\, p$$

The proof of this lemma is similar to that of Law(F1) in Chapter 5, and is omitted.

Corollary 9.3.3

$$\text{FILTER } s_1 \circ \text{FILTER } s_2 \;=\; \text{FILTER } s_2 \circ \text{FILTER } s_1$$

Proof $\text{FILTER } s_2 \circ \text{FILTER } s_1 = \text{FILTER } (s_2 \& s_1)$ by 9.3.1

$= \text{FILTER } (s_1 \& s_2)$ by 9.3.2

$= \text{FILTER } s_1 \;\&\; \text{FILTER } s_2$

by 9.3.1, as required

This corollary is a necessary feature of multi-level scheduling; that filtering out some communications from a process with respect to one scheduler and then another should be the same as reversing the order in which the schedulers were used.

The following theorem is also useful when considering multi-level scheduling:

Theorem 9.3.4

$$(\text{FILTER } s\ p) \| q = \text{FILTER } s\ (p \| q)$$

where $p{:}L \cup A^*$, $s{:}\overline{B'}$, $q{:}M$, $B \subseteq A$, $\gamma \notin L \cup M$ and $A^* \cap M = \phi$, that is, there are no common marking synchronisers in p and q and between their renewals. The proof of this theorem appears in the appendix.

Informally, this theorem states that scheduling a process p using our multi-level technique and then allowing it to communicate with process q, is the same as if p and q communicate first and the resulting process is then scheduled, provided that p and q do not contain common marking labels. This theorem can be extended to the case where both p and q are scheduled by disjoint schedulers .

Theorem 9.3.5

$$(\text{FILTER } s_1\ p) \| (\text{FILTER } s_2\ q) = \text{FILTER } (s_1 \& s_2)(p \| q)$$

where $p{:}L \cup A^*$, $q{:}M \cup C^*$, $s_1{:}\overline{B^*}$, $s_2{:}\overline{D^*}$, $B \subseteq A$, $D \subseteq C$ and $A \cap C = \phi$. L and M do not contain marking or termination scheduling labels.

Proof

$(\text{FILTER } s_1\ p) \| (\text{FILTER } s_2\ q) =$

$\quad \text{FILTER } s_1 (p \| (\text{FILTER } s_2\ q))$

$\qquad$ by corollary 9.3.3

$\quad = \text{FILTER } s_1\ ((\text{FILTER } s_2\ q) \| p)$

$\qquad$ by commutivity of $\|$

$\quad = \text{FILTER } s_1\ (\text{FILTER } s_2 (q \| p))$

$\qquad$ by corollary 9.3.3

$= \text{FILTER } s_1 \ (\text{FILTER } s_2 \ (p \parallel q))$

by commutivity of $\parallel$

$= \text{FILTER } (s_1 \& s_2)(p \parallel q)$

by Theorem 9.3.1, as required

Theorems similar to 9.3.4 and 9.3.5 but using $\#$ can also be given when $\Upsilon \in L$ and/or $\Upsilon \in M$. As an example of the use of both the scheduling techniques and the theorems we give two process semantics for a concurrent programming language due to Habermann [Hab] and following from the work of Campbell and Habermann [Cam], and we prove these two semantics equivalent. As these two semantics only differ in how they deal with scheduling, details of the language and its semantics which are not concerned with scheduling are not relevant to the proof. These language details are included only to illustrate how the process model can be used as a semantic domain when specifying the semantics of a concurrent programming language.

9.4 Denotational semantics of PPL

We give two process semantics of the language PPL; the Path Program Language. This language is similar to the one given by Habermann [Hab] except that we leave out his & operator and also provide a simple syntax for the program body, involving concurrency.

This language uses path expressions to schedule the occurrence of procedures present in the program body. A typical path expression is path a;b;c end; where a,b and c are procedure names. Such a path expression means that the only permissible execution sequence involving these procedures is: abcabc ... . Other procedures can execute anywhere relative to a,b and c provided that restrictions due to any other path expression for the same path program are met.

Certain path expressions are regular expressions, and these expressions denote possible execution sequences. The syntax of PPL is defined as follows

path program ::= body | multipath; body

multipath ::= path | path and multipath

path ::= path seqpath end

seqpath ::= sequence | sequence *

sequence ::= orelement | orelement; sequence

orelement ::= cond | cond + orelement

cond ::= element | [$bool_1$:$element_1$, ...,$bool_n$:$element_n$,$element_{n+1}$]

element ::= procname | (seqpath)

procname ::= $character^+$

character ::= a | b | .... | z

bool ::= var iop con | true | false

con ::= 0 | 1 | 2 | ....

var ::= m_1 | m_2 | .... | m_n

iop ::= > | < | = | ≥ | ≤

body ::= sproc | sproc par body

sproc ::= procedure | procedure; sproc

procedure ::= procname: exps | (body)

exps ::= exps | assig; exps

assig ::= var ← con | print (var)

This syntax, though simple, contains many familiar language features. We assume that the only tests allowed within our path expressions are integer tests using iop. The body of a PPL program is restricted to a number of procedures acting concurrently (via par) and all that these procedures can do is to assign integer values to memory variables,

or output some contents of memory. These simplifying assumptions allow us to concentrate on producing a process semantics, but other language features should be easily added.

A semantics for PPL may be represented by a number of functionals taking syntactic arguments and producing processes.

$$\mathcal{M}: \text{pathprogram} \rightarrow P_{\{out_1,\ldots,out_n\} \cup \{\tau\}}$$

for some integer n, the size of memory attached to the program.

$$\mathcal{C}: \quad \text{con} \rightarrow N$$

$$\mathcal{R}: \quad \text{body} \rightarrow P_{\{out_1,\ldots,out_n\} \cup B' \cup \{\tau\}}$$

where B = I (names (body))

$$\mathcal{S}: \quad \text{multipath} \rightarrow P_{\bar{c} \cup \{\tau\}}$$

where c=I (names (multipath))

$$\mathcal{N}: \quad \text{body} \rightarrow P_{\{out_1,\ldots,out_n\} \cup \{\tau\}}$$

$$\mathcal{B}: \quad \text{bool} \rightarrow P_{\{r,\overline{str},\varkappa,\tau\}}$$

The functional $\mathcal{N}$ is used to give the semantics of a PPL program which contains no path expressions; that is no scheduler is constructed to control those processes which are denoted by procedures in the program body.. The functional $\mathcal{R}$ is used to give the semantics of the body of a PPL program which does involve path expressions. Path expressions will be modelled by scheduler processes and the $\#$ combinator is used (along with the $\&$ combinator) in an application of our one-level scheduling technique.

The function I takes a set of procedure names and creates scheduling labels out of them by $I(B) = \{a\text{-in}', a\text{-out}' \mid a \in B\}$. We create scheduling labels rather than marking labels as we are using one-level scheduling.

The function names is used to extract those procedure names used in parts of the syntax. It can be defined informally by:

$$\text{names}[\![F(\text{procname}_1,\ldots,\text{procname}_n)]\!] = \{\text{procname}_1,\ldots,\text{procname}_n\}$$

where $F(\text{procname}_1,\ldots,\text{procname}_n) = \text{body}$.

The functions I and names are used by our scheduling technique to label scheduling synchronisers.

The functionals $\mathfrak{M}$, $\mathcal{C}$, $\mathcal{R}$, $\mathcal{S}$, $\mathfrak{N}$, $\mathcal{B}$ are defined by

$$\mathfrak{M}[\![\text{multipath;body}]\!] = (\mathcal{S}[\![\text{multipath}]\!]) \,\#\, \mathcal{R}[\![\text{body}]\!] \,\#\, \text{MONITOR}_A \,\#\, \text{MEMORY}(b)$$

where $A = I(\text{names}[\![\text{body}]\!] - \text{names}[\![\text{multipath}]\!])$

and $b:N^n = \underbrace{\langle o,\ldots o\rangle}_{n \text{ times}}$

$$\mathfrak{M}[\![\text{body}]\!] = \mathfrak{N}[\![\text{body}]\!]$$

$$\mathfrak{N}[\![\text{sproc } \underline{\text{par}} \text{ body}]\!] = \mathfrak{N}[\![\text{sproc}]\!] \,\#\, \mathfrak{N}[\![\text{body}]\!] \,\#\, \text{MEMORY}(b)$$

$$\mathfrak{N}[\![\text{procedure;sproc}]\!] = \text{SERIAL}(\mathfrak{N}[\![\text{procedure}]\!], \mathfrak{N}[\![\text{sproc}]\!])$$

$$\mathfrak{N}[\![(\text{body})]\!] = \mathfrak{N}[\![\text{body}]\!]$$

$$\mathfrak{N}[\![\text{procname: exps}]\!] = \mathfrak{N}[\![\text{exps}]\!]$$

$$\mathfrak{N}[\![\text{assig; exps}]\!] = \text{SERIAL}(\mathfrak{N}[\![\text{assig}]\!], \mathfrak{N}[\![\text{exps}]\!])$$

$$\mathfrak{N}[\![m_i \leftarrow \text{con}]\!] = \{\omega : \langle (\mathcal{C}[\![\text{con}]\!], i), K\{\Upsilon\}\rangle\}$$

$$\mathcal{C}[\![\text{con}]\!] = \text{con}$$

$$\mathfrak{N}[\![\underline{\text{print}}\ m_i]\!] = \{r:\langle i, K\{\overline{\text{str}}:\langle o, \lambda u.\{\text{out}_i:\langle u, K\{\Upsilon\}\rangle\}\rangle\}\rangle\}$$

$$\mathcal{R}[\![\text{sproc } \underline{\text{par}} \text{ body}]\!] = \mathcal{R}[\![\text{sproc}]\!] \,\#\, \mathcal{R}[\![\text{body}]\!]$$

$$\mathcal{R}[\![\text{procedure;sproc}]\!] = \text{SERIAL}(\mathcal{R}[\![\text{procedure}]\!], \mathcal{R}[\![\text{sproc}]\!])$$

$$\mathcal{R}[\![(\text{body})]\!] = \mathcal{R}[\![\text{body}]\!]$$

$$\mathcal{R}[\![\text{procname: exps}]\!] = \text{BRACKET}_{a\text{-in}', a\text{-out}'}(\mathfrak{N}[\![\text{procname: exps}]\!])$$

where $a = \text{procname}$

$$\mathcal{S}[\![\text{path } \underline{\text{and}} \text{ multipath}]\!] = \mathcal{S}[\![\text{path}]\!] \,\&\, \mathcal{S}[\![\text{multipath}]\!]$$

$$\mathcal{S}[\![\underline{\text{path}} \text{ seqpath } \underline{\text{end}}]\!] = Y(\lambda s.\text{SERIAL}(\mathcal{S}[\![\text{seqpath}]\!], s))$$

$$\mathcal{S}[\![\text{sequence}*]\!] = \text{STAR}(\mathcal{S}[\![\text{sequence}]\!])$$

$\mathcal{S}[\![\text{orelement;sequence}]\!] = \text{SERIAL}(\ \mathcal{S}[\![\text{orelement}]\!],\ \mathcal{S}[\![\text{sequence}]\!]\ )$

$\mathcal{S}[\![\text{cond + orelement}]\!] = \mathcal{S}[\![\text{cond}]\!] \cup \mathcal{S}[\![\text{orelement}]\!]$

$\mathcal{S}[\![\,[\text{bool}_1:\text{element}_1,\ldots,\text{element}_{n+1}]\,]\!] =$

$\quad \text{COND}(\ \mathcal{B}[\![\text{bool}_1]\!],\ \mathcal{S}[\![\text{element}_1]\!],\text{COND}(\ldots$

$\quad \ldots\ \text{COND}(\ \mathcal{B}[\![\text{bool}_n]\!],\ \mathcal{S}[\![\text{element}_n]\!],\ \mathcal{S}[\![\text{element}_{n+1}]\!]\ ))\ldots)$

$\mathcal{S}[\![(\text{seqpath})]\!] = \mathcal{S}[\![\text{seqpath}]\!]$

$\mathcal{S}[\![\text{procname}]\!] = \{\!\!\{\ \overline{\text{a-in}}':\langle o,K\{\!\!\{\overline{\text{a-out}}':\langle o,K\ \{\!\!\{\Upsilon\}\!\!\}>\}\!\!\}>\}\!\!\}$

where a = procname

$\mathcal{B}[\![\underline{\text{true}}]\!] = \{\!\!\{\tau:\langle tt,K\ \{\!\!\{\Upsilon\}\!\!\}>\}\!\!\}$

$\mathcal{B}[\![\underline{\text{false}}]\!] = \{\!\!\{\tau:\langle ff,K\ \{\!\!\{\Upsilon\}\!\!\}>\}\!\!\}$

$\mathcal{B}[\![m_i\,\text{iop con}]\!] =$

$\quad \{\!\!\{\ r:\langle i,K\{\!\!\{\overline{\text{str}}:\langle o,\lambda i.(i\,\text{iop}\,\mathcal{C}[\![\text{con}]\!])\rightarrow$

$\quad \{\!\!\{\tau:\langle tt,K\ \{\!\!\{\Upsilon\}\!\!\}>\}\!\!\}\ ,$

$\quad \{\!\!\{\tau:\langle ff,K\ \{\!\!\{\Upsilon\}\!\!\}>\}\!\!\}\ >\}\!\!\}>\}\!\!\}$

The following functions are used above:

$\text{MONITOR}_A:\overline{A} = \{\!\!\{\ \overline{\alpha}:\langle o,K(\text{MONITOR}_A)>\ |\ \alpha\in A\ \}\!\!\}$

which removes via $\#$ extra scheduling lines added by $\mathcal{R}$ and not removed by $\mathcal{S}$. This applies to those scheduling lines due to procedure names appearing in the program body and not in path expressions

$\text{MEMORY}:N^n \rightarrow P_{\{\overline{\omega},\overline{r},\text{str}\}}$ is defined by

$\text{MEMORY}(m) = \{\!\!\{\ \overline{\omega}:\langle o,\lambda(v,\text{locn}).\text{MEMORY}(m[i/v])>,$

$\qquad \overline{r}:\langle o,\ \lambda i.\{\!\!\{\text{str}:\langle m(i),\text{MEMORY}(m)>\}\!\!\}>\}\!\!\}$

$\text{BRACKET}_{\alpha,\beta}:L\rightarrow L\cup\{\alpha,\beta\}$ is defined by

$\text{BRACKET}_{\alpha,\beta}(p) = \text{FRONT}_\alpha \circ \text{BACK}_\beta(p)$

where FRONT and BACK are defined in Chapter 8.

SERIAL: $L \times M \rightarrow L \cup M$ is defined by

$$\text{SERIAL}(p,q) = \{\!\!\{ \alpha:\langle u, \lambda v.\text{SERIAL}(fv,q)\rangle \mid \alpha:\langle u,f\rangle \in p,\ \alpha \neq \Upsilon \}\!\!\} \cup \bigcup \{\!\!\{ q \mid \Upsilon \in p \}\!\!\}$$

STAR: $L \rightarrow L$ is defined by

$$\text{STAR}(p) = \Upsilon \cup \text{SERIAL}(p, \text{STAR}(p))$$

COND: $(Q \cup \{\pi\}) \times L \times M \rightarrow L \cup M \cup Q$, where $Q = \{\Upsilon, r, \overline{str}\}$, is defined by

$$\text{COND}(b,p,q) = \{\!\!\{ \overline{\pi}:\langle o, \lambda t.(t=tt) \rightarrow p, ((t=ff) \rightarrow q, \bot)\rangle \}\!\!\} \not\Vdash b$$

This denotational semantics of PPL utilises a single scheduler $\mathcal{S}[\![\text{multipath}]\!]$ which is formed using our $\&$ combinator if multipath consists of more than one path. We then use our one-level scheduling technique, that is in $\mathcal{S}[\![\text{multipath}]\!] \not\Vdash \mathcal{R}[\![\text{body}]\!]$. An alternative semantics uses our multilevel scheduling technique.

9.5 Alternative semantics for PPL

We may rewrite the syntactic clause for a pathprogram as

pathprogram::=body | path_1 <u>and</u> ... <u>and</u> path_n;body

and leave the other clauses unchanged. This does not alter the syntax of PPL.

We introduce two additional functionals $\mathcal{E}$ and $\mathcal{F}$.

$\mathcal{E}$: multipath $\rightarrow (P_S \rightarrow P_S)$

$\mathcal{F}$: path $\rightarrow (P_S \rightarrow P_S)$

where $S = J(\text{names}(\text{body})) \cup \{\Upsilon\} \cup \{\text{out}_1, \ldots, \text{out}_n\}$

and $J(B) = \{\text{a-in}^*, \text{a-out}^* \mid a \in B\}$.

The function J takes procedure names and creates marking labels out of them. These will be used by the function SCHED to add guard and monitor synchronisers to the process being filtered. The

functionals that we use in this semantics will be identical to the previous semantics except that we replace $\mathcal{M}$ by $\mathcal{M}'$ and $\mathcal{R}$ by $\mathcal{R}'$ where

$$\mathcal{M}' [\![\text{path}_1 \ \underline{\text{and}} \ \ldots \ \underline{\text{and}} \ \text{path}_n ; \text{body}]\!] =$$

$$(\mathcal{E}[\![\text{path}_1 \ \underline{\text{and}} \ \ldots \ \text{path}_n]\!])\mathcal{R}'[\![\text{body}]\!] \not\parallel \text{MONITOR}_{J(\text{names(body)})} \not\parallel \ldots$$

$$\ldots \text{MEMORY}(b)$$

$$\text{with } b = \underbrace{\langle o, o \ldots o \rangle}_{n \text{ times}}$$

$$\mathcal{M}'[\![\text{body}]\!] = \mathcal{N}[\![\text{body}]\!]$$

and $\mathcal{R}'$ is identical to $\mathcal{R}$ except for the clause

$$\mathcal{R}'[\![\text{procname:exps}]\!] = \text{BRACKET}_{a\text{-in}^*, a\text{-out}^*}(\mathcal{N}[\![\text{procname:exps}]\!])$$

$$\text{where } a = \text{procname}$$

$\mathcal{E}$ and $\mathcal{F}$ are defined as

$$\mathcal{E}[\![\text{path}_1 \ \underline{\text{and}} \ \ldots \ \underline{\text{and}} \ \text{path}_n]\!] = (\mathcal{F}[\![\text{path}_1]\!]) \circ \ldots \circ (\mathcal{F}[\![\text{path}_n]\!])$$

$$\mathcal{F}[\![\text{path}]\!] = \lambda p \in P_S . \text{SCHED}_{K(\text{names(path)})}(p) \not\parallel \mathcal{S}[\![\text{path}]\!]$$

where $K(B) = \{a\text{-in}, a\text{-out} \mid a \in B\}$ and SCHED is defined earlier in this chapter.

The definition of $\mathcal{E}$ and $\mathcal{F}$ and the use of SCHED and marker labels, illustrate how multilevel scheduling on the schedulers $\mathcal{S}[\![\text{path}_1]\!], \ldots, \mathcal{S}[\![\text{path}_n]\!]$ can be used in the specification of the denotational semantics of PPL.

We would expect that $\mathcal{M}$ and $\mathcal{M}'$ are equivalent, for our treatment of the two semantics to be consistent. The following theorem states this.

Theorem 9.5.1

For $\mathcal{M}$ and $\mathcal{M}'$ as defined above,

$$\mathcal{M}[\![\text{pathprogram}]\!] = \mathcal{M}'[\![\text{pathprogram}]\!]$$

The proof of this theorem requires the additional lemma:

Lemma 9.5.2 For $p: L \cup A^*$, $B \subseteq A$, $\Upsilon \in L$ and L contains no marking or scheduling labels, then

$$SCHED_B(p) \parallel MONITOR_{A*} = DASH(p) \parallel MONITOR_{(A-B)'}$$

where MONITOR is defined above and $DASH: L \cup A^* \to L \cup A'$ is defined by

$$DASH(p) = \{\alpha : \langle u, \lambda v. DASH(fv)\rangle \mid \alpha : \langle u, f\rangle \in p, \alpha \notin A^* \}$$
$$\{\alpha' : \langle o, K\ DASH(q)\rangle \mid \alpha^* : \langle o, Kq\rangle \in p, \alpha^* \in A^* \}$$

The function DASH is used to change marking synchronisers into scheduling synchronisers by relabelling. This lemma is proved in the appendix.

Proof of Theorem 9.5.1

We argue by case analysis;

Case 1 pathprogram = body

then $\mathcal{M}[\![\text{pathprogram}]\!] = \mathcal{R}[\![\text{body}]\!] = \mathcal{M}'[\![\text{pathprogram}]\!]$ as required,

by definition of $\mathcal{M}$ and $\mathcal{M}'$.

Case 2 pathprogram = $path_1$ and ... and $path_n$; body

$\mathcal{M}[\![\text{pathprogram}]\!] =$

$$\mathcal{S}[\![path_1 \text{ and } \dots \text{ and } path_n]\!] \parallel \mathcal{R}[\![\text{body}]\!] \parallel MONITOR_A \parallel MEMORY(b)$$

where A = I (names(body) - names(multipath))

and $I(B) = \{a\text{-in}', a\text{-out}' \mid a \in B\}$, by definition of $\mathcal{M}$,

$$= (\mathcal{S}[\![path_1]\!] \,\&\, \dots \,\&\, \mathcal{S}[\![path_n]\!]) \parallel \mathcal{R}[\![\text{body}]\!] \parallel MONITOR_A \parallel MEMORY(b)$$

by definition of $\mathcal{S}$

$= \quad (\mathcal{S}[\![\text{path}_1]\!] \& \ldots \& \mathcal{S}[\![\text{path}_n]\!]) \,\#\, \text{DASH}(\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_A \,\#\, \text{MEMORY}(b)$

by definition of DASH and $\mathcal{R}'$

$= \quad (\mathcal{S}[\![\text{path}_1]\!] \& \ldots \& \mathcal{S}[\![\text{path}_n]\!]) \,\#\, \text{DASH}(\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_C \,\#\, \text{MEMORY}(b)$

where C = K (names(body) - names(multipath))'

by definition of K, I and A.

$= \quad (\mathcal{S}[\![\text{path}_1]\!] \& \ldots \& \mathcal{S}[\![\text{path}_n]\!]) \,\#\, \text{SCHED}_D(\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_E \,\#\, \text{MEMORY}(b)$

where D = K(names(multipath))

and E = J(names(body))

by Lemma 9.5.2 and the definition of K and J

$= \quad \text{FILTER}(\mathcal{S}[\![\text{path}_1]\!] \& \ldots \& \mathcal{S}[\![\text{path}_n]\!])(\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_E \,\#\, \text{MEMORY}(b)$

by definition of FILTER, as $\tau \in$ sort of $\mathcal{R}'[\![\text{body}]\!]$

$= \quad \text{FILTER}(\mathcal{S}[\![\text{path}_1]\!]) \circ \ldots \circ \text{FILTER}(\mathcal{S}[\![\text{path}_1]\!])(\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_E \,\#\, \text{MEMORY}(b)$

by Theorem 9.3.1, n-1 times

$= \quad \mathcal{J}[\![\text{path}_1]\!] \circ \ldots \circ \mathcal{J}[\![\text{path}_n]\!] \; (\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_E \,\#\, \text{MEMORY}(b)$

since $\mathcal{J}[\![\text{path}]\!]\, p = \text{SCHED}_{K(\text{names}(\text{path}))}(p) \,\#\, \mathcal{S}[\![\text{path}]\!]$

$= \text{FILTER}(\mathcal{S}[\![\text{path}]\!])p$, for $\tau \in$ sort of p.

$= \quad \mathcal{E}[\![\text{path}_1 \; \underline{\text{and}} \; \ldots \; \underline{\text{and}} \; \text{path}_n]\!](\mathcal{R}'[\![\text{body}]\!]) \,\#\, \text{MONITOR}_E \,\#\, \text{MEMORY}(b)$

by definition of $\mathcal{E}$

$= \quad \mathcal{M}'[\![\text{path}_1 \; \underline{\text{and}} \; \ldots \; \underline{\text{and}} \; \text{path}_n ; \text{body}]\!]$

by definition of $\mathcal{M}'$ as required .

Hence $\mathcal{M} \equiv \mathcal{M}'$ and we have a special case of the equivalence of our multilevel and one-level scheduling techniques.

CONCLUDING REMARKS

We have produced a mathematical model for concurrent computation in which processes and the communication between them can be studied. This model does not concern itself with practical questions such as implementation, efficiency and hardware constraints but rather tries to represent the primitive concepts of concurrent computation in a precise, intuitive way. Practical questions of concurrency are important but should be investigated in some other framework where the quantitative aspects of concurrency can be represented.

Some shortcoming of the process model have been mentioned previously; for instance in Chapter 6 we point out deficiencies of the ordering used when modelling deadlock. Further work is therefore needed to allow us to model deadlock in a more accurate manner and in general to relate the model to the many practical problems of concurrent computation.

The work reported in the previous chapters attempts to show in an experimental fashion that many features of concurrent computation are modelled effectively by processes. But to show whether our model is adequate as a framework in which to formally represent such real computing agents as operating systems, computer networks and hardware modules will require extensive experimentation. Only then will we know if our model is "on the right track" and only then will deficiencies of the model become apparent and thereby suggest a better model.

It may be the case that other flow algebras are needed to provide a realistic model of certain features of concurrent computation. For instance a more operational flow algebra (such

as a flow algebra of computation trees similar to those in Chapter 6) may be more suitable for some purposes than the algebra of processes. Future research should aim to investigate other flow algebras as candidates for "better" models and also determine what classes of concurrent computation are best modelled in different flow algebras.

Finally we should mention that a flow algebra of nets ([Mil 1] and [Mil 4]) which was suggested by the process algebra may be the basis of a programming language which utilises the full power of concurrency. This approach, from semantics to syntax, may well produce a useful language for concurrency particularly if it corresponds to a language developed from a more practical standpoint (such as that of Hoare [Hoa 3]). Further work in developing such a language would be valuable.

APPENDIX

Many of the proofs by induction in this appendix are rather similar. Unfortunately we have been unable to find general results more powerful than those mentioned in Chapter 4 which might shorten these proofs. They are therefore presented in all their detail.

Theorem 1.5.1 for p:L and q:M $p \| q = p \|' q$

where $p \| q \;=\; (p|q)\backslash\alpha_1 \ldots \backslash\alpha_n,$

$L.M \;=\; \{\alpha_1, \bar{\alpha}_1, \ldots, \alpha_n, \bar{\alpha}_n\}$

and

$$\begin{aligned} p \|' q \;=\; & \{\!| \lambda{:}\langle u, \lambda v.(fv \|' q)\rangle \mid \lambda \in L{-}L.M, \lambda{:}\langle u,f\rangle \in p |\!\} \\ & \cup \{\!| \lambda{:}\langle x, \lambda y.(p \|' gy)\rangle \mid \lambda \in M{-}L.M, \lambda{:}\langle x,g\rangle \in q |\!\} \\ & \cup\, \mathrm{U}\{\!| (fx \|' gu) \mid \lambda, \bar{\lambda} \in L.M, \lambda{:}\langle u,f\rangle \in p, \bar{\lambda}{:}\langle x,g\rangle \in q |\!\} \end{aligned}$$

Proof we show that

$$\forall p,q.\; (p\,|_i\,q)\backslash\alpha_1 \ldots \backslash\alpha_n \;=\; p \|_i' q \text{ for all } i \geq 0 \qquad (1)$$

where $L.M = \{\alpha_1, \bar{\alpha}_1, \ldots \alpha_n, \bar{\alpha}_n\}$

Basis

$$\begin{aligned} (p\,|_0\,q)\backslash\alpha_1 \ldots \backslash\alpha_n \;&=\; \bot \quad \text{by definition of } \backslash\alpha \\ &=\; p \|_0' q \quad \text{as required} \end{aligned}$$

Induction step Assume (1) holds for some $i \geq 0$ and show for $i+1$.

$$\begin{aligned} (p\,|_{i+1}\,q)\backslash\alpha_1 \ldots \backslash\alpha_n \;=\; & \{\!| \lambda{:}\langle u, \lambda v.(fv\,|_i\,q)\backslash\alpha_1\rangle \mid \lambda \in L{-}\{\alpha_1, \bar{\alpha}_1\}, \lambda{:}\langle u,f\rangle \in p |\!\}\backslash\alpha_2 \ldots \backslash \\ & \cup \{\!| \lambda{:}\langle x, \lambda y.(p\,|_i\,gy)\backslash\alpha_1\rangle \mid \lambda \in M{-}\{\alpha_1, \bar{\alpha}_1\}, \lambda{:}\langle u,f\rangle \in q |\!\}\backslash\alpha_2 \ldots \backslash \\ & \cup\, \mathrm{U}\{\!| (fx\,|_i\,gu)\backslash\alpha_1 \mid \lambda \in L, \bar{\lambda} \in M, \lambda{:}\langle u,f\rangle \in p, \bar{\lambda}{:}\langle x,g\rangle \in q |\!\}\backslash\alpha_2 \ldots \backslash\alpha_n \end{aligned}$$

by the definition of $|$, $\backslash\alpha_1$ and the composition theorems and lemma

$$\begin{aligned} = \; & \{\!| \lambda{:}\langle u, \lambda v.(fv\,|_i\,q)\backslash\alpha_1 \ldots \backslash\alpha_n \mid \lambda \in L{-}\{\alpha_1, \bar{\alpha}_1, \ldots \alpha_n, \bar{\alpha}_n\}, \lambda{:}\langle u,f\rangle \in p |\!\} \\ & \cup \{\!| \lambda{:}\langle x, \lambda y.(p\,|_i\,gy)\backslash\alpha_1 \ldots \backslash\alpha_n \mid \lambda \in M{-}\{\alpha_1, \bar{\alpha}_1, \ldots \alpha_n, \bar{\alpha}_n\}, \lambda{:}\langle u,f\rangle \in q |\!\} \\ & \cup\, \mathrm{U}\{\!| (fx\,|_i\,gu)\backslash\alpha_1 \ldots \backslash\alpha_n \mid \lambda \in L, \bar{\lambda} \in M, \lambda{:}\langle u,f\rangle \in p, \bar{\lambda}{:}\langle x,g\rangle \in q |\!\} \end{aligned}$$

by the definition of $\backslash\alpha_2, \ldots, \backslash\alpha_n$.

$= \quad p \,\|'_{i+1} q$ by induction hypothesis since $\lambda \in L \wedge \overline{\lambda} \in M \Rightarrow \lambda, \overline{\lambda} \in L.M.$

Hence (1), and so also the theorem by computation induction.

The two definitions of $\|$ are therefore equivalent.

<u>Theorem 1.5.2</u> $p \,\|\, q \;=\; q \,\|\, p$

<u>Proof</u> for p:L and q:M and $L.M = \{\alpha_1, \overline{\alpha}_1, \ldots \alpha_n, \overline{\alpha}_n\}$

$$p \,\|\, q = (p|q)\backslash\alpha_1 \ldots \backslash\alpha_n \text{ by definition of } \|$$
$$= (q|p)\backslash\alpha_1 \ldots \backslash\alpha_n \text{ by Law (F1)}$$
$$= q \,\|\, p\text{, as required by definition of } \| .$$

<u>Theorem 1.5.3</u> for p:L, q:M, r:N where $(L \cap M \cap \overline{N}) \cup (L \cap \overline{M} \cap N) \cup (\overline{L} \cap M \cap N) = \emptyset$

$$p \,\|\, (q \,\|\, r) = (p \,\|\, q) \,\|\, r$$

<u>Proof</u> $p \,\|\, (q \,\|\, r) = (p|(q \,\|\, r))\backslash A$ by definition of $\|$

where $A = \backslash\alpha_1 \ldots \backslash\alpha_n$ and

$\{\alpha_1, \overline{\alpha}_1 \ldots \alpha_n, \overline{\alpha}_n\} = L.(M \cup N-(M.N))$

$= (p|((q|r)\backslash B))\backslash A$ by definition of $\|$

where $B = \backslash\beta_1 \ldots \backslash\beta_m$ and

$\{\beta_1, \overline{\beta}_1 \ldots \beta_m, \overline{\beta}_m\} = M.N$

$= ((p\backslash B)|((q|r)\backslash B))\backslash A$ by Law (F3)

n times since $(M.N) \cap L = \emptyset$

$= (p|(q|r))\backslash B\backslash A$ by Law (F5)

since $(M.N) \cap L = \emptyset$ again

$= (p|q|r)\backslash B\backslash A$ by Law (F2).

Now

$(p \,\|\, q) \,\|\, r = (p|q|r)\backslash\Delta\backslash\Gamma$ in a similar

manner to the above where

$\Gamma = \backslash\gamma_1 \ldots \backslash\gamma_p$ and

$\{\gamma_1, \overline{\gamma}_1, \ldots \gamma_p, \overline{\gamma}_p\} = N.(L \cup M-(L.M))$,

and $\Delta = \backslash\delta_1 \ldots \backslash\delta_q$ where $\{\delta_1, \overline{\delta}_1, \ldots \delta_q, \overline{\delta}_q\} = L.M.$

Now $\backslash\Delta\backslash\Gamma = \backslash\Sigma$ where $\Sigma = \backslash\varepsilon_1 \ldots \backslash\varepsilon_r$ and

$$\{\varepsilon_1, \overline{\varepsilon}_1, \ldots \varepsilon_r, \overline{\varepsilon}_r\} = (N.(L \cup M - (L.M))) \cup L.M$$

$$= (M \cap \overline{N}) \cup (\overline{M} \cap N) \cup (L \cap \overline{M}) \cup (L \cap \overline{N}) \cup (\overline{L} \cap M) \cup (\overline{L} \cap N)$$

by restrictions on the sorts L, M and N

$$= L.(M \cup N - (M.N)) \cup M.N$$

by restrictions again.

Hence $\backslash\Delta\backslash\Gamma = \backslash\Sigma = \backslash B \backslash A$, thus

$(p \parallel q) \parallel r = p \parallel (q \parallel r)$ as required .

Theorem 4.4.1 (Composition)

for $F(p) = \{\!\{ f(x) \mid \phi_1(x), x \in p \}\!\}$ and $G(q) = \{\!\{ g(y) \mid \phi_2(y), y \in q \}\!\}$

then $G o F(p) = \{\!\{ g o f(x) \mid \phi_2(f(x)) \wedge \phi_1(x), x \in p \}\!\}$

Proof we require the following lemma

Lemma 4.4.2 If $S = RC\{f(x) \mid x \in RC(X), P(x) \sqsubseteq true\}$

and $S' = RC\{f(x) \mid x \in X, P(x) \sqsubseteq true\}$

then $S = S'$

Proof in two parts:

$S \supseteq S'$ $RC(X) \supseteq X$ by definition of RC

hence $Y \supseteq Y'$ where $S = RC(Y)$ and $S' = RC(Y')$

by set-theoretic $\{-|-\}$,

hence $S \supseteq S'$ as RC preserves the property $\supseteq$.

$S \subseteq S'$ Suppose that $y \in S$. Then $y \sqsupseteq f(x)$ for some $x \in RC(X)$ such that $P(x) \sqsubseteq true$. As $x \sqsupseteq x' \in X$ by definition of RC then by monotonicity $y \sqsupseteq f(x')$ and $P(x') \sqsubseteq true$. Hence $y \in S'$, and therefore $y \in S \Rightarrow y \in S'$, i.e. $S \subseteq S'$. Hence Lemma.

Returning to the proof of Theorem 4.4.1 we have that $(G o f)(p) = G(F(p))$

$$G(F(p)) = RC\{g(y) \mid y \in RC\{f(x) \mid x \in p, \phi_1(x) \sqsubseteq true\}, \phi_2(y) \sqsubseteq true\}$$

by definition of F and G

$$= RC(z) \text{ by Lemma 4.4.2, where}$$

$$z = \{g(y) \mid y \in \{f(x) \mid x \in p, \phi_1(x) \sqsubseteq true\}, \phi_2(y) \sqsubseteq true\}$$

and $z = \{g(f(x)) | x \in p, \phi_1(x) \sqsubseteq \text{true}, \phi_2(f(x)) \sqsubseteq \text{true}\}$

by normal set-theoretic properties.

Now $\phi_1(x) \sqsubseteq \text{true}$ and $\phi_2(y) \sqsubseteq \text{true}$ iff $\phi_1(x) \wedge \phi_2(y) \sqsubseteq \text{true}$ for various alternatives of $\wedge$.

For instance

$\wedge$	$\perp$	f	t
$\perp$	$\perp$	f	$\perp$
f	f	f	f
t	$\perp$	f	t

or

$\wedge$	$\perp$	f	t
$\perp$	$\perp$	?	$\perp$
f	?	?	?
t	$\perp$	?	t

where t is true and f is false.

This gives us that

$z = \{(g \circ f)(x) | x \in p, \phi_1(x) \wedge \phi_2(f(x)) \sqsubseteq \text{true}\}$

hence $G \circ F(p) = \mathcal{RC}(z) = \{\!\{ g \circ f(x) | \phi_1(x) \wedge \phi_2(f(x)), x \in p \}\!\}$

as required .

Theorem 4.4.3 (Composition)

for $F(p) = U\{\!\{ f(x) | \phi_1(x), x \in p \}\!\}$

and $G(q) = \{\!\{ g(y) | \phi_2(y), y \in q \}\!\}$ then,

$G \circ F(p) = U\{\!\{ G \circ f(x) | \phi_1(x), x \in p \}\!\}$

$f: D_L \to P_M$, $g: D_M \to D_N$, $F: L \to M$ and $G: M \to N$;

where $P_R = \wp(D_R)$, for any sort R.

Proof we require two lemmas;

Lemma 4.4.4 $U\mathcal{RC}(Z) = \mathcal{RC}(UZ)$

Proof let $S = U\mathcal{RC}(Z)$ and $S' = \mathcal{RC}(UZ)$ then the proof proceeds in two parts:

$S \subseteq S'$ suppose $y \in S$, then $y \in T$ where $T \supseteq R$

and $R \in Z$, by definition of $\mathcal{RC}$.

Thus $y \in R$ and $y \in Uz$, by definition of U. As $\mathcal{RC}$ preserves $\supseteq$ then $y \in \mathcal{RC}(UZ)$, hence $S \subseteq S'$.

<u>$S \supseteq S'$</u> suppose $y \in S'$, then $y \sqsupseteq x$ where $x \in R$ and $R \in Z$, by definition of $\mathcal{R}C$ and U.

As $y \sqsupseteq x$ then $y \in Y$ where $Y \sqsupseteq R$ (by $\sqsupseteq$), hence $Y \in \mathcal{R}C(Z)$ by definition of $\mathcal{R}C$. Thus $y \in U\mathcal{R}C(Z)$, by definition of U, hence $S \supseteq S'$. Hence lemma as required.

<u>Lemma 4.4.5</u>

Let $S = U\{\mathcal{R}C\{g(y) | y \in f(x), \phi_2(y) \sqsubseteq true\} | x \in X, \phi_1(x) \sqsubseteq true\}$

and $S' = \mathcal{R}C\{g(y) | y \in f(x), x \in X, \phi_1(x) \sqsubseteq true, \phi_2(y) \sqsubseteq true\}$

then $S = S'$.

<u>Proof</u> we use the element wise argument, and proceed in two parts:

<u>$S \subseteq S'$</u> if $m \in S$ then by definition of U,

$m \in \mathcal{R}C\{g(y) | y \in f(x), \phi_2(y) \sqsubseteq true\}$

for some $x \in X$ such that $\phi_1(x) \sqsubseteq true$.

By $\mathcal{R}C$, $m \sqsupseteq g(y)$ for some $y \in f(x)$ and some $x \in X$ such that $\phi_1(x) \sqsubseteq true$ and $\phi_2(y) \sqsubseteq true$. Thus $m \in S'$ and $S \subseteq S'$ as required.

<u>$S \supseteq S'$</u> if $m \in S'$ then $m \sqsupseteq g(y)$ for some x and y such that $x \in X$, $y \in f(x)$, $\phi_1(x) \sqsubseteq true$, $\phi_2(y) \sqsubseteq true$; by the definition of $\mathcal{R}C$.

For such a y, $m \in \mathcal{R}C\{g(y) | y \in f(x), \phi_2(y) \sqsubseteq true\}$, by definition of $\mathcal{R}C$.

Hence $m \in U\{\mathcal{R}C\{g(y) | y \in f(x), \phi_2(y) \sqsubseteq true\} | x \in X, \phi_1(x) \sqsubseteq true\}$ by the definition of U. Thus $m \in S' \Rightarrow m \in S$ and $S \supseteq S'$. Hence lemma .

Returning to the proof of Theorem 4.4.3 we have that

$GoF(p) = \{\!\{ g(y) | \phi_2(y), y \in U\mathcal{R}C\{f(x) | \phi_1(x) \sqsubseteq true, x \in p\} \}\!\}$

by definition of G, F and $\{\!\{-|-\}\!\}$

$= \{\!\{ g(y) | \phi_2(y), y \in \mathcal{R}C(U\{f(x) | \phi_1(x) \sqsubseteq true, x \in p\} \}\!\}$

by Lemma 4.4.4

$= \{\!\{ g(y) | \phi_2(y), y \in U\{f(x) | \phi_1(x) \sqsubseteq true, x \in p\} \}\!\}$

by Lemma 4.4.5

$= \mathcal{R}C\{g(y) | \bigwedge \phi_2 \sqsubseteq true, \phi_1(x) \sqsubseteq true, y \in f(x), x \in p\}$

by definition of $\{\!\{-|-\}\!\}$ and set-theoretic properties

$= \mathcal{RC}(\mathcal{RC}\{g(y)|\phi_2(y) \sqsubseteq \text{true}, \phi_1(x) \sqsubseteq \text{true}, y \in f(x), x \in p\})$

since $\mathcal{RC} \circ \mathcal{RC}(X) = \mathcal{RC}(X)$ by definition of $\mathcal{RC}$

$= \mathcal{RC}(\cup\{\mathcal{RC}\{g(y)|y \in f(x), \phi_2(y) \sqsubseteq \text{true}\}|x \in X, \phi_1(x) \sqsubseteq \text{true}\})$

by Lemma 4.4.5

$= \mathcal{RC}(\cup\{\{\!\{g(y)|\phi_2(y), y \in f(x)\}\!\}|x \in X, \phi_1(x) \sqsubseteq \text{true}\})$

by definition of $\{\!\{-|-\}\!\}$

$= \mathcal{RC}(\cup\{G \circ f(x)|x \in X, \phi_1(x) \sqsubseteq \text{true}\})$

by definition of G

$= \cup\mathcal{RC}\{G \circ f(x)|x \in X, \phi_1(x) \sqsubseteq \text{true}\}$

by Lemma 4.4.4

$= \cup\{\!\{G \circ f(x)|\phi_1(x), x \in X\}\!\}$

by definition of $\{\!\{-|-\}\!\}$, as required .

<u>Theorem 4.4.6 (Composition)</u>

for $F(p) = \cup\{\!\{f(x)|\phi_1(x), x \in p\}\!\}$

and $G(q) = \{\!\{g(y)|\phi_2(y), y \in q\}\!\}$ then,

$F \circ G(q) = \cup\{\!\{f \circ g(y)|\phi_2(y) \wedge \phi_1(g(y)), y \in q\}\!\}$

where $f: D_L \rightarrow P_M$, $g: D_M \rightarrow D_N$ and $P_R = \mathcal{P}(D_R)$ for any sort R, and $\wedge$ as in Theorem 4.4.1.

<u>Proof</u> Let $F(p) = \cup F'(p)$ where $F'(p) = \{\!\{f(x)|\phi_1(x), x \in p\}\!\}$.

$F \circ G(p) = \cup F' \circ G(p)$

$= \cup F'(G(p))$

$= \cup(F' \circ G(p))$

$= \cup\{\!\{f \circ g(y)|\phi_1(g(y)) \wedge \phi_2(y), y \in q\}\!\}$

by Theorem 4.4.1, as required .

The following lemma allows us to define corollaries to the above composition theorems.

<u>Lemma 4.4.7 (Composition)</u> for $F: S \rightarrow L$, $p: S$ and $q: S$

$F(p \cup q) = F(p) \cup F(q)$

The proof of this lemma follows immediately from the interpretation of our set-forming construct and the usual set-theoretic properties.

<u>Corollary 4.4.8</u>

if $F(p) = \{\!\{ f(x) \mid \phi_1(x), x \in p \}\!\} \cup \{\!\{ f'(x) \mid \phi_1'(x), x \in p \}\!\}$

and $G(q) = \{\!\{ g(y) \mid \phi_2(y), y \in q \}\!\}$ then

$$GoF(p) = \{\!\{ gof(x) \mid \phi_2(f(x)) \wedge \phi_1(x), x \in p \}\!\}$$
$$\cup \{\!\{ gof'(x) \mid \phi_2(f'(x)) \wedge \phi_1'(x), x \in p \}\!\}$$

<u>Proof</u> Let $F(p) = F_1(p) \cup F_2(p)$ where

$F_1(p) = \{\!\{ f(x) \mid \phi_1(x), x \in p \}\!\}$ and

$F_2(p) = \{\!\{ f'(x) \mid \phi_1'(x), x \in p \}\!\}$

$GoF(p) = G(F_1(p) \cup F_2(p))$

$= GoF_1(p) \cup GoF_2(p)$ by Lemma 4.4.7

$= \{\!\{ gof(x) \mid \phi_2(f(x)) \wedge \phi_1(x), x \in p \}\!\}$

$\cup \{\!\{ gof'(x) \mid \phi_2(f'(x)) \wedge \phi_1'(x), x \in p \}\!\}$

by Theorem 4.4.1 as required .

<u>Theorem 7.6.1</u>

$$LINEAR_\alpha \circ LINEAR_\beta = LINEAR_\beta \circ LINEAR_\alpha$$

where $LINEAR_\gamma : L \to L, \gamma \in L, \gamma \neq \gamma', U_\gamma = V_\gamma = 1$ and

$$LINEAR_\gamma(p) = \{\!\{ \gamma : \langle o, K(LIN_\gamma(p)) \rangle \mid \gamma : \langle o, K(p') \rangle \in p \}\!\}$$
$$\cup \{\!\{ \lambda : \langle u, \lambda v.LINEAR_\gamma(fv) \rangle \mid \lambda : \langle u, f \rangle \in p, \lambda \neq \gamma \}\!\}$$

Proof

Show that for all p,

$$(\text{LINEAR}_\alpha \circ \text{LINEAR}_\beta)(p) = (\text{LINEAR}_\beta \circ \text{LINEAR}_\alpha)(p) \tag{1}$$

If $\alpha = \beta$ the result is immediate.

If $\alpha \neq \beta$ (1) follows by computation induction from

$$\forall i \geq o.(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_i)(p) \sqsubseteq (\text{LINEAR}_\beta \circ \text{LINEAR}_\alpha)(p) \tag{2.1}$$

and

$$\forall i \geq o.(\text{LINEAR}_\beta \circ (\text{LINEAR}_\alpha)_i)(p) \sqsubseteq (\text{LINEAR}_\alpha \circ \text{LINEAR}_\beta)(p) \tag{2.2}$$

Proof of (2.1)

Basis

$$\begin{aligned}(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_o)(p) &= \text{LINEAR}_\alpha(\perp)\\ &= \perp, \text{ by the definition of LINEAR}\\ &\sqsubseteq (\text{LINEAR}_\beta \circ \text{LINEAR}_\alpha)(p), \text{ as required}\end{aligned}$$

Induction step

Assume $\forall p.(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_j)(p) \sqsubseteq (\text{LINEAR}_\beta \circ \text{LINEAR}_\alpha)(p)$ (3)

for all $j \leq i$

$$\begin{aligned}&(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_{i+1})(p) =\\ &\quad \{\beta:\langle o,K(\text{LINEAR}_\alpha \circ (\text{LIN}_\beta)_i)(p)\rangle \mid \beta:\langle o,Kp'\rangle \in p\}\\ &\quad \cup \{\alpha:\langle o,K(\text{LIN}_\alpha \circ (\text{LINEAR}_\beta)_i)(p')\rangle \mid \alpha:\langle o,Kp'\rangle \in p\}\\ &\quad \cup \{\lambda:\langle u,\lambda v.(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_i)(fv)\rangle \mid \lambda:\langle u,f\rangle \in p,\\ &\qquad \lambda \neq \alpha \wedge \lambda \neq \beta\}\end{aligned} \tag{4}$$

by composition theorem 4.4.1 and composition lemma 4.4.7.

Now $(\text{LINEAR}_\alpha \circ (\text{LIN}_\beta)_i)(p) = \cup\{(\text{LINEAR}_\alpha \circ (\text{LINEAR}_\beta)_i)(p') \mid \beta:\langle o,Kp'\rangle \in p\}$

(5)

by composition theorem 4.4.3

while

$$(\mathrm{LIN}_\alpha \circ (\mathrm{LINEAR}_\beta)_i)(p') = \bigcup \{(\mathrm{LINEAR}_\alpha \circ (\mathrm{LINEAR}_\beta)_{i-1})(p'') \mid \alpha:\langle o,Kp''\rangle \in p'\}$$

by composition theorem 4.4.6.

From (4), (5), (6) and induction hypothesis (3),

$$(\mathrm{LINEAR}_\alpha \circ (\mathrm{LINEAR}_\beta)_{i+1})(p) \sqsubseteq$$

$$\{\beta:\langle o,K(\bigcup\{(\mathrm{LINEAR}_\beta \circ \mathrm{LINEAR}_\alpha)(p') \mid \beta:\langle o,Kp'\rangle \in p\})\rangle \mid \beta:\langle o,Kp'\rangle \in p\}$$

$$\cup \{\alpha:\langle o,K(\bigcup\{(\mathrm{LINEAR}_\beta \circ \mathrm{LINEAR}_\alpha)(p'') \mid \alpha:\langle o,Kp''\rangle \in p'\})\rangle \mid \alpha:\langle o,Kp'\rangle \in p\}$$

$$\cup \{\lambda:\langle u,\lambda v.(\mathrm{LINEAR}_\beta \circ \mathrm{LINEAR}_\alpha)(fv)\rangle \mid \lambda:\langle u,f\rangle \in p, \lambda\neq\alpha \wedge \lambda\neq\beta\}$$

$$= (\mathrm{LINEAR}_\beta \circ \mathrm{LINEAR}_\alpha)(p)$$

by composition theorems 4.4.1, 4.4.3, 4.4.6 and composition lemma 4.4.7.

Hence (2.1), and (2.2) follows similarly by symmetry .

Theorem 7.6.2 For processes $p:L$ and $q:M$, where $\alpha \in M, \bar{\alpha} \in L$ and $U_\alpha = V_\alpha = 1$

$$p \parallel q \quad = \quad p \parallel \mathrm{LINEAR}_\alpha(q)$$

Proof Using computational induction on $\parallel$

Basis $p \parallel_o q \; = \; \perp \; = p \parallel_o \mathrm{LINEAR}_\alpha(q)$, as required

Induction step assume that $p \parallel_i q = p \parallel_i \mathrm{LINEAR}_\alpha(q)$ (1)

$p \parallel_{i+1} \mathrm{LINEAR}_\alpha(q) =$

$$\{\!\!\{ \lambda:\langle u,\lambda v.fv \;\|_i \mathrm{LINEAR}_\alpha(q)\rangle \mid \lambda\in L-\overline{M}, \lambda:\langle u,f\rangle\in p \}\!\!\}$$
$$\cup \{\!\!\{ \mu:\langle o,K(p \;\|_i \mathrm{LINEAR}_\alpha(q'))\rangle \mid \mu\in M-\overline{L}\wedge\mu\neq\alpha, \mu:\langle o,K(q')\rangle\in q \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ fv \;\|_i \mathrm{LINEAR}_\alpha(q') \mid \lambda,\overline{\lambda}\in L.M\wedge\lambda\neq\alpha, \overline{\lambda}:\langle u,f\rangle\in p, \lambda:\langle o,K(q')\rangle\in q \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ p' \;\|_i \mathrm{LIN}\;(q) \mid \overline{\alpha}:\langle o,K(p')\rangle\in p, \alpha:\langle o,K(q')\rangle\in q \}\!\!\}$$

by the definition of $\|$ and LINEAR_α and composition theorems

$$= \{\!\!\{ \lambda:\langle u,\lambda v.fv \;\|_i \mathrm{LINEAR}_\alpha(q)\rangle \mid \lambda\in L-\overline{M}, \lambda:\langle u,f\rangle\in p \}\!\!\}$$
$$\cup \{\!\!\{ \mu:\langle o,K(p \;\|_i \mathrm{LINEAR}_\alpha(q'))\rangle \mid \mu\in M-\overline{L}, \mu:\langle o,K(q')\rangle\in q \}\!\!\} \qquad (2)$$
$$\cup \bigcup \{\!\!\{ fv \;\|_i \mathrm{LINEAR}_\alpha(gu) \mid \lambda,\overline{\lambda}\in L.M, \overline{\lambda}:\langle u,f\rangle\in p, \lambda:\langle v,g\rangle\in q \}\!\!\}$$

as $(\mu\in M-\overline{L})\wedge(\mu\neq\alpha) \;=\; (\mu\in M-\overline{L})$, $\overline{\alpha}\in L$,

$$p' \;\|_i \mathrm{LIN}_\alpha(q) = p' \;\|_i \bigcup \{\!\!\{ \mathrm{LINEAR}_\alpha(q') \mid \alpha:\langle o,K(q')\rangle\in q \}\!\!\}$$
$$= \bigcup \{\!\!\{ p' \;\|_i \mathrm{LINEAR}_\alpha(q') \mid \alpha:\langle o,K(q')\rangle\in q \}\!\!\}$$

by using the definition of $\bigcup$,

hence

$$\bigcup \{\!\!\{ p' \;\|_i \mathrm{LIN}_\alpha(q) \mid \overline{\alpha}:\langle o,K(p')\rangle\in p, \alpha:\langle o,K(q')\rangle\in q \}\!\!\}$$
$$= \bigcup \{\!\!\{ p' \;\|_i \mathrm{LINEAR}_\alpha(q') \mid \overline{\alpha}:\langle o,K(p')\rangle\in p, \alpha:\langle o,K(q')\rangle\in q \}\!\!\}$$

by composition theorem 2

$$= \bigcup \{\!\!\{ fv \;\|_i \mathrm{LINEAR}_\alpha(gu) \mid \overline{\alpha}:\langle u,f\rangle\in p, \alpha:\langle v,g\rangle\in q \}\!\!\}$$

using our alternative notation.

Then

$$(2) = \{\!\!\{ \lambda:\langle u,\lambda v.fv \;\|_i q\rangle \mid \lambda\in L-\overline{M}, \lambda:\langle u,f\rangle\in p \}\!\!\}$$
$$\cup \{\!\!\{ \mu:\langle o,K(p \;\|_i q')\rangle \mid \mu\in M-\overline{L}, \mu:\langle o,K(q')\rangle\in q \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ fv \;\|_i gu \mid \lambda,\overline{\lambda}\in L.M, \overline{\lambda}:\langle u,f\rangle\in p, \lambda:\langle v,g\rangle\in q \}\!\!\}$$

by induction hypothesis (1)

$= p \;\|_{i+1} q$, by definition of $\|$, as required .

The following Lemmas are used in the proof of Theorem 9.3.1:

<u>Lemma A1</u> For $s_1 : S_1$ and $s_2 : S_2$

$LIN_{\overline{\alpha}'}(s_1 \& s_2) = LIN_{\overline{\alpha}'}(s_1) \& s_2$

where $\overline{\alpha}' \in S_1$ and $\overline{\alpha}' \notin S_2$

<u>Proof</u> $LIN_{\overline{\alpha}'}(s_1 \& s_2) = \bigcup \{ LINEAR_{\overline{\alpha}'}(s') \mid \overline{\alpha}' : \langle o, K(s') \rangle \in (s_1 \& s_2) \}$

by definition of $LIN_{\overline{\alpha}'}$

$= \bigcup \{ LINEAR_{\overline{\alpha}'}(s_1' \& s_2) \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1 \}$

by definition of $\&$ as $\overline{\alpha}' \notin S_2$

Now $LIN_{\overline{\alpha}'}(s_1) \& s_2 = (\bigcup \{ LINEAR_{\overline{\alpha}'}(s_1') \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1 \}) \& s_2$

by definition of $LIN_{\overline{\alpha}'}$

$= \bigcup \{ LINEAR_{\overline{\alpha}'}(s_1') \& s_2 \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1 \}$

by definition of $\bigcup$

$= \bigcup \{ LINEAR_{\overline{\alpha}'}(s_1' \& s_2) \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1 \}$

by definition of $LINEAR_{\overline{\alpha}'}$ and $\&$ as $\overline{\alpha}' \notin S_2$

$= LIN_{\overline{\alpha}'}(s_1 \& s_2)$ as required .

<u>Lemma A2</u> For $s_2 : S_1$ and $s_2 : S_2$

$LIN_{\overline{\alpha}'}(s_1 \& s_2) = LIN_{\overline{\alpha}'}(s_1) \& LIN_{\overline{\alpha}'}(s_2)$

where $\overline{\alpha}' \in S_1 \cap S_2$.

<u>Proof</u> $LIN_{\overline{\alpha}'}(s_1 \& s_2) = \bigcup \{ LINEAR_{\overline{\alpha}'}(s_1' \& s_2') \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1,$

$\overline{\alpha}' : \langle o, K(s_2') \rangle \in s_2 \}$

by definition of LIN and $\&$, using composition

theorem 1

Now $LIN_{\overline{\alpha}'}(s_1) \& LIN_{\overline{\alpha}'}(s_2) =$

$\bigcup \{ LINEAR_{\overline{\alpha}'}(s_1') \& LINEAR_{\overline{\alpha}'}(s_2') \mid \overline{\alpha}' : \langle o, K(s_1') \rangle \in s_1,$

$\overline{\alpha}' : \langle o, K(s_2') \rangle \in s_2 \}$

by definition of $\bigcup$ and $LIN_{\overline{\alpha}'}$

$= \bigcup \{ LINEAR_{\overline{\alpha}'}(s_1' \& s_2') \mid \overline{\alpha}' : \langle o, K(s_1')\rangle \in s_1,$

$\overline{\alpha}' : \langle o, K(s_2')\rangle \in s_2 \}$

by definition of $LINEAR_{\overline{\alpha}'}$ and $\&$ when $\overline{\alpha}' \in S_1 \cap S_2$

$= LIN_{\overline{\alpha}'}(s_1 \& s_2)$ as required .

Theorem 9.3.1

$$FILTER\ S_2 \circ FILTER\ S_1 = FILTER\ (S_2 \& S_1)$$

Proof

We proceed in two parts, due to the definition of FILTER:

Part 1 $\tau \notin L$ prove $\forall p : L \cup A^*,\ S_1 : \overline{B}',\ S_2 : \overline{D}'$.

$$SCHED_D(SCHED_B(p) \parallel S_1) \parallel S_2 = SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \quad (1)$$

where L contains no marking or scheduling labels, $\tau \notin L$ and $D \cup B \subseteq A$.

To show (1) we prove that

$$\forall p, S_1, S_2 . SCHED_D(SCHED_B(p) \parallel S_1) \parallel S_2 \sqsubseteq SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \quad (2)$$

and

$$\forall p, S_1, S_2 . SCHED_D(SCHED_B(p) \parallel S_1) \parallel S_2 \sqsupseteq SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \quad (3)$$

under our restrictions on sorts as in (1), using our elementwise proof technique.

To prove (2) we prove that

$$\forall p, S_1, S_2 . SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \sqsubseteq SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \quad (4)$$

for all $i \geq o$.

Basis $i = o$; then $SCHED_D(SCHED_B(p) \parallel S_1) \parallel_o S_2 = \perp \sqsubseteq SCHED_{D \cup B}(p) \parallel (S_2 \& S_1)$ as required.

<u>Induction step</u> For this it is sufficient to show that

$$\forall p, S_1, S_2 . m \in SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \Rightarrow m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \qquad (5)$$

for some i>o such that

$$SCHED_D(SCHED_B(p) \parallel S_1) \parallel_j S_2) \sqsubseteq SCHED_{D \cup B}(p) \parallel (S_2 \& S_1) \qquad (6)$$

where $j<i$

By the definition of $\parallel$, $m \in SCHED_{D \cup B}(p) \parallel (S_2 \& S_1)$ in two ways.

<u>Case 1</u> $\alpha : \langle u, \lambda v.p' \rangle \in p$, where $\alpha \notin D^* \cup B^*$ (7)

hence $\alpha : \langle u, \lambda v.SCHED_{D \cup B}(p') \rangle \in SCHED_{D \cup B}(p)$

by definition of SCHED

and $m \in \{ \alpha : \langle u, \lambda v.SCHED_{D \cup B}(p') \parallel (S_2 \& S_1) \rangle \mid \alpha : \langle u, \lambda v.SCHED_{D \cup B}(p') \rangle \in SCHED_{D \cup B}(p) \}$ (8)

by definition of $\parallel$

so either $SCHED_{D \cup B}(p) = \perp$, in which case $p = \perp$, thus

$SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 = \perp$, and

$m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2$ as required .

or

$$m \sqsupseteq \alpha : \langle u, \lambda v.SCHED_{D \cup B}(p') \parallel (S_2 \& S_1) \rangle \qquad (9)$$

From (7) since $\alpha \notin B^*$,

$$\alpha : \langle u, \lambda v.SCHED_B(p') \rangle \in SCHED_B(p) \qquad (10)$$

by definition of SCHED

and as $\alpha \in L$ then $\overline{\alpha} \notin \overline{B}'$,

$$\alpha : \langle u, \lambda v.SCHED_B(p') \parallel S_1 \rangle \in SCHED_B(p) \parallel S_1 \qquad (11)$$

by definition of $\parallel$

similarly

$$\alpha : \langle u, \lambda v.SCHED_D(SCHED_B(p') \parallel S_1) \parallel_{i-1} S_2 \rangle \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \qquad (12)$$

by definition of SCHED and $\parallel$

hence

$$\alpha:\langle u, \lambda v.SCHED_{D\cup B}(p') \parallel (S_2 \& S_1)\rangle \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \qquad (13)$$

by induction hypothesis (6), monotonicity (M) and right closure (ℓRC).

Thus from (9) and (13) by ℓRC,

$$m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2) \text{ as required .}$$

<u>Case 2</u> $\alpha^*: \langle o, K(p')\rangle \in p$, where $\alpha^* \in D^* \cup B^*$ (14)

Thus be definition of $\parallel$ and SCHED,

$$m \in \{\alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_{D\cup B}(p'))\rangle\} \parallel (S_2 \& S_1)\rangle \mid \alpha^* \in D^* \cup B^*, \quad \alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_{D\cup B}(p'))\rangle\}\rangle \in SCHED_{D\cup B}(p)\} \qquad (15)$$

so either $SCHED_{D\cup B}(p) = \perp$, and by definition of SCHED, $p = \perp$ and by the reasoning as used in Case 1,

$m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2$ as required

or $m \sqsupseteq \alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_{D\cup B}(p'))\rangle\} \parallel (S_2 \& S_1)\rangle$ (16)

and we have the following two cases:

<u>Case 2.1</u> $\overline{\alpha}' \notin label(S_2 \& S_1)$, hence from (16) and $\parallel$, (17)

$m \sqsupseteq \alpha^*:\langle o, K\,\emptyset\rangle$ (18)

As $\alpha^* \in D \cup B$ the $\overline{\alpha}' \in \overline{D}' \cup \overline{B}'$. By definition of $\&$ and (17) we have three cases:

<u>Case 2.1.1</u> $\overline{\alpha}' \in \overline{B}' - \overline{D}'$ and $\overline{\alpha}' \notin label\ (S_1)$ (19)

By (14) and the definition of SCHED,

$$\alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_B(p'))\rangle\}\rangle \in SCHED_B(p) \qquad (20)$$

as $\alpha^* \in B^* - D^*$

hence, from (19) and (20)

$\alpha^*:\langle o, K\emptyset\rangle \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2$

by the definition of SCHED and $\parallel$ (21)

and from (18) and (21) by RC,

$$m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \text{ as required .}$$

Case 2.1.2 $\overline{\alpha}' \in \overline{B}' - \overline{D}'$ and $\overline{\alpha}' \notin label(S_2)$. This is similar to Case 2.1.1 and is omitted.

Case 2.1.3 $\overline{\alpha}' \in \overline{B}' \cap \overline{D}'$ and $\overline{\alpha}' \notin label(S_2) \cap label(S_1)$. This gives us that either a) $\overline{\alpha}' \notin label(S_2) \wedge \alpha' \in label(S_1)$, or b) $\overline{\alpha}' \notin label(S_1) \wedge \alpha' \in label(S_2)$, or c) $\overline{\alpha}' \notin label(S_1) \wedge \alpha' \notin label(S_2)$. In each of these we have a proof similar to Case 2.1.1 and these are omitted.

Case 2.2 $\overline{\alpha}' \in label(S_2 \& S_1)$ and from the definition of $\&$ we have three cases:

Case 2.2.1 $\overline{\alpha}' : \langle o, K(S_1') \rangle \in S_1$ and $\overline{\alpha}' \in \overline{B}' - \overline{D}'$ (22)

thus $\overline{\alpha}' : \langle o, K(S_2 \& S_1') \rangle \in S_2 \& S_1$, and so

$$\overline{\alpha}' : \langle o, K(LIN_{\overline{\alpha}'}(S_2 \& S_1)) \rangle \in LINEAR_{\overline{\alpha}'}(S_2 \& S_1).$$

By our definition of LINEAR we have that $\overline{\alpha}' : \langle o, K(LIN_{\overline{\alpha}'}(S_2 \& S_1)) \rangle$ is the only member of $LINEAR_{\overline{\alpha}'}(S_2 \& S_1)$ labelled by $\overline{\alpha}'$. (23)

Now

$$\alpha^* : \langle o, K(\{\!\{ \alpha' : \langle o, K(SCHED_{B \cup D}(p')) \rangle \}\!\} \parallel (S_2 \& S_1)) \rangle$$

$$= \alpha^* : \langle o, K(\{\!\{ \alpha' : \langle o, K(SCHED_{B \cup D}(p')) \rangle \}\!\} \parallel LINEAR_{\overline{\alpha}'}(S_2 \& S_1)) \rangle$$

by Theorem 7.6.3

$$= \alpha^* : \langle o, K(SCHED_{B \cup D}(p') \parallel LIN_{\overline{\alpha}'}(S_2 \& S_1)) \rangle \quad (24)$$

by definition of and (23)

hence from (16) and (24)

$$m \sqsupseteq \alpha^* : \langle o, K(SCHED_{B \cup D}(p') \parallel LIN_{\overline{\alpha}'}(S_2 \& S_1)) \rangle \quad (25)$$

From (14), since $\alpha^* \in B^*$

$$\alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_B(p'))\rangle\}\rangle \in SCHED_B(p) \qquad (26)$$

and as $\bar{\alpha}':\langle o,K(S_1')\rangle \in S_1$ then $\bar{\alpha}':\langle o,K(LIN_{\bar{\alpha}'}(S_1))\rangle \in LINEAR_{\bar{\alpha}'}(S_1)$ (27)

From (26) and (27)

$$\alpha^*:\langle o,K(SCHED_B(p') \parallel LIN_{\bar{\alpha}'}(S_1))\rangle \in SCHED_B(p) \parallel S_1 \qquad (28)$$

by Theorem 7.6.3

As $\alpha^* \notin D^*$,

$$\alpha^*:\langle o,K(SCHED_D(SCHED_B(p')) \parallel LIN_{\bar{\alpha}'}(S_1)) \parallel_{i-1} S_2\rangle \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \qquad (29)$$

by (28), SCHED and $\parallel$.

Hence from (29)

$$\alpha^*:\langle o,K(SCHED_{D\cup B}(p') \parallel (LIN_{\bar{\alpha}'}(S_1) \& S_2))\rangle \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2 \qquad (30)$$

by induction hypothesis (6), RC and monotonicity.

Thus from (25), (30), Lemma A1 and RC, as $\bar{\alpha}' \notin \bar{D}'$

$m \in SCHED_D(SCHED_B(p) \parallel S_1) \parallel_i S_2$, as required .

<u>Case 2.2.2</u> $\bar{\alpha}':\langle o,K(S_2')\rangle \in S_2$, $\bar{\alpha}' \in \bar{D}'-\bar{B}'$, $\alpha^* \in D^*-B^*$.

This case is similar to 2.2.1 and is omitted.

<u>Case 2.2.3</u> $\bar{\alpha}':\langle o,K(S_1')\rangle \in S_1$,

$\bar{\alpha}':\langle o,K(S_2')\rangle \in S_2$ and $\bar{\alpha}' \in \bar{B}_1' \cap \bar{B}_2'$.

This case is again similar to that of 2.2.1, except that Lemma A2 is used in place of Lemma A1. Hence Case 2, and also (5). Thus (2) by computational induction.

To prove (3) we prove that

$$\forall p,S_1,S_2.SCHED_{D\cup B}(p) \parallel_i (S_2 \& S_1) \sqsubseteq SCHED_D(SCHED_B(p) \parallel S_1) \parallel S_2 \qquad (31)$$

for all $i \geq o$. (3) follows by computational induction technique.

<u>Basis</u> i=o, then

$$\mathrm{SCHED}_{D\cup B}(p) \parallel_o (S_2 \& S_1) = \bot \sqsubseteq \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) \parallel S_2$$

as required .

<u>Induction step</u> For this it is sufficient to show that

$$\forall p, S_1, S_2 . m \in \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) \parallel S_2 \Rightarrow m \in \mathrm{SCHED}_{D\cup B}(p) \parallel_i (S_2 \& S_1) \quad (32)$$

for some $i>o$ such that

$$\mathrm{SCHED}_{D\cup B}(p) \parallel_j (S_2 \& S_1) \sqsubseteq \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) \parallel S_2 \quad (33)$$

for $j<i$.

$m \in \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) \parallel S_2$ in <u>one</u> way, that is

$\alpha : \langle u, \lambda v.q \rangle \in \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1)$ and so

$m \in \{\alpha : \langle u, \lambda v.q \parallel S_2 \rangle \mid \alpha : \langle u, \lambda v.q \rangle \in \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1)\}$

Now either $\mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) = \bot$. Thus $\mathrm{SCHED}_B(p) \parallel S_1 = \bot$

and either $p = \bot$ or $S_1 = \bot$. In either case $\mathrm{SCHED}_{D\cup B}(p) \parallel (S_1 \& S_2) = \bot$

and by ⓇC, $m \in \mathrm{SCHED}_{D\cup B}(p) \parallel_i (S_1 \& S_2)$ as required.

$$\text{or } m \sqsupseteq \alpha : \langle u, \lambda v.q \parallel S_2 \rangle \quad \text{where} \quad (34)$$

$$\alpha : \langle u, \lambda v.q \rangle \in \mathrm{SCHED}_D(\mathrm{SCHED}_B(p) \parallel S_1) \quad (35)$$

(35) can arise in two ways:

<u>Case 1</u> $\alpha : \langle u, \lambda v.r \rangle \in \mathrm{SCHED}_B(p) \parallel S_1$ and $\alpha \notin B_2^*$ (36)

and $\alpha : \langle u, \lambda v.q \rangle \sqsupseteq \alpha : \langle u, \lambda v.\mathrm{SCHED}_D(r) \rangle$ (37)

By definition of $\parallel$, (36) can only arise if

$\alpha : \langle u, \lambda v.t \rangle \in \mathrm{SCHED}_B(p)$ and (38)

$\alpha : \langle u, \lambda v.r \rangle \sqsupseteq \alpha : \langle u, \lambda v.t \parallel S_1 \rangle$ (39)

(38) can arise in two ways, via the definition of SCHED:

<u>Case 1.1</u> $\alpha : \langle u, \lambda v.p' \rangle \in p, \alpha \notin B_1^*$ (40)

and $\alpha : \langle u, \lambda v.t \rangle \sqsupseteq \alpha : \langle u, \lambda v.\mathrm{SCHED}_B(p') \rangle$ (41)

From (34), (37), (39), (41) and monotonicity

$$m \sqsupseteq \alpha:\langle u, \lambda v.SCHED_D(SCHED_B(p') \parallel S_1) \parallel S_2\rangle \tag{42}$$

From (36), (40) and the definition of SCHED

$$\alpha:\langle u, \lambda v.SCHED_{D\cup B}(p')\rangle \in SCHED_{D\cup B}(p), \tag{43}$$

and as $\overline{\alpha} \notin \overline{B} \cup \overline{D}$ then

$$\alpha:\langle u, \lambda v.SCHED_{D\cup B}(p') \parallel_{i-1}(S_2 \& S_1)\rangle \in SCHED_{D\cup B}(p) \parallel_i (S_2 \& S_1) \tag{44}$$

by the definition of $\parallel$.

Thus from (44) using the induction hypothesis and RC

$\alpha:\langle u, \lambda v.SCHED_D(SCHED_B(p') \parallel S_1) \parallel S_2\rangle \in SCHED_{D\cup B}(p) \parallel_i (S_2 \& S_1)$ and

from (42), $m \in SCHED_{D\cup B}(p) \parallel_i (S_2 \& S_1)$, as required.

<u>Case 1.2</u> $\alpha^*:\langle o, K(p')\rangle \in p$ and $\alpha^* \in B^*$ (45)

and $\alpha:\langle u, \lambda v.t\rangle \sqsupseteq \alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_B(p'))\rangle\}\rangle$ (46)

By (39) and (46)

$$\alpha:\langle u, \lambda v.r\rangle \sqsupseteq \alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_B(p'))\rangle\} \parallel S_1\rangle \tag{47}$$

and <u>either</u> $\overline{\alpha}' \in label(S_1)$ <u>or</u> not.

<u>Case 1.2.1</u> $\overline{\alpha}' \notin label(S_1)$ (48)

From (47) and the definition of $\parallel$,

$$\alpha:\langle u, \lambda v.r\rangle \sqsupseteq \alpha^*:\langle o, K(\emptyset)\rangle \tag{49}$$

and from (34) and (37)

$$m \sqsupseteq \alpha^*:\langle o, K(\emptyset)\rangle \tag{50}$$

and from (45), $\alpha^*:\langle o, K\{\alpha':\langle o, K(SCHED_{D\cup B}(p'))\rangle\}\rangle \in SCHED_{D\cup B}(p)$ (51)

As $\overline{\alpha}' \in \overline{B}' - \overline{D}'$ and $\overline{\alpha}' \notin label(S_1)$, then $\overline{\alpha}' \notin label(S_2 \& S_1)$.

Thus

$$\alpha^*:\langle o, K(\emptyset)\rangle \in SCHED_{D\cup B}(p) \parallel_i (S_2 \& S_1) \tag{52}$$

from (51) and $\parallel$.

Hence from (50), (52) and ℝC;

$m \in SCHED_{D \cup B}(p) \;\|_i (S_2 \& S_1)$, as required.

Case 1.2.2 $\bar{\alpha}' \in label(S_1)$ and therefore

$$\bar{\alpha}' : \langle o, K(S_1') \rangle \in S_1 \qquad (53)$$

and by definition of LINEAR,

$$\bar{\alpha}' : \langle o, K(LIN_{\bar{\alpha}'}(S_1)) \rangle \in LINEAR_{\bar{\alpha}'}(S_1) \text{ and} \qquad (54)$$

this is the only $\bar{\alpha}'$ labelled synchronisation in $LINEAR_{\bar{\alpha}'}(S_1)$.

From (47), (54) and Theorem 7.6.3,

$$\alpha : \langle u, \lambda v.r \rangle \sqsupseteq \alpha^* : \langle o, K(SCHED_B(p') \| LIN_{\bar{\alpha}'}(S_1)) \rangle \qquad (55)$$

and from (34), (35) and (55)

$$m \sqsupseteq \alpha^* : \langle o, K(SCHED_D(SCHED_B(p') \| LIN_{\bar{\alpha}'}(S_1)) \| S_2) \rangle \qquad (56)$$

by monotonicity.

$$\text{As } \alpha^* \in B^* - D^*,\ \bar{\alpha}' : \langle o, K(S_2 \& S_1') \rangle \in (S_2 \& S_1) \qquad (57)$$

$$\text{hence } \bar{\alpha}' : \langle o, K(LIN_{\bar{\alpha}'}(S_2 \& S_1')) \rangle \in LINEAR_{\bar{\alpha}'}(S_2 \& S_1) \qquad (58)$$

by the definition of $LINEAR_{\bar{\alpha}'}$

From (45) and the definition of SCHED,

$$\alpha^* : \langle o, K\{\alpha' : \langle o, K(SCHED_{D \cup B}(p')) \rangle\} \rangle \in SCHED_{D \cup B}(p)$$

and from (57), (58), $\|$ and Theorem 7.6.3

$$\alpha^* : \langle o, K(SCHED_{D \cup B}(p') \;\|_{i-2} (LIN_{\bar{\alpha}'}(S_2 \& S_1'))) \rangle \in SCHED_{D \cup B}(p) \;\|_i (S_2 \& S_1) \qquad (59)$$

From (56) by induction hypothesis, monotonicity and ℝC

$$m \sqsupseteq \alpha^* : \langle o, K(SCHED_{D \cup B}(p') \;\|_{i-2} (LIN_{\bar{\alpha}'}(S_1) \& S_2)) \rangle$$

and from this, (59), Lemma A1 and ℝC

$m \in SCHED_{D \cup B}(p) \;\|_i (S_2 \& S_1)$ as required.

Case 2 $\alpha^*:\langle o,K(r)\rangle \in SCHED_B(p) \| S_1$, and $\alpha^* \in B_2{}^*$ (60)

and $\alpha:\langle u, \lambda v.q\rangle \sqsupseteq \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_D(r))\rangle\}\rangle$ (61)

(60) can arise only from:

$\alpha^*:\langle o,K(t)\rangle \in SCHED_B(p)$ and (62)

$\alpha^*:\langle o,K(r)\rangle \sqsupseteq \alpha^*:\langle o,K(t \| S_1)\rangle$ (63)

(62) can arise in two ways:

Case 2.1 $\alpha^*:\langle o,K(p')\rangle \in p$, $\alpha^* \notin B_1{}^*$ (64)

and then $\alpha^*:\langle o,K(t)\rangle \sqsupseteq \alpha^*:\langle o,K(SCHED_B(p'))\rangle$ (65)

From (61), (63), (65), (34)

$m \sqsupseteq \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_D(SCHED_B(p') \| S_1))\rangle\} \| S_2\rangle$ (67)

either $\bar{\alpha}' \in label(S_2)$ or not

Case 2.1.1 $\bar{\alpha}' \in label(S_2)$

and then $m \sqsupseteq \alpha^*:\langle o,K(\phi)\rangle$, from (67) (68)

From (60), (64) and as $\bar{\alpha}' \in \bar{B}_2 \setminus \bar{B}_1{}'$ and $\bar{\alpha}' \notin label(S_2)$

$\bar{\alpha}' \notin label\ (S_2 \& S_1)$ (69)

From (60), (64) and the definition of SCH D,

$\alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_{D\cup B}(p'))\rangle\}\rangle \in SCHED_{D\cup B}(p)$ (70)

and $\alpha^*:\langle o,K(\phi)\rangle \in SCHED_{D\cup B}(p) \|_i (S_2 \& S_1)$ (71)

from (69), (70) and $\|$.

Hence from (68), (71) and RC; $m \in SCHED_{D\cup B}(p) \|_i (S_2 \& S_1)$ as required.

Case 2.1.2 $\bar{\alpha}' \in label(S_2)$, hence $\bar{\alpha}':\langle o,K(S_2{}')\rangle \in S_2$ (72)

By definition of $LINEAR_{\bar{\alpha}'}$, then

$\overline{\alpha}':\langle o,K(LIN_{\overline{\alpha}'}(S_2))\rangle \in LINEAR_{\overline{\alpha}'}(S_2)$ and is

the only synchronisation in $LINEAR_{\overline{\alpha}'}(S_2)$ labelled by $\overline{\alpha}'$. (73)

By Theorem 7.6.3,

$\alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_D(SCHED_B(p') \| S_1)\rangle\} \| S_2\rangle$

$= \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_D(SCHED_B(p') \| S_1)\rangle\} \| LINEAR_{\overline{\alpha}'}(S_2)\rangle$ (74)

and from (67), (73) and (74)

$$m \sqsupseteq \alpha^*:\langle o,K(SCHED_D(SCHED_B(p') \| S_1) \| LIN_{\overline{\alpha}'}(S_2))\rangle \quad (75)$$

From (60), (64), $\overline{\alpha}' \in \overline{D}'\ \overline{B}'$ and by definition of $\&$

$$\overline{\alpha}':\langle o,K(S_2'\&S_1)\rangle \in (S_2\&S_1) \text{ and} \quad (76)$$

$$\overline{\alpha}':\langle o,K(LIN_{\overline{\alpha}'}(S_2\&S_1))\rangle \in LINEAR_{\overline{\alpha}'}(S_2\&S_1) \quad (77)$$

From (60), (64)

$$\alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_{D\cup B}(p'))\rangle\}\rangle \in SCHED_{D\cup B}(p) \quad (78)$$

Hence by (76), (77), (78), and Theorem 7.6.3

$$\alpha^*:\langle o,K(SCHED_{D\cup B}(p') \|_{i-2}(LIN_{\overline{\alpha}'}(S_2\&S_1))\rangle \in SCHED_{D\cup B}(p) \|_i(S_1\&S_2) \quad (79)$$

From (75) by monotonicity, induction hypothesis and RC

$$m \sqsupseteq \alpha^*:\langle o,K(SCHED_{D\cup B}(p') \|_{i-2}(LIN_{\overline{\alpha}'}(S_2)\&S_1))\rangle \quad (80)$$

Hence from (79), (80), Lemma A1 and RC

$$m \in SCHED_{D\cup B}(p) \|_i(S_2\&S_1), \text{ as required .}$$

<u>Case 2.2</u> $\alpha^*:\langle o,K(p')\rangle \in p, \alpha^* \in B^*$ (81)

and $\alpha^*:\langle o,K(t)\rangle \sqsupseteq \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_B(p'))\rangle\}\rangle$. (82)

From (63) and (82)

$$\alpha^*:\langle o,K(r)\rangle \sqsupseteq \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_B(p')\rangle\} \| S_1\rangle \quad (83)$$

by monotonicity.

We have <u>either</u> that $\overline{\alpha}' \in label\ (S_1)$ <u>or</u> not

<u>Case 2.2.1</u> $\bar{\alpha}' \notin label(S_1)$, which is similar to Case 1.2.1 and is omitted.

<u>Case 2.2.2</u> $\bar{\alpha}' \in label(S_1)$ and then

$$\bar{\alpha}':\langle o,K(S_1')\rangle \in S_1 \tag{84}$$

By the definition of LINEAR and from (84),

$$\bar{\alpha}':\langle o,K(LIN_{\bar{\alpha}'}(S_1))\rangle \in LINEAR_{\bar{\alpha}'}(S_1) \tag{85}$$

and this is the only synchronisation so labelled in $LINEAR_{\bar{\alpha}'}(S_1)$.
From (83) and (85) using Theorem 7.6.3 and the definition of $\|$;

$$m \sqsupseteq \alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_D(SCHED_B(p') \| LIN_{\bar{\alpha}'}(S_1)))\rangle\} \| S_2\rangle \tag{86}$$

and <u>either</u> $\bar{\alpha}' \in label(S_2)$ <u>or</u> not.

<u>Case 2.2.2.1</u> $\bar{\alpha}' \notin label(S_2)$ and this case is similar to Case 2.1.1 and is omitted.

<u>Case 2.2.2.2</u> $\bar{\alpha}' \in label(S_2)$ and then

$$\bar{\alpha}':\langle o,K(S_2')\rangle \in S_2 \text{ and} \tag{87}$$

from the definition of LINEAR ;

$\bar{\alpha}':\langle o,K(LIN_{\bar{\alpha}'}(S_2))\rangle \in LINEAR_{\bar{\alpha}'}(S_2)$ and this is the only synchronisation so labelled in $LINEAR_{\bar{\alpha}'}(S_2)$. (88)

From (86) and (88) using Theorem 7.6.3 and the definition of $\|$;

$$m \sqsupseteq \alpha^*:\langle o,K(SCHED_D(SCHED_B(p') \| LIN_{\bar{\alpha}'}(S_1)) \| LIN_{\bar{\alpha}'}(S_2))\rangle \tag{89}$$

and by the induction hypothesis, monotonicity and RC;

$$m \sqsupseteq \alpha^*:\langle o,K(SCHED_{D \cup B}(p') \|_{i-2}(LIN_{\bar{\alpha}'}(S_1) \& LIN_{\bar{\alpha}'}(S_2)))\rangle \tag{90}$$

As $\bar{\alpha}' \in \bar{B}' \cap \bar{D}'$, and from (84) and (87);

$\bar{\alpha}':\langle o,K(S_2' \& S_1')\rangle \in (S_2 \& S_1)$ and by the definition of LINEAR ;

$$\overline{\alpha}':\langle o,K(LIN(S_2\&S_1))\rangle\in LINEAR_{\overline{\alpha}'}(S_2\&S_1) \quad . \qquad (91)$$

From (81) and SCHED;

$$\alpha^*:\langle o,K\{\alpha':\langle o,K(SCHED_{D\cup B}(p'))\rangle\}\rangle\in SCHED_{D\cup B}(p) \quad . \qquad (92)$$

From (92), (91), Theorem 7.6.2 and the definition of $\|$;

$$\alpha^*:\langle o,K(SCHED_{D\cup B}(p')\ \|_{i-2} LIN(S_2\&S_1))\rangle\in SCHED_{D\cup B}(p)\ \|_i(S_2\&S_1) \qquad (93)$$

Hence from (90), (93), Lemma A2 and RC;

$m\in SCHED_{D\cup B}(p)\ \|_i(S_2\&S_1)$, as required .

This gives us (3), and as we also have (2), Part 1 follows.

<u>Part 2</u> we prove $\forall p:L\cup A^*, S_1:\overline{B'}, S_2:\overline{D'}$ where L contains no scheduling or marking labels, $\tau\in L$ and $D\cup B\subseteq A$;

$$SCHED_D(SCHED_B(p)\nparallel S_1)\nparallel S_2 = SCHED_{D\cup B}(p)\nparallel(S_2\&S_1)$$

First we note that $\tau\notin D\cup B$, and schedulers never terminate (1)

$SCHED_D(SCHED_B(p)\nparallel S_1)\nparallel S_2 =$

$SCHED_D(((SCHED_B(p))\{\tau_1/\tau\}\|S_1)\mid J_{12})\backslash\tau_1\backslash\tau_2 \nparallel S_2$

by Lemma 6.2.1 and (1)

$= SCHED_D(((SCHED_B(p\{\tau_1/\tau\})\ \|\ S_1)\mid J_{12})\backslash\tau_1\backslash\tau_2)\nparallel S_2$

by definition of $\{\alpha/\beta\}$ and SCHED

$= (((SCHED_D(((SCHED_B(p\{\tau_1/\tau\})\ \|\ S_1)\mid J_{12})\backslash\tau_1\backslash\tau_2))\{\tau_3/\tau\}\ \|\ S_2)\mid$

$J_{34})\backslash\tau_3\backslash\tau_4$ by Lemma 6.2.1 and (1)

$= ((SCHED_D((SCHED_B(p\{\tau_1/\tau\})\ \|\ S_1)\mid J_{12}\{\tau_3/\tau\})\ \|\ S_2)\mid J_{34})\backslash$

$\tau_3\backslash\tau_4\backslash\tau_1\backslash\tau_2$ by definition of $\{\alpha/\beta\}$ and law F5 .

$= ((SCHED_D(SCHED_B(p\{\tau_1/\tau\})\ \|\ S_1)\ \|\ S_2)\mid(J_{12}\{\tau_3/\tau\})\mid J_{34})\backslash$

$\tau_3\backslash\tau_4\backslash\tau_1\backslash\tau_2$ by definition of SCHED and $\mid$

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid (J_{12}\{\tau_3/\tau\}) \mid J_{34}) \backslash$
$\tau_3 \backslash \tau_4 \backslash \tau_1 \backslash \tau_2$ by Part 1

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid ((J_{12}\{\tau_3/\tau\} \mid J_{34}) \backslash \tau_3 \backslash \tau_4)) \backslash \tau_1 \backslash \tau_2$
by Laws F3, F4 and F5

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid K_{12}) \backslash \tau_1 \backslash \tau_2$
by the definition of J, $\mid$ and $\backslash\alpha$ where
$K_{12} = \{\!\{ \bar{\tau}_1 : \langle o, K \{\!\{ \bar{\tau}_2 : \langle o, K\, \phi \rangle \}\!\} \rangle,$
$\bar{\tau}_2 : \langle o, K \{\!\{ \bar{\tau}_1 : \langle o, K\, \phi \rangle \}\!\} \rangle \}\!\}$

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid (K_{12} \backslash \tau_2)) \backslash \tau_1$
by laws F3, F4 and F5

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid \{\!\{ \bar{\tau}_1 : \langle o, K\phi \rangle \}\!\}) \backslash \tau_1$ (2)
by definition of $\backslash \tau_2$.

Now

$SCHED_{D\cup B}(p) \not\parallel (S_2 \& S_1) =$
$((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid J_{12}) \backslash \tau_1 \backslash \tau_2$
by the Lemma 6.2.1, SCHED, $\{\alpha/\beta\}$ and (1)

$= ((SCHED_{D\cup B}(p\{\tau_1/\tau\}) \parallel (S_2 \& S_1)) \mid \{\!\{ \bar{\tau}_1 : \langle o, K\phi \rangle \}\!\}) \backslash \tau_1$
by definition of J_{12}, $\backslash \tau_2$ and laws F3, F4 and F5

= (2), as required .

<u>Theorem 9.3.4</u>

$(FILTER\ s\ p) \parallel q = FILTER\ s(p \parallel q)$

where $p{:}L\cup A^*$, $s{:}\bar{B}'$, $q{:}M$, $B \subseteq A$, $A^* \cap M = \phi$ and $\tau \notin L \cup M$.

Proof

$$(\text{FILTER s p}) \parallel q = (\text{SCHED}_B(p) \parallel s) \parallel q$$

by definition of FILTER

$$= \text{SCHED}_B(p) \parallel q \parallel s$$

by associativity and commutivity of $\parallel$

and

$$\text{FILTER } s(p \parallel q) = \text{SCHED}_B(p \parallel q) \parallel s$$

by definition of FILTER

We therefore prove that

$$\text{SCHED}_B(p) \parallel q = \text{SCHED}_B(p \parallel q) \qquad (1)$$

under the sort restrictions above.

We prove (1) by computational induction on the definition of $\parallel$.

That is, we prove that

$$\forall i \geq o. \text{SCHED}_B(p) \parallel_i q = \text{SCHED}_B(p \parallel_i q) \qquad (2)$$

Basis

$$\text{SCHED}_B(p) \parallel_o q = \perp = \text{SCHED}_B(p \parallel_o q)$$

as SCHED is strict from its definition .

Induction step prove that

$$\text{SCHED}_B(p) \parallel_i q = \text{SCHED}_B(p \parallel_i q),\ i > o \qquad (3)$$

given that

$$\text{SCHED}_B(p) \parallel_j q = \text{SCHED}_B(p \parallel_j q),\ j < i \qquad (4)$$

under the sort restrictions above.

$$\text{SCHED}_B(p \parallel_i q) =$$

$$\{\!\{ \lambda : \langle u, \lambda v.\text{SCHED}_B(fv \parallel_{i-1} q)\rangle \mid \lambda : \langle u, f\rangle \in p, \lambda \notin B^* \}\!\}$$

$$\cup \{\!\{ \lambda^* : \langle o, K\{\!\{ \lambda' : \langle o, K(\text{SCHED}_B(p' \parallel_{i-1} q))\rangle \}\!\} \rangle \mid \lambda^* : \langle o, K(p')\rangle \in p, \lambda^* \in B^* \}\!\}$$

$$\cup \{\!\{ \lambda : \langle v, \lambda u.\text{SCHED}_B(p \parallel_{i-1} gu)\rangle \mid \lambda : \langle v, g\rangle \in q, \lambda \in M \cup N^* - (\overline{L \cup A^*}) \}\!\}$$

$$\cup \bigcup \{\!\{ \text{SCHED}_B(fv \parallel_{i-1} gu) \mid \lambda : \langle u, f\rangle \in p, \overline{\lambda} : \langle v, g\rangle \in q, \lambda, \overline{\lambda} \in (M \cup N^*).(L \cup A^*) \}\!\}$$

by the definition of SCHED, $\parallel$ and composition lemmas and theorems.

$= \{\!\!\{ \lambda:\langle u, \lambda v.SCHED_B(fv) \parallel_{i-1} q\rangle \mid \lambda:\langle u,f\rangle \in p, \lambda \notin B^* \}\!\!\}$

$\cup \{\!\!\{ \lambda^*:\langle o, K\{\!\!\{\lambda':\langle o, K(SCHED_B(p') \parallel_{i-1} q)\rangle\}\!\!\}\rangle \mid \lambda^*:\langle o, K(\cdot p')\rangle \in p,$

$\lambda^* \in B^* \}\!\!\}$

$\cup \{\!\!\{ \lambda:\langle v, \lambda u.SCHED_B(p) \parallel_{i-1} gu)\rangle \mid \lambda:\langle v,g\rangle \in q, \lambda \in M \cup N^* - (\overline{L \cup A^*}) \}\!\!\}$

$\cup \bigcup \{\!\!\{ SCHED_B(fv) \parallel_{i-1} gu \mid \lambda:\langle u,f\rangle \in p, \overline{\lambda}:\langle v,g\rangle \in q, \lambda, \overline{\lambda} \in (M \cup N^*).(L \cup A^*) \}\!\!\}$

by induction hypothesis (4)

$= \quad SCHED_B(p) \parallel_i q$

by the definition of SCHED, $\parallel$ and composition lemmas and theorems .

Lemma 9.5.2

$$SCHED_B(p) \nparallel MONITOR_{A^*} = DASH(p) \nparallel MONITOR_{(A-B)'},$$

where $p:L \cup A^*$, $B \subseteq A$, $\Upsilon \in L$ and L contains no marking or scheduling labels.

Proof Notice that:

$SCHED_B(p) \nparallel MONITOR_{A^*} = ((SCHED_B(p\{\Upsilon_1/\Upsilon\}) \parallel MONITOR_{A^*}) \mid J_{12}) \backslash \Upsilon_1 \backslash \Upsilon_2$

by Lemma 6.2.1, the definitions of SCHED and $\{\alpha/\beta\}$ and that $MONITOR_{A^*}$ does not terminate,

and

$DASH(p) \nparallel MONITOR_{(A-B)'} = ((DASH(p\{\Upsilon_1/\Upsilon\}) \parallel MONITOR_{(A-B)'}) \mid J_{12}) \backslash \Upsilon_1 \backslash \Upsilon_2$

again by Lemma 6.2.1, the definitions of DASH and $\{\alpha/\beta\}$ and using that $MONITOR_{(A-B)'}$ does not terminate.

It is therefore sufficient to prove that

$$SCHED_B(p\{\Upsilon_1/\Upsilon\}) \parallel MONITOR_{A^*} = DASH(p\{\Upsilon_1/\Upsilon\}) \parallel MONITOR_{(A-B)'} \qquad (1)$$

We thus prove that

$$\mathrm{SCHED}_B(q) \parallel \mathrm{MONITOR}_{A*} = \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'} \qquad (2)$$

where $q{:}M$ and $\Upsilon \notin M$.

We prove (2) by showing that

$$\mathrm{SCHED}_B(q) \parallel \mathrm{MONITOR}_{A*} \sqsubseteq \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'} \qquad (3)$$

and

$$\mathrm{SCHED}_B(q) \parallel \mathrm{MONITOR}_{A*} \sqsupseteq \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'} \qquad (4)$$

We do not prove the equality (2) directly by computation inducation as the two $\parallel$ combinators iterate at different rates with respect to $\mathrm{SCHED}_B/\mathrm{MONITOR}_{A*}$ and $\mathrm{DASH}(p)/\mathrm{MONITOR}_{(A-B)'}$.

To prove (3) use computation induction. We show that

$$\forall i \geq o.\mathrm{SCHED}_B(q) \parallel_i \mathrm{MONITOR}_{A*} \sqsubseteq \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'} \qquad (5)$$

<u>Basis</u> $\mathrm{SCHED}_B(q) \parallel_o \mathrm{MONITOR}_{A*} = \perp \sqsubseteq \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'}$

as required.

<u>Induction step</u> Assume that for all $j \leq i$

$$\mathrm{SCHED}_B(q) \parallel_j \mathrm{MONITOR}_{A*} \sqsubseteq \mathrm{DASH}(q) \parallel \mathrm{MONITOR}_{(A-B)'} \qquad (6)$$

then

$$\mathrm{SCHED}_B(q) \parallel_{i+1} \mathrm{MONITOR}_{A*} =$$

$$\{\!\{ \alpha : \langle u, \lambda v.\mathrm{SCHED}_B(fv) \parallel_i \mathrm{MONITOR}_{A*} \rangle \mid \alpha : \langle u,f \rangle \in q, \alpha \notin A* \}\!\}$$
$$\cup \bigcup \{\!\{ \mathrm{SCHED}_B(fv) \parallel_i \mathrm{MONITOR}_{A*} \mid \alpha : \langle u,f \rangle \in q, \alpha \in (A-B)* \}\!\}$$
$$\cup \{\!\{ \alpha' : \langle o, K(\mathrm{SCHED}_B(q') \parallel_{i-1} \mathrm{MONITOR}_{A*}) \rangle \mid \alpha* : \langle o, K(q') \rangle \in q,$$
$$\alpha* \in B* \}\!\}$$

by the definition of SCHED_B, $\mathrm{MONITOR}_{A*}$, $\parallel$ and composition theorems

$$\sqsubseteq \{\alpha:\langle u,\lambda v.\mathrm{DASH}(fv)\parallel \mathrm{MONITOR}_{(A-B)'}\rangle \mid \alpha:\langle u,f\rangle\in q, \alpha\notin A^* \}$$
$$\cup \bigcup\{\mathrm{DASH}(fv)\parallel \mathrm{MONITOR}_{(A-B)'} \mid \alpha:\langle u,f\rangle\in q, \alpha\in(A-B)^* \}$$
$$\cup\{\alpha':\langle o,K(\mathrm{DASH}(q')\parallel \mathrm{MONITOR}_{(A-B)'})\rangle \mid \alpha^*:\langle o,K(q')\rangle\in q, \alpha^*\in B^* \}$$

by induction hypothesis (6) and monotonicity

$$= \quad \mathrm{DASH}(p)\parallel \mathrm{MONITOR}_{(A-B)'}$$

by definition of DASH, $\parallel$, $\mathrm{MONITOR}_{(A-B)'}$ and composition theorems.

Hence (5) and then (3) by computation induction.

To prove (4) we show that

$$\forall i\geq o.\mathrm{DASH}(q)\parallel_i \mathrm{MONITOR}_{(A-B)'} \sqsubseteq \mathrm{SCHED}_B(q)\parallel \mathrm{MONITOR}_{A^*} \qquad (7)$$

<u>Basis</u> $\mathrm{DASH}(q)\parallel_o \mathrm{MONITOR}_{(A-B)'} = \perp \sqsubseteq \mathrm{SCHED}_B(q)\parallel \mathrm{MONITOR}_{A^*}$ as required.

<u>Induction step</u> Assume that for all $j\leq i$

$$\mathrm{DASH}(q)\parallel_j \mathrm{MONITOR}_{(A-B)'} \sqsubseteq \mathrm{SCHED}_B(q)\parallel \mathrm{MONITOR}_{A^*} \qquad (8)$$

then

$$\mathrm{DASH}(q)\parallel_{i+1} \mathrm{MONITOR}_{(A-B)'} =$$
$$\{\alpha:\langle u,\lambda v.\mathrm{DASH}(fv)\parallel_i \mathrm{MONITOR}_{(A-B)'}\rangle \mid \alpha:\langle u,f\rangle\, q, \alpha\notin A^* \}$$
$$\cup \bigcup\{\mathrm{DASH}(fv)\parallel_i \mathrm{MONITOR}_{(A-B)'} \mid \alpha:\langle u,f\rangle\in q, \alpha\in(A-B)^* \}$$
$$\cup\{\alpha':\langle o,K(\mathrm{DASH}(q')\parallel_i \mathrm{MONITOR}_{(A-B)'})\rangle \mid \alpha^*:\langle o,K(q')\rangle\in q, \alpha^*\in B^* \}$$

by definition of DASH, $\mathrm{MONITOR}_{(A-B)'}$, $\parallel$ and composition theorems.

$$\sqsubseteq \quad \{\!\!\{ \alpha:\langle u, \lambda v.\mathrm{SCHED}_B(fv) \parallel \mathrm{MONITOR}_{A*}\rangle \mid \alpha:\langle u,f\rangle \in q, \alpha \notin A^* \}\!\!\}$$
$$\cup \bigcup \{\!\!\{ \mathrm{SCHED}_B(fv) \parallel \mathrm{MONITOR}_{A*} \mid \alpha:\langle u,f\rangle \in q, \alpha \in (A-B)^* \}\!\!\}$$
$$\{\!\!\{ \alpha':\langle o, K(\mathrm{SCHED}_B(q') \parallel \mathrm{MONITOR}_{A*})\rangle \mid \alpha^*:\langle o, K(q')\rangle \in q,$$
$$\alpha^* \in B^* \}\!\!\}$$

by induction hypothesis (8) and monotonicity

$$= \quad \mathrm{SCHED}_B(p) \parallel \mathrm{MONITOR}_{A*}$$

by the definition of SCHED_B, $\mathrm{MONITOR}_{A*}$, $\parallel$ and composition theorems.

Hence (7) and then (4), thus (2) and so the lemma holds.

REFERENCES

[Bek] H. Bekic, "Towards a Mathematical Theory of Processes", Technical Report TR25.125, IBM Laboratory, Vienna, 1971.

[Bri 1] P. Brinch Hansen, "Operating System Principles", Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Bri 2] P. Brinch Hansen, "Distributed Actions; A concurrent programming concept", University of Southern California, April 1977.

[Cam] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronisation by Path Expressions", Lecture Notes in Computer Science, 16, Springer-Verlag, 1974.

[Cou] P.J. Courtois, F. Heymans and D.L. Parnas, "Concurrent Control with 'readers' and 'writers'", Comm. ACM, 14, 10, 1971.

[Den] J.B. Dennis and J.B. Fosseen, "Introduction to data flow schemas", Computation Structures Group, Memo 81-1, Project MAC, MIT, 1973.

[Dig] PDP11 Peripherals and Interfacing Handbook, Digital Equipment Corporation, 1971.

[Dij 1] E.W. Dijkstra, "Cooperating Sequential Processes", in Programming Languages, F. Gennys (ed.), Academic Press, New York, 1968.

[Dij 2] E.W. Dijkstra, "Hierarchial Ordering of Sequential Processes", in Operating System Techniques, Hoare and Perrot (eds.), Academic Press, London, 1972.

[Flo] R.W. Floyd, "Assigning Meaning to Programs", in Proc. of Symposium in Applied Mathematics, Vol. 19, J.T. Schwartz (ed.), American Mathematical Society, Providence, Rhode Island, 1967.

[Gog] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, "Initial Algebra Semantics and Continuous Algebras", Journal ACM 24, 1, Jan. 1977.

[Gre 1] I. Greif, "Semantics of Communicating Parallel Processes", Thesis TR-154, Project MAC, MIT, 1975.

[Gre 2] I. Greif, "On Proofs of Programs for Synchronisation", Proc. 3rd International Colloquium on Automata,Languages and Programming, Edinburgh, Edinburgh University Press, 1976.

[Hab] A.N. Habermann, "Path Expressions", Report of Computer Science Dept., Carnegie-Mellon University, Pittsburgh, June 1975.

[Hew] C. Hewitt, "Protection and Synchronisation in Actor Systems", Working Paper 83, Artificial Intelligence Lab., MIT, 1974.

[Hoa 1] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Comm. ACM, 12, 10. 1969.

[Hoa 2] C.A.R. Hoare, "Towards a Theory of Parallel Programming", in Operating System Techniques, Hoare and Perrot (eds.), Academic Press, London, 1972.

[Hoa 3] C.A.R. Hoare, "Communicating Sequential Processes", Unpublished memorandum, Computer Science Dept., Queens University, Belfast, Dec. 1976.

[Kah 1] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", Proc. of IFIP Congress 74, North-Holland, 1974.

[Kah 2] G. Kahn and D. MacQueen, "Co-routines and Networks of Parallel Processes", Research Report 202, IRIA, Paris, 1976.

[Man] Z. Manna, Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

[McC] J. McCarthy, "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Systems, Braffort and Hirschberg (eds.), North-Holland, 1963.

[Mil 1] G.J. Milne and A.J.R.G. Milner, "Concurrent processes and their syntax", Report CSR-2-77, Computer Science Dept., University of Edinburgh, 1977.

[Mil 2] R.E. Milne, "The formal semantics of computer languages and their implementations", Thesis, University of Cambridge, 1974.

[Mil 3] A.J.R.G. Milner, "Processes, a Mathematical Model of Computing Agents", Proc. Logic Colloquium, Bristol, North-Holland, 1973.

[Mil 4] A.J.R.G. Milner, "Flowgraphs and Flow Algebras", Report CSR-5-77, Computer Science Dept., University of Edinburgh, 1977.

[Owi 1] S. Owicki, "An Axiomatic Proof Technique for Parallel Programs II: shared data abstractions", Draft, Stanford University, Aug. 1976.

[Owi 2] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica, 6, 1976.

[Par] D. Park, "Fixpoint induction and proofs of program properties", Machine Intelligence, Vol. 5, Meltzer and Mickie (eds.), Edinburgh University Press, 1970.

[Plo 1] G. Plotkin, "A Powerdomain Construction", SIAM Journal on Computing, 5, 3, 1976.

[Plo 2] G. Plotkin, "Sketch of semantics for the class of programs introduced in a paper of Lauer and Campbell", Unpublished report, Dept. of Artificial Intelligence, University of Edinburgh, 1976.

[Sco 1] D. Scott, "Lattice Theoretic Models for various type-free calculi", Proc. 4th International Congress in Logic, Methodology and the Philosophy of Science, Bucharest, 1972.

[Sco 2] D. Scott and C. Strachey, "Towards a Mathematical Semantics for Computer Languages", Proc. Symposium on Computers and Automata, Microwave Res. Inst. Symposia series, Vol. 21, Polytechnic Institute of Brooklyn, 1971.

[Str] C. Strachey and C.P. Wadsworth, "Continuations - A mathematical semantics for handling full jumps", Monograph PRG-11, Programming Research Group, Oxford Univ. Computing Lab., 1974.

[Smy] M. Smyth, "Powerdomains", Internal Report, Computer Science Dept., University of Warwick, Coventry, 1976.

[Wir 1] N. Wirth, "MODULA: A language for modular multiprogramming", Research report 18, Institute für Informatik, Zurich, 1975.

[Wir 2] N. Wirth, "The use of MODULA and Design and Implementation of MODULA", Research report 19, Institute für Informatik, Zurich, 1975.

[Yoe] M. Yoeli, "Petri Nets and Asynchronous Control Networks", Research report CS-73-07, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, Canada, 1973.