The equivalence of an operational and a denotational semantics for pure dataflow

by

Antony Azio Faustini

A dissertation submitted for the degree of Doctor of Philosophy.

Department of Computer Science
University of Warwick
Coventry
UK

April 1982

Contents

Abstract

In this thesis we prove the equivalence of an operational and a denotational semantics for pure dataflow.

The term pure dataflow refers to dataflow nets in which the nodes are functional (i.e. the output history is a function of the input history only) and the arcs are unbounded fifo queues.

Gilles Kahn gave a method for the representation of a pure dataflow net as a set of equations; one equation for each arc in the net. Kahn stated, and we prove, that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the net's associated set of equations.

In our model we do not require that nodes be sequential nor deterministic, not even the functional nodes. As a consequence our model has a claim of being completely general. In particular our nets have what we call the encapsulation property in that any subnet can be replaced in any pure dataflow context by a node having exactly the same input/output behaviour. Our model is also complete in the sense that our nodes have what we call the universality property, that is, for any continuous history function there exists a node that will compute it.

The proof of the Kahn Principle given in this thesis makes use of infinite games of perfect information. Infinite games turn out to be an extremely useful tool for defining and proving results about operational semantics. We use infinite games to give for the first time a completely general definition of subnet functionality. In addition their use in certain proofs is effective in reducing notational complextity.

Finally we look at possible ways of extending Kahn's denotational model by the introduction of pause objects called hiatons. Finally we describe interesting ways of refining our operational model.

Acknowledgements

I would like to thank my supervisor, Bill Wadge for his continuous interest and encouragement and for numerous discussions that provided the stimulation for this research.

I would also like to thank all the members, past and present, of the Warwick dataflow group who have devoted many hours of their time to help me with this research. Special thanks must go to Stephen Matthews for proof reading.

In addition, the financial support of the Science Research Council is gratefully acknowledged.

Finally, I would like to thank my wife for her patience, moral support and for the preperation of the diagrams that appear in the text.

Chapter 0

Introduction

In this thesis we present for the first time, a proof, in a completely general context, of the equivalence of an operational and a denotational semantics for pure dataflow.

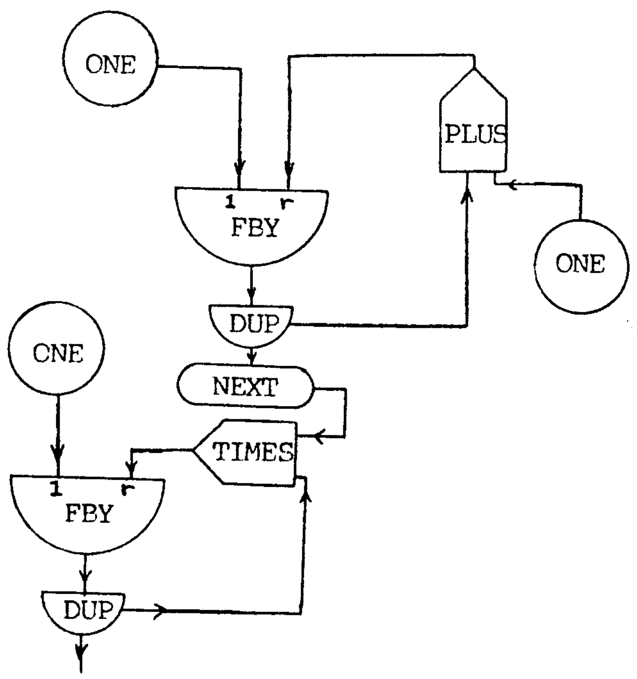
Dataflow is currently an active area of research, with teams of researchers working on dataflow machines (Gurd and Watson[23], Davis[16],Misunas[39],Dennis[20]...); dataflow programming languages (CAJOLE[24],ID[51],VAL[1],..) and Software design (Bic[34], Yourdon and Constantine[52], Cameron[14] ...). However, the idea of data flowing between concurrently executing processes is an old one, dating back nearly twenty years to a paper written by Conway[15]. In this paper Conway describes how it is possible to divide certain processing activity into a number of autonomous modules. Conway writes "A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items; (2) the flow of each of these items is along fixed, one-way paths; ....". Conway even predicted the advantages of dataflow in truly distributed systems. He writes "When true parallel processors are available the fact that the coroutines of a separable program may be executed simultaneously becomes even more significant".

Recent years have seen an unprecedented interested in parallel (distributed) computing and the dataflow concept, thought of so many years ago, is proving to be extremely fruitful. As a result of this widespread interest there has emerged a number of different dataflow models.

In this thesis we are interested in a model of dataflow in which the computing stations are autonomous machines and the fixed unidirectional communication lines are unbounded fifo queues along which discrete items of data flow. A discrete item of data can be any datum such as a natural number or a matrix of natural numbers or a real number or a set of real numbers. We shall refer to these discrete units of data as datons. The asynchronous computing stations in our model can be thought of as continuously operating 'black boxes' that consume datons one by one from each of their input arcs and after some internal computation output datons one by one on each of their output arcs. Simple dataflow nodes usually produce and consume datons at the same rate, however the more complex nodes may produce outputs at a different rate to that at which they consume inputs. The only way in which our nodes may communicate with another is by sending datons along the fixed arcs which interconnect the producer and consumer nodes. We can think of an arc as a 'pipe' along which a producer node dispatches datons one by one to some consumer node, the producer having no knowledge of who the consumer is and vice versa. This is analagous to the way in which two UNIX processes, connected by a UNIX pipe, communicate. In our model if an observer is placed on one of the 'pipes' he is able to record its entire activity, a possibly infinite sequence of datons called the history of the pipe. The model of dataflow we have just described has been referred to as pipeline dataflow or stream flow and it has been studied by many including Adams[2], Kahn[26], Karp & Miller[28], Arnold[4], Arvind & Gostelow[3] and McIlroy[37]. Although McIlroy has not published many papers he has been influential in the incorporation of pipeline dataflow into UNIX [53].

An example of a dataflow model which is not pipeline dataflow is that of Kowsinski[31]. In this model the datons are tagged and so the order in which they arrive at a computing station is unimportant, all that has to be done is to match on tags. Arvind and Gostelow[3] have also investigated such a tagged interpreter. They have shown in a simple context that the tagged model is sometimes able to compute more than the pipeline model. (i.e it doesn't have to process its inputs using a fifo ordering). The tagged model has also been used very successfully, by the dataflow team of Gurd and Watson[23], as the basis of a dataflow machine.

However in this thesis we are exclusively interested in pipeline dataflow. Figure A shows a simple example of a dataflow net which produces as output the infinite sequence 1,2,6,24,120,..... of factorials.



The node labelled TIMES repeatedly awaits the arrival of a daton on both its input arcs and as soon as both datons arrive they are consumed and a daton representing their product is output. The node labelled PLUS processes similarly except that it outputs the sum of the incoming datons.

The node labelled ONE is a constant node (having no input arcs) that produces as output the infinite sequence 1,1,1,1,......

The nodes we have just described are all examples of nodes that process their inputs, however not all of our nodes process their inputs. The following nodes, also used in Figure A, are examples of nodes that manipulate their inputs. The node labelled NEXT awaits the arrival of its first input daton and as soon as it arrives it is consumed. Thereafter the node repeatedly awaits of the arrival of inputs and as soon as a daton is available it is consumed and a copy is produced as output. The node labelled FBY (followed by) awaits for the first daton to arrive through the input port labelled 'l' and as soon as this daton arrives it is consumed and a copy of the daton is produced as the node's first output. Thereafter the node repeatedly awaits the arrival of datons through the input port labelled 'r' and as soon as a daton arrives it is comsumed and a copy of the daton is produced as output. The node labelled DUP (duplicator) repeatedly awaits the arrival of datons on its input arc and as soon as a daton arrives it is consumed and a copy is sent along both output arcs.

These are by no means the only nodes we are interested in but even with this small selection it is possible to describe interesting dataflow nets.

Pure Dataflow

All the nodes described in the previous examples have one property in common, namely they all compute functions from the histories of their input lines to the histories of their output lines. For example the NEXT node computes the history function:

next: $\langle a_0,a_1,a_2,\ldots\rangle \longmapsto \langle a_1,a_2,\ldots\rangle$

e.g. if the sequence <1,2,3,4,....> is the entire history of the NEXT nodes's input line then <2,3,4,....> will be the entire history of the NEXT node's output line. In a similar way the node labelled PLUS computes the history function:

plus: $(\langle a_0,a_1,\ldots\rangle,\langle b_0,b_1,\ldots\rangle) \longmapsto \langle a_0+b_0,a_1+b_1,\ldots\rangle$

and the FBY node computes:

fby: $(\langle a_0,a_1,\ldots\rangle,\langle b_0,b_1,b_2,\ldots\rangle) \longmapsto \langle a_0,b_0,b_1,b_2,\ldots\rangle$

Although all the nodes we have described so far are functional (i.e. compute history functions) this is not true for all 'pipeline' dataflow nodes. A classic example of a non-functional node is the MERGE node. In its simplest form the node passes on down its only output arc the first daton to appear on either of its input arcs. However if datons are available on both input arcs one arc is choosen at random and the first daton in its associated queue is output. The MERGE node is interesting because it exhibits two properties not possessed by functional nodes. One of the properties is time-sensitivity, that is, the rate of input of datons effects more than just the the rate of output; it effects what is output. A functional node may allow the input rate to effect the output rate; but it can never allow the input rate to determine what is output. The other property possesed by MERGE is internal randomness.

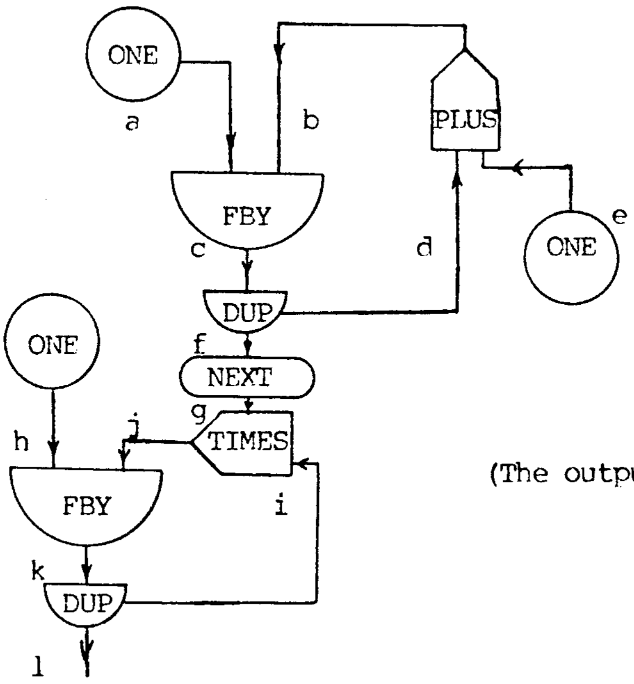
This property tells use that even if there is no time-sensitive behaviour (e.g. MERGE begins computation with all the inputs it is ever to receive queued on its input arcs) the inputs will still be merged in a non-deterministic manner. It may even be the case that one of the inputs is eventually ignored completely, thus the merge may be unfair. Although simple nodes such as PLUS are deterministic, in that there is no choice in how to compute, the more complex functional nodes do permit a choice. This choice, unlike the random choice of the MERGE node, is usually based on a strategy that ensures that the node computes the required function. If a random strategy were used by these complex functional nodes then the only possible adverse effect would be that the entire output history would be an initial segment of that produced by the correct strategy.

Functionality is extremely important because it allows us to associate simple mathematical objects with complex operational entities. In particular it allows us to associate with each arc in a net a countable sequence of datons (the history of the arc) and with each node a history function which describes the relationship between a node's entire input/output activity.

In this thesis we shall use the term pure dataflow to describe dataflow nets in which all the nodes are functional. Gilles Kahn[26] was the first to study pure dataflow and he pointed out that a pure dataflow net can be represented by a set of equations. Kahn stated that given a pure dataflow net, such as that in figure B, it is possible to assign a variable to each arc in the net, and to each node it is possible to assign a history function computed by the node.

The set of equations representing a net are generated by associating an equation with each output arc in the net. For example the output arc of node NEXT in figure B (labelled by the letter g) is the result of applying 'next' (the history function computed by NEXT) to NEXT's single input arc (labelled by the letter f). This generates the equation:

$g = \mathrm{next}(f)$



$$\left.\begin{aligned}
a &= 1\\
b &= \mathrm{plus}(d,e)\\
c &= \mathrm{fby}(a,b)\\
d &= \mathrm{dup}_1(c)\\
e &= 1\\
f &= \mathrm{dup}\ (c)\ (\emptyset)\\
g &= \mathrm{next}(f)\\
h &= 1\\
i &= \mathrm{dup}\ (k)\ (1)\\
j &= \mathrm{multiply}(g,i)\\
k &= \mathrm{fby}(h,j)\\
\text{(The output)} \rightarrow \quad l &= \mathrm{dup}_0(k)
\end{aligned}\right\} \Sigma_B$$

Applying this same rule to each output arc in Figure B we generate the set of equations Σ_B.

It is a well known result that under certain conditions a system of equations such as Σ_B has a least fixed point solution (see |3.B). The principle that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the net's equations we call the <u>Kahn Principle</u>. Although Kahn was the first to realise this principle he never published a formal proof nor did he define precisely the concepts of "node", "net", etc.

One of the main objectives of this thesis is to give a satisfactory operational semantics for pure dataflow and thus give for the first time a satisfactory proof of the Kahn principle.

The Kahn principle has many important consequences, for example, we can use sets of equations (such as Σ_B) as a dataflow programming language. This equational language, which is close to the language Lucid[6], is above all easy to reason about because it is simply mathematics as it stands.

An additional consequence is that we have complementary ways of viewing dataflow. On the one hand we have simple sets of equations, on the other we have complex operational behaviours. The equivalence of the operational and the denotational semantics of pure dataflow has been used by Wadge[46] to give a denotational (extensional) treatment to dataflow deadlock. A similar approach has been used by Pilgram[40] to analyse pure dataflow nets for queueing properties.

The Inadequacy of current models of pure dataflow

An important objective of this thesis is to give a proof of the Kahn principle for an operational semantics which has a claim to being completely general. Current models of pure dataflow restrict themselves to nets in which nodes are sequential and deterministic.

A sequential node is, roughly speaking, one in which a node is either computing or is blocked waiting for input. The nodes we described earlier such as PLUS, FBY and NEXT are all examples of sequential nodes.

A deterministic node is, roughly speaking, one in which there is no choice in behaviour. For example the NEXT node destructively consumes its first input and thereafter passes on any future input. The description of the behaviour of node NEXT given in a previous section allows no choice in the node's behaviour. However the fact that a node is deterministic does not mean that the node is functional. An example of a deterministic non-functional node is the node that outputs a copy of the daton that it has just consumed from the head of the queue associated with the nodes input arc. If the queue associated with the node's input arc is empty the node outputs a zero. The reason that this deterministic node is not functional is that it is time sensitive. On the other hand as we will see later not all non-deterministic nodes are non-functional. In fact there are many useful non-deterministic nodes that are functional. Examples of some of these useful nodes are 'wise' if-then-else, parallel 'or' and parallel 'and' (see ∫2.A for more details).

Since Kahn stated his principle there have been three publications, one by Wiedmer[49], one by Arvind and Gostelow[3], and one by Arnold[4], each of which attempts to give a proof of the Kahn principle. The first of these publications is the paper by Arvind and Gostelow in which they define a select set of primitive functional nodes and consider only nets built up using these primitives. As the primitives are sequential and deterministic they have no difficulty in proving the Kahn principle; for a extremley limited context.

Almost at the same time as Arvind and Gostelow's paper Edwin Wiedmer published, in German, his doctoral dissertation in which a sketch proof is given of the Kahn Principle. In our opinion Wiedmer came closest to giving a completely general proof of the Kahn principle. Unfortunately Wiedmer's dissertation is not in English and so has been ignored by dataflow researchers outside Europe. However a recent translation of some of his thesis has appeared recently and this should make his work more accesible (Wiedmer[48]). Wiedmer follows the advise given by Kahn [26] and gives a formal description of his nodes and nets in terms of Turing machines, interconnected by one way infinite tapes. The reason that we stated that Wiedmer came closest to proving the Kahn principle is that he outlines a definition of functional behaviour which can handle more than sequential and deterministic behaviour. The class of history functions computable by his nodes he calls the 'approximation computable' functions. On the other hand the class of history functions computable by the sequential and deterministic nodes described by Kahn, he calls 'rigidly computable' functions. Wiedmer is on the right track in that he has broadened his view as to what operational behaviour assocaited with a node deserves to be called functional. However, we feel that his operational model is not very natural as it is outlined in terms of Turing machines connected by one way infinite tapes. In addition his proof of the Kahn principle (which is not one of the objectives of his dissertation) is only given in the form of a sketch proof.

The most Recent attempt to prove the Kahn principle appears in a recent paper by Andre Arnold[4]. In this paper Arnold claims to give a satisfactory operational semantics corresponding to Kahn's simple language for parallel processing. However Arnold restricts his operational model to sequential nodes that produce as much output as they consume input. Once again the restriction of the operational model to this limited context means that there is little difficulty in proving the Kahn principle. The inadequacy of Arnold's model is demonstrated by the fact that the simple NEXT described earlier can be realised in Kahn's language but cannot be realised in Arnold's model. (We have more to say about this restriction in∫1.F)

Of the three works mentioned Arnold's is by far the most formal, he gives precise defintions of nodes, nets, node computations, net computations etc... In his operational semantics a node is defined in terms of a sequential transducer of infinite words. Although the transducers described may be non-deterministic, it would seem that in his limited proof of the Kahn principle only deterministic transducers are used.

We argue that previous operational models of dataflow are not truly distributed as underlying the distributed nature of these dataflow models are sequential and deterministic nodes. In addition previous models lack modularity; in that there are history functions computed by nets that cannot be realised by sequential nodes (see ∫2.A). Gilles Kahn[26] pointed out that in pure dataflow top down-design finds mathematical justification. The reason given by Kahn is that the decision to implement a history function by a single process or a network of processes can be delayed without introducing side effects into the overall system design. Any formal operational model of pure dataflow wishing to allow top down-design cannot restrict itself to sequential and deterministic nodes.

Our Operational model and a completely general Proof of the Kahn Principle

In this thesis we prove, for the first time, the Kahn principle in its most general context. In our model we allow functional nodes to be both non-sequential and non-deterministic.

A non-sequential node is, roughly speaking, one which is capable of performing other activities (such as output) while waiting for input - in other words, it is essentially able to do more that one thing at the same time. A very simple example of such a node is the 'double identity' node. This node has two inputs and two outputs and echos the first input on the first output, and the second input on the second output. Such a node cannot be sequential because it cannot allow both outputs to 'run dry' when only one of the inputs does so. This ability to compute while waiting is essential if our model is to be in any sense general.

To give directly a general operational semantics for pure dataflow is extremely difficult. The reason for this is that we require nodes to be completely general and at the same time we require them to be functional. This combination of generality and functionality is very difficult to capture in one step. In this thesis we solve this problem in two steps. As a first step we give a formal operational semantics to the whole of pipeline dataflow. Although this means we include non-functional nodes it does give us the generality we require. The second step is to give a precise definition of what it means for a node to compute a history function. We are then able to use our definition of functionality to select that subset of pipeline dataflow that deserves to be called pure dataflow. Using this approach we are able to describe any functional node no matter how bizzare its behaviour.

Besides Wiedmer others who have studied pure dataflow have given restricted definitions of what it means for a node to compute a history function. Arvind and Gostelow[3] consider only nodes which are all obviously functional, completely avoiding the question. On the other hand Kahn[26] assumes (without proof) that the processes definable in his simple language are all functional. Arnold[4] takes a different approach and associates with each of his nodes a function from the set of possible input histories to the powerset of the possible output histories. In addition he proves that the deterministic nodes compute history functions that are continuous in the sense of Kahn[26].

In this thesis we present for the first time a completely general definition of what it means for a node to compute a history function. Our definition uses a new approach based on infinite games (see ∫3E).

In our model we allow functional nodes to be non-deterministic and non-sequential and as such our model has a claim to being completely general. In particular our nets have what we call the <u>Encapsulation</u> property in that any network of interconnected nodes can be replaced in any dataflow context by a single node having exactly the same input/output behaviour. In other words our model is modular in that it allows top down-design (none of the previous models posseses the modularity property). In addition our model is complete in the sense that any continuous history function can be realised by some node.

As our operational semantics is that of pipeline dataflow and our definition of node functionality is completely general our proof of the Kahn principle has a claim to be in some sense complete.

Using a functional programming approach to extend pure dataflow

The equational dataflow programming language referred to earlier is limited in that the programs are built from a finite set of equations in which the left hand side of every equation is a variable and the right hand side is a finite expression involving variables, constants and history functions. A typical program is:

```
         x = fby (1 , x+1)
         y = next(x)
         z = fby (1 , y*z)
    output = z
```

This set of equations (program) is related to the graph in Figure A. The least fixed point solution to the equations gives:

```
         x is <1,2,3,4,......>
         y is <2,3,4,5,......>
         z is <1,2,6,24,120,....>
    output is <1,2,6,24,120,....>
```

The infinite sequence corresponding to the variable output is exactly the output produced by the net in Figure A.

To allow the user to develop programs in a structured way we extend this 'simple' language by allowing equations defining functions, including recursive definitions. Some typical user defined functions (UDF's) are :

```
first(x)  = fby ( x, first(x) )
f(x)      = if x leq 1 then 1 else x*f(x-1) fi
```

The implementation of this extended language (which is similar to Structured Lucid[6]) involves either dynamically growing nets or (notionally) infinite nets (but still pure datflow). The methods of this thesis extend naturally to such nets and permit us to give, for the first time, a proof of the correspondingly extended Kahn principle.

Further extensions and refinements

In our formal model of pipeline dataflow, a node is defined to be a non-deterministic automaton. In general it is very difficult, simply by looking at the set of transitions associated with a node, to say whether or not that node is functional. As possible refinements to our operational model we look at some intersesting node transition properties which may guarantee node functionality. One of these properties is the simple one step Church-Rosser like property. If a node posseses this property the it is guaranteed to be functional. (note: note all functional nodes have this property). Other interesting cases are based on the transition relation associated with a node being in some sense 'monotonic'.

A more ambitious extension is to extend the denotational semantics to handle a broader class of nodes and nets (i.e not just pure dataflow). One such extension involves changing the basic domain of histories by introducing a special kind of object called a "hiaton" (from "hiatus" meaning a pause; the term is due to W. Wadge and E. A. Ashcroft). A hiaton can be thought of as a unit of delay that (notionally) travels along with the ordinary datons and allows a node to produce something regularly even if it has no real output. Hiatonic streams code up timing information and they can be used to handle nodes and nets which are time senstive. Frederic Boussinot in his recent Doctorat D'Etat, entitled "Reseaux de processus avec melange equitable: une approache du temps reel", presents a denotational semantics based on hiatonics, for an operational semantics which combines the sequential model of Arnold[4] and a fair merge operator. Thus Boussinot's denotational semantics describes a much larger class of operational behaviours than does the denotational semantics of Kahn[26]. Even more recently, David Park[41] has found a denotational semantics that makes use of the hiaton to describe the operational model corresponding to a combination of our model of pure dataflow and merge operators that behave "fairly" (i.e. we call these nets F-nets). The Park Principle states that the operational behaviour of an F-net is exactly described by the de-hiatonised set of solutions associated with the F-net's associated set of hiatonised relations.

Mathematical Notation

In this thesis we use a particularly simple representation of the natural numbers due to Von Neumann, it has not gained complete acceptance but we find it extremely convenient. The number zero is the empty set, the number one is the set { 0 },.. in general the number n is the set { 0,1,2,...,n-1 } of all smaller numbers. This process of constructing numbers goes on endlessly however, for the purposes of this thesis we are interested in only the numbers up to the first infinite ordinal namely:

ω (= { 0,1,2,3,4,5,... }).

Thus ω represents the natural numbers.

Besides the concepts of sets and numbers two other important concepts are those of a relation and a function. A relation R between two sets A and B is represented by a subset (possibly empty) of the set of all ordered pairs (a,b) of elements a and b from A and B respectively. If the relation R is such that

$\forall a \in A \; \exists! \, b \in B \; (a,b) \in R$ then R is said to be a function. Thus a function is a special kind of relation.

The domain of a function f (written dom(f)) is the set of all left hand components of elelments of f i.e $\{ a \mid (a,b) \in f \}$. Similarly the range of a function f (written rg(f)) is the set $\{ b \mid (a,b) \in f \}$. Notice that the empty set is also a function (the empty relation). It is the only function whose domain and range are both empty. Note that here and throughout this thesis we use the conventional set builder notation {...|... }.

Given two sets A and B, the set B^A is the set of all functions from

A to B.

Sequences in our notation are (by definition) simply functions. The finite sequences over a set A (= Sq(A)) have the natural numbers as their domain thus:

$Sq(A) = \{ A^k \mid k \in \omega \}$

The infinite sequences over a set A have the ordinal ω as their domain i.e. A^ω is the set of all infinite sequences over the set A. In this thesis we are especially interested in sequences over the natural numbers in particular we refer frequently to the set of finite and infinite sequences of natural numbers. Thus for notational convenience we define this set to be

Ka $(= Sq(\omega) \cup \omega^\omega)$ Since a sequence is a function the length of a sequence is its domain. For example the sequence

<1,2,7,9> is the function

{ <0,1>,<1,2>,<2,7>,<3,9> }

and its domanin is thus { 0,1,2,3 } which we know to be the natural number 4, the length of the sequence. The elements in the range of a sequence are called the components of the sequence. Since sequences are just functions, the sequence indexing operation is just function application. For example the 10th component of the sequence s is simply s(10). Often it is convenient to use the conventional subscripting s_{10}, we therefore adopt the convention that subscripting denotes function application. However, we still use the functional subscripting since it is useful in avoiding multiple levels of subscripting. For example:

instead of s_{i_j} we write $s(i)_j$

If s is a sequnce and n is a natural number, the function $s|n$ (s restricted to n) is simply the initial segment of s of length n (or simply s itself if n is greater than the length of s).

In writing expressions denoting sequences we will use a sequence builder notation which is similar to the set builder notation except that angular brackets are used instead of curly brackets. Thus $<0,1,2,3>$ is a sequnce of length 4 and $< i \mid i \in \omega >$ is the ordered sequence of natural numbers. This second form of the sequence builder notation is just a variation of λ-notation; the sequence is the value of the λ-expression

$\lambda\, i \in \omega\, .i$

Sometimes we will use the direct form together with triple dots to denote an infinite sequence:

$<2,4,6,8,10,\ldots>$

We are assuming that the first few values given are sufficient to make out the pattern for the rest of the sequence. We also use the triple dot notation for finite sequences

$<s_0,s_1,\ldots,s_{j-1},s_{j+1},\ldots,s_{n-1}>$.

In our notation there is no requirement for a sequence to be represented using angular bracket notation. Often we can refer to a sequence by its name alone. Thus instead of writing

$<s_0,s_1,\ldots>$ or $< s_i \mid i \in \omega >$ we simply write s.

If s is a sequence and i is a natural number, the function $s\downarrow_i$ (s drop i) is simply the sequence s with its i^{th} component dropped. For example $<1,2,3,4>\downarrow_2$ is the sequence $<1,2,4>$.

If d is some expression then the function

$s\uparrow_i d$ (s insert d at i) is simply the sequence s with the value of expression d inserted after the i^{th} component. For example

$<1,2,3,4,5>\uparrow_3 99$ is simply $<1,2,3,4,99,5>$

Note in general

$(s\downarrow_i)\uparrow_{i-1}s_i = (s\uparrow_{i-1}X)\downarrow_i = s$

If s and t are finite sequences then the function

$s^\frown t = \langle s_0,\ldots,s_{n-1},t_0,\ldots,t_{m-1}\rangle$

where $\mathrm{dom}(s) = n$ and $\mathrm{dom}(t) = m$

In addition to the above notion of sequence we add the generalised notion of sequence in which the indexing set is not required to be a natural number or ω. These generalised sequences are called "families" and can be thought of as labelled sets. It should however be apparent that a family is nothing more than a simple function, the domain of which is the indexing set. In this thesis we use the notion of a "family" in conjunction with subscripting.

Let f be a family over elements of a set D indexed by the set I

(in other words $f \in D^I$)

Let $i \in I$ and $d \in D$

the function

$f/_i$ (f drop i) is simply the set $\{ (x,y) \mid (x,y) \in f \text{ and } x \neq i \}$

the function

$f/_i d$ (f add (i,d)) is simply the set $\{ (i,d) \} \cup f/_i$

Note

$(f/_i)/_i f(i) = (f/_i)/_i f(i) = f$

To aid the reader bear in mind the nature of particular objects we introduce some notatonal conventions.

We will generally use the variables:

i,j,k,n,m for natural numbers.

s,t,u,v,w,... for finite sequences.

$\alpha,\beta,\delta,\ldots$ for infinite sequences

A,B,C,D,... for sets

Chapter 1

A Review of Related Work

In this chapter we shall briefly describe what we consider to be important models of pipeline dataflow. One of the main aims of this chapter is to assess to what extent the various models relate to the following topics:

i) A completely formal operational semantics for pipeline dataflow.

ii) A completely general definition of what it means for a pipeline dataflow node to compute a history function.

iii) A proof of the Kahn principle in a completely general context.

A. Computation Graphs

The first paper to attempt to formalise pipeline dataflow was a paper by Karp and Miller[28] in which they refer to dataflow nets as computation graphs. A computation graph is defined in terms of a directed graph the arcs of which are unbounded FIFO queues and the nodes of which are determinate computing stations.

In this graph based model of computation a node is an operator that computes after a pre-determined number of operands (i.e. datons) have arrived along the nodes input arcs. Computation involves the removal of the operands and the production of a number of datons on the operators output arcs. In the networks described by Karp and Miller each node computes only a finite number of times. Nodes that compute ad infinitum are considered to be in some sense faulty. This design decision is not surprising since at the time that Karp and Miller developed their computation graphs conventional programs which failed to terminate were considered to be incorrect; correctness being defined as partial correctness plus termination.

A node in the Karp and Miller model is associated with a single valued function which determines for each computation step the relationship between a node's inputs and outputs. A node with n-input arcs and m-output arcs is associated with an input/output function of the following form:

$$f : \omega^{I(0)} \times \ldots \times \omega^{I(n-1)} \rightarrow \omega^{O(0)} \times \ldots \times \omega^{O(m-1)}$$

where	$i \in n$	the natural number $I(i)$ denotes the number of datons required by input arc i for each computation step.
	$j \in m$	the natural number $O(j)$ denotes the number of datons produced by output arc j for each computation step.

(Remember that throughout this thesis we are assuming that datons are of type natural number).

Thus in the Karp and Miller model a node is a machine that repeatedly computes its associated input/output function. As long as the node is supplied with the appropriate inputs the node will produce the required outputs thus the node is able to compute ad infinitum if given inputs ad infinitum. However in all the examples given by Karp and Miller the entire input activity of a node is always finite.

The following is an example of a Karp and Miller operator; it has two input arcs, one output arc and is associated with the input/output function:

$$h: \omega \times \omega^2 \to \omega$$

$$\forall a \in \omega,\ b \in \omega^2\ h(a,b) = \begin{cases} b_0 + b_1 & \text{if } a \neq 0 \\ b_0 \times b_1 & \text{otherwise} \end{cases}$$

This node awaits the arrival of a single daton on its 0th input arc and two datons on its 1st input arc. Upon arrival of the required inputs they are consumed and a single daton is sent on as output. If the daton that arrives along the 0th input arc is non-zero then the value of the daton output is the sum of the two datons that arrived along the node's 1st input arc; if the daton to arrive along the 0th input arc is zero then the daton output is the product of the two datons that arrived along the nodes 1st input arc. (Thus the claim of Adams [2] and others that data dependent decisions are not allowed in this model is not completely correct).

As well as allowing data dependent decisions such as in the last example many other useful nodes can be realised in this model e.g. plus, times ...etc.

Now that we have briefly described the Karp and Miller model let us see how the model relates to the issues set out in the introduction to this chapter.

The first issue described is that of a completely general model of pipeline dataflow. It is not difficult, even from our brief description of this model, to think of nodes that cannot be realised. An example of such a node is the WHENEVER node which requires the arrival of a daton on both its input arcs and passes on the daton to arrive on its 0th input arc if the daton to arrive on its 1st input arc is "true" (i.e.non-zero) and produces no output if the daton to arrive on the 1st input arc is "false". Thus nodes which vary the number of datons they output depending on the value of the datons input are not realisable in the Karp and Miller model. (It is this form of data dependent decision that is referred to by Adams[2] and others).

However the main limitation of the Karp and Miller model is that each computation step requires a fixed number of inputs and produces a fixed number of outputs. This restriction means that many useful nodes cannot be realised in this model and thus the model cannot be considered as a completely general model of pipeline dataflow.

The second issue is that of a precise definition of what it means for a node to compute a history function. We feel that it is not possible to relate this particular model to this particular issue. The reason for this is that this model was developed before Kahn[26] developed the notion of nodes computing history functions. However it is still possible to think of the entire output activity of a node as being a function of the entire input activity and in this respect the nodes in this model are all functional (i.e. they all compute a history function). The reason for this is that each of a node's computation steps is associated with a single-valued input/output function and thus the entire input activity is related to the entire output activity by a history function derivable from the node's input/output function. This means that we can think of the Karp and Miller model as the first model of pure dataflow.

The third issue mentioned earlier was a proof of the Kahn principle. Again we feel that it is not possible to relate this particular model to a proof of the Kahn principle. The reason is again that the Kahn principle had not been formulated when Karp and Miller designed their computation graphs.

B. A model of parallel computation with dataflow sequencing

The next significant development to follow the Karp and Miller model was the dataflow model described by Duane Adams in his Ph.D dissertation[2]. One of the interesting goals of this dissertation was a graphical programming language for dataflow. An important feature of this language was that nets could be named and used as nodes in other nets. This feature of the language is similar in certain respects to the declaration of procedures in a conventional programming language. In textual programming languages such as Pascal and Algol it is meaningful for a procedure declaration to contain a reference to itself. In a similar way Adams allows a net definition to contain references to itself. The user of the graphical language is asked to think of a net that is recursively defined in terms of dynamically contracting and expanding net. In Chapter 5 we will deal with recursively defined nets, however we think of these recursively defined nets as infinite nets in which only a finite part is ever active.

The primitive nodes described by the Adams model are divided into two classes the r-nodes and the s-nodes.

The r-nodes are a simple extension of the Karp and Miller nodes which allows operators to produce no output for selected inputs. To be more precise an r-node is a single valued input/output function which is permitted to return Ø (the empty output) for certain inputs. A node that makes use of this extension is the WHENEVER node described in the previous section. This node is not realisable by any operator in the Karp and Miller model but in the Adams model it is realisable in terms of the following r-node:

The single valued function associated with each computation step of the WHENEVER node is the following:

$$wvr : \omega \times \omega \rightarrow \omega$$

$$\text{s.t. } \forall a,b \in \omega \quad wvr(a,b) = \begin{cases} b & \text{if } a \neq 0 \\ \emptyset & \text{otherwise} \end{cases}$$

where $\emptyset$ denotes the empty output

Although nodes like WHENEVER can be realised by r-nodes it is not difficult to think of more complex nodes that cannot be realised in terms of r-nodes. An example of such a node is the node with one input arc and one output arc that repeatedly awaits the arrival of a daton on its input arc and as soon as it becomes available consumes it and produces as output n-copies of the consumed input, where n is equal to the value of the consumed input. For example if the node receives as input a daton representing the number 4 then four datons each representing the natural number 4 are output. The fact that the node just described varies the number of datons output depending on the value of the daton input means that it cannot be realsied by any r-node. In fact the node just described cannot be realised by any node in the Adams model.

The second class of primitive nodes are what Adams describes as the s-nodes. These nodes are able to compute by ignoring some but never all of their inputs. An example of such a node is the ALTERNATE merge node which has two input arcs and one output arc. This node begins computation by ignoring its 1st input arc and passing on the daton to arrive on its 0th input arc, after which it ignores its 0th input arc and passes on whatever arrives on its 1st input arc. The node continues merging the datons from alternate input arcs ad infinitum, hence the name ALTERNATE merge node.

In the Adams model an s-node has a boolean flag associated with each of its input arcs. Initially the arcs of an s-node are either L (locked) or U (unlocked), if an arc is locked it means that no input is expected or if input does arrive it is ignored. If an arc is unlocked then input is expected on that arc and until input arrives the node is blocked waiting for input. Thus an s-node with n-input arcs is associated with a vector of boolean flags, one for each input arc. This vector is called the nodes input status. For example a two input s-node may have an initial input status of (U,L) which means that the nodes 0th input arc is initially unlocked and that the 1st input arc is initially locked. S-nodes repeatedly await for a daton to arrive on each of their unlocked input arcs; on arrival of the required inputs, the nodes compute, erasing the required inputs, possibly outputing datons on some or all output arcs and possibly changing the current input status. An s-node with n-input arcs and m-output arcs is thus associated with a single-valued input/output function of the form:

$s : (\omega \cup \{ \emptyset \})^n \text{ x } (\{ L,U \})^n \rightarrow (\omega^{Sq(\omega)} \cup \{ \emptyset \})^m \text{x}(\{ L,U \})^n$

where $\emptyset$ denotes either the empty output or the empty input.

The only restriction to this input/output function is that it is not permissible to have an input status in which all input arcs are locked.

The single valued function associated with the ALTERNATE merge node is :

$alt : (\omega \cup \{ \emptyset \})^2 \text{ x } (\{ L,U \})^2 \rightarrow (\omega^{Sq(\omega)} \cup \{ \emptyset \})\text{x}(\{ L,U \})^2$

s.t. $\forall a,b \in \omega$ $alt(a,\emptyset,U,L) = (a,L,U)$

$alt(\emptyset,b,L,U) = (b,U,L)$

where the initial input status is (U,L)

Even with this more general model it is not difficult to find nodes which cannot be realised. One such node is the NEXT node which consumes its first input and passes on any future input. The reason why nodes like NEXT cannot be realised is that their behaviour cannot be described by a unique single valued function.

Let us now look at how the Adams model of dataflow relates to the topics described at the beginning of the chapter.

The first topic is that of a completely general model of pipeline dataflow. As we have already seen there are many nodes that cannot be described by the Adams model. This means that we cannot think of the Adams model as a completetely general model of pipeline dataflow.

The second topic is that of a precise definition of what it means for a node to compute a history function. With respect to this topic the Adams model is identical to the Karp and Miller model in that they were both developed before Kahn[26] came up with the idea of a node computing a history function. However, it is still possible to think of the entire output activity of an s-node or an r-node as being a function of the entire input activity. In other words the nodes in this model are functional. Given any particular node in the model the history function associated with the node is derivable from a node's single valued input/output function. Although we do not provide a proof it is not difficult to see that the Adams model is another pure dataflow model.

The third topic that was mentioned in the introduction to this chapter was that of a proof of the Kahn principle. Again with respect to this topic the Adams model is identical with that of Karp and Miller in that both models where developed before Kahn[26] formulated the Kahn principle. Therefore we see no point in trying to relate this model to the last topic.

C. A simple language for parallel processing

One of the most significant contributions to the development of pipeline dataflow was the 1974 IFIP paper of Gilles Kahn[26]. The reason for its importance is that Kahn gives for the first time a denotational semantics for an important subset of pipeline dataflow. In addition Kahn describes an operational semantics in terms of a textual dataflow language. In our opinion Kahn's textual language can be thought of as a generalisation of the graphical dataflow language of Adams.

The model of computation underlying Kahn's language is based on the directed graph the nodes of which are continuously operating computing stations and the arcs of which are unbounded fifo queues.

The computing stations in the Kahn model are much more general than in previous models. In particular a computing station has a possibly unbounded amount of internal memory with which it is able to remember all previous inputs.

Kahn was also the first to think of dataflow programs as continuously operating. Designers of previous models restricted themselves in that their dataflow models were based on the traditional notion of a correct program terminating.

Kahn's textual language is similar to ALGOL but with the addition of a few extra features. These new features are based on the process declaration, which is used to define a computing station. The process declaration is similar to an ALGOL procedure declaration except that in the heading of the process we declare how it is linked to its outside world. In other words the input and output arcs are given formal names (similar to formal parameters of a procedure). The body of a process is a usual ALGOL program except for the use of two primitive procedures called PUT and GET. The primitive procedure PUT(E,A) places a daton whose value is equal to the expression E onto the output arc named A. The primitive function GET(A) returns as as its result the value of the daton at the head of the fifo queue associated with input arc A. Nothing can ever prevent a computing station from placing output on an output arc but if a GET(A) is invoked and the queue associated with input arc A is empty then the computing station is forced to wait until a daton shows up.

The NEXT node which was not realisable in the previous models is defined by the following Kahn process:

```
process next(integer in x; integer out y);
          begin
               integer temp;
               temp : = GET(x);
               while true do PUT(GET(x),y);
          end;
```

Unlike previous models of pipeline dataflow the Kahn model allows nodes to have memory. A simple example of such a node is one that outputs the running total of the datons input. In Kahn's textual language this would be written as follows:

```
    Process runtoatal(integer in x; integer out y);
            begin
               integer sum;
               sum : = 0;
               repeat
                 sum : = sum+GET(x);
                 PUT(sum,y);
               end;
            end;
```

Even with these few examples we can see that Kahn's language is extremely powerful. In fact in a later paper with McQueen[27] they look more closely at an implementation of this language based on co-routines. In the 1974 IFIP's paper Kahn points out that all processes definable in his language are functional (something he does not prove).

Some of the restrictions imposed by Kahn on his model are that computing stations (i.e. the nodes) have to follow a sequential program and that at any given time a computing station is either computing or waiting for input on one of its input arcs but not both.

Now that we have briefly described Kahn's textual language let us look at the operational model underlying the language and see how this model relates to to the topics listed in the introduction to this chapter.

The first topic to consider is whether or not the operational model underlying Kahn's language is a completely general model of pipeline dataflow. Since the nodes in this model must be sequential then the underlying model cannot be completely general. In fact it is not difficult to think of nodes that are functional but cannot be realised in Kahn's language. A simple example of a functional node that cannot be realised in Kahn's textual language is the DOUBLEID node. This node has two input arcs and two output arc and passes onto its 0th output arc whenever appears on its 0th input arc and passes onto its 1st output arc whatever appears on its 1st input arc. If we try to code this up in Kahn's language we get the following process:

```
process doubleid(integer in x,y; integer out p,q);
        begin
          repeat
             PUT(GET(x),p);
             PUT(GET(y),q);
          end;
        end;
```

If the process is always given an infinite number of inputs on both input arcs then the node produces the correct output. However if one of the input arcs dries up then the whole node is blocked waiting for inputs. Thus the process is unable to compute the general double identity function (i.e. identity: (Ka^2->$Ka)^2$).

We must therefore conclude that the Kahn model is not suitable as a completely general model of pure dataflow . This is not surprising since as we explained in chapter 0 the topics of generality and functionality are difficult to capture directly. Thus the operational model underlying Kahn's textual language is not a completely general model of pipeline dataflow nor is it a completely general model of pure dataflow. However the Kahn model can be thought of as a compleletly general model of pure dataflow in which the nodes are are sequential and deterministic.

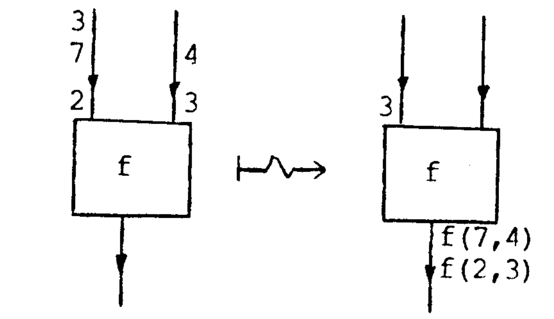
The second topic is that of a precise definition of what it means for a computing station to compute a history function. Unfortunately Kahn never defined precisely the concepts of "nodes", "nets", etc. and as a result never gave a formal definition of what it means for a node to compute a history function. In his IFIP's paper he used the fact that all processes definable in his language compute history functions but he never gave a proof of this. In particular he does not state whether or not processes may have formal parameters that are called by name. We assume that these are not allowed as otherwise processes could have side effects and hence not be functional.

In chapter 0 we briefly described what is meant by the Kahn principle, we now examine to what extent Kahn proved his principle. In fact it turns out to the surprise of many computer scientists that Kahn never published a formal proof of the Kahn principle.

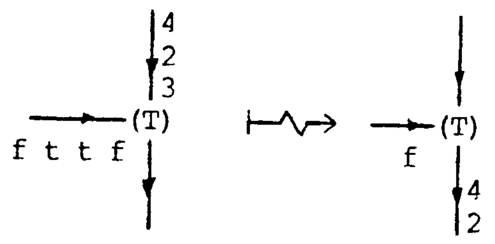
D. An asynchronous interpreter for the dataflow language DDF

The first published paper to attempt a proof of the Kahn principle was that of Arvind and Gostelow[3]. In their paper they describe an interpreter called the queued interpreter (QI) which is thought of as a machine for the excecution of programs for an early dataflow language of Jack Dennis[19]. This language is referred to by Arvind and Gostelow as DDF (Dennis DataFlow).

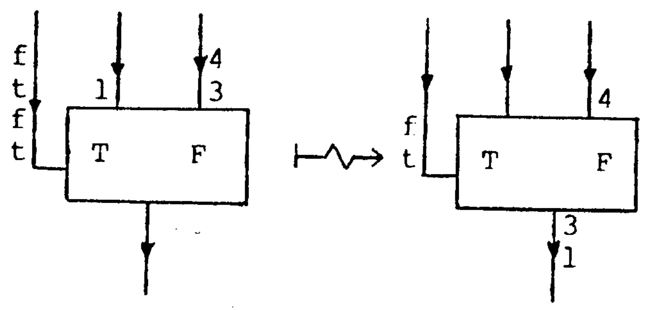
A program in DDF is, roughly speaking, a directed graph whose arcs are unbounded fifo queues and whose nodes are choosen from the following 5 primitives:



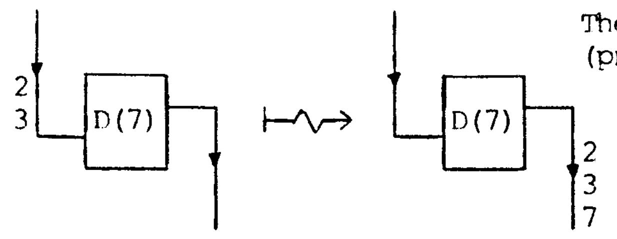
The results of all firings of a function or predicate operator under QI.



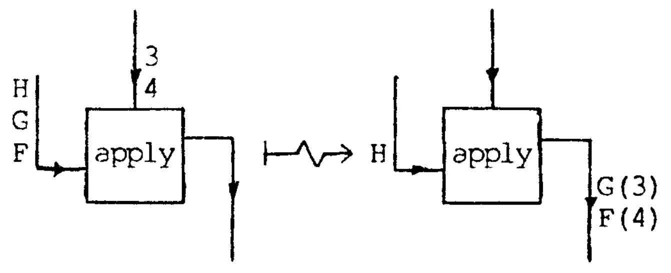
The results of a gate if-true operator, where ff, tt denote false and true respectively.



The merge operator



The D-Box operator
(produces an initial daton)



The apply operator

According to Arvind and Gostelow dataflow is based upon two principles :

(a) An operator fires (produces an output) whenever the inputs required by that operator are present.

(b) All operators are functional and produce no side effects.

This view of dataflow is exactly the same as that of Karp and Miller[28] (i.e. each set of inputs is required to produce a set of outputs) and so many of the remarks made about the Karp and Miller model (see section A) also apply to this model.

Let us now examine to what extent this model relates to the issues set forth in the introduction to this chapter.

The queued interpreter just described is based on five primitive operators and as such we cannot think of this model as a completely general model of pipeline dataflow. (i.e. there are infinitely many pure dataflow nodes).

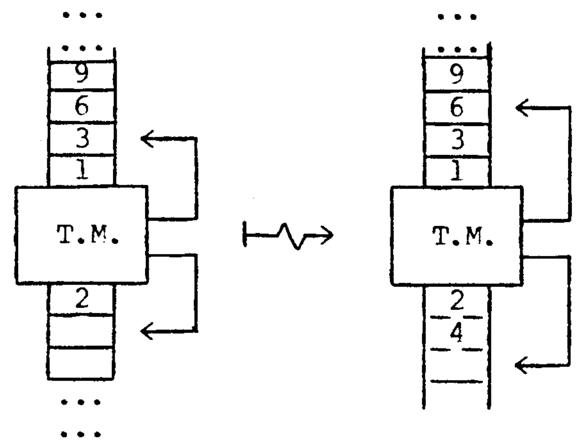
The 5 primitive nodes used by Arvind and Gostelow are all obviously functional and so the more fundamental problem of defining what it means in general for a node to compute a history function is completely avoided.

The proof of the Kahn principle given by Arvind and Gostelow is not presented in a very formal way. For example there is no precise definition of a dataflow net computation. Presumably the assumption is that the model is so simple that no foraml definition of computation is necessary. Moreover even if it was presented formally their proof is only for nets built using the 5 primitives described earlier. Thus the proof of the Kahn principle is for a very limited context.

E. Computing with infinite objects

A second attempted proof of the Kahn principle is outlined in the Ph.D. thesis of Edwin Wiedmer[49]. In this thesis Wiedmer is interested mainly in exact computations over the real numbers. However in the second part of his thesis he describes some of the theoretical issues underlying computation over finite or infinite objects. Using infinite sequences of natural numbers to represent the reals he describes machines which compute operations over these infinite objects (i.e.the reals). Each machine reads in natural numbers one by one from its input arcs and produces one by one natural numbers on its output arc. The entire history of the machine's input activity denotes the real numbers given to the machine as input and the entire history of the machine's output activity, the real numbers produced as output. More complicated operations over the reals often require a network of machines. Wiedmer thinks of these networks as dataflow nets the arcs of which are unidirectional communication lines and nodes of which are continuously operating computing stations.

Wiedmer follows the advise of Gilles Kahn[26] and defines nodes as Turing machines and arcs as one way infinite tapes upon which a Turing machine reads and writes. A node with n input arcs and one output arc is defined to be a Turing machine with n input tapes and 1 output tapes. The Turing machine has a separate read head for each input tape and a separate write head for the output tape. Once an item has been read from a square on an input tape the read head is forced to move to the next square away from the Turing machine. (Initially all heads are over the square nearest the Turing machine). In a similar way once a write has been performed the write head moves to the next square away from the Turing machine. The diagram illustrates a computation step for a one/input one/output Turing machine:



In addition a node (i.e.Turing machine) is able to use an auxiliary tape (not shown in the diagram) onto which it can record all the inputs it has received and all the results of intermediate computations.

Now that we have given a brief description of the model let us relate it to the issues listed in the introduction to this chapter. The fact that Wiedmer uses Turing machines as nodes means that his nodes are, at least in terms of computability, the most general form for a node. However we feel that Turing machines are not a very natural model of dataflow. Nodes are no longer simple black boxes rather they are Turing machines together with those portions of the tape which have been read or written on. Arcs are no longer pipes along which datons flow but rather they are infinite one-way tapes along which a read or write head travels. Thus the model of Wiedmer is not a very natural model of pipeline dataflow.

One of the most important aspects of Wiedmer's work is his definition of what it means for a node (Turing machine) to compute a history function. He is the first to realise that Kahn's definition of node functionality, based on processes that communicate using only PUT and GET primitives, is not compeletely general. In fact Wiedmer refers to functions computable by Kahn's process as 'rigidly computable functions'. The function f(a,b) = max{ a,b } where a,b are finite sequences of natural numbers; is given as an example of a function not computable using the simple language for parallel programming described by Kahn.

Wiedmer refers to functions computable by his machines as 'approximation computable functions'. What Wiedmer means by this is that Turing machines are allowed to compute given only an approximation to the input i.e. even when inputs are absent. This means that his Turing machines are able to realise nodes such as the DOUBLEID node, in fact any non-sequential node may be realised.

Although Wiedmer's definition of node functionality is more general than that of Kahn we still feel that it is not an adequate definition. The reason for this is that Wiedmer's Turing machines are deterministic and we feel that a completely general definition of node functionality should include nodes that are not only non-sequential but also non-deterministic. The reason we say this is that a completely general defintion of node functionality should be able to describe any node that computes a history function no matter how bizarre its behaviour is.

The remainder of Wiedmer's work is extremely sketchy; for example there is no definition of a network of machines nor is there a definition of a net computation (this is not surprising since these issues were not of major importance in Wiedmer's work). Towards the end of part two of Wiedmer's thesis we find a schetch proof of the Kahn principle which is based on nodes being Turing machines, the proof is not a very convincing one.

F. Semantique des processus communicant

Recently Andre Arnold[4] published a paper in which he attempts to prove the Kahn principle. The author claims that the main goal of his paper is to give a satisfactory operational semantics corresponding exactly to Gilles Kahn's denotational semantics for pure dataflow (i.e. the Kahn principle). However there seems to be a substantial error in this work in that it is not general enough to cover all Kahn's networks.

Operationally a node in the Arnold model has the following properties:

i) they have a finite number of input and output arcs.

ii) each input arc is an unbounded fifo queue.

iii) each node has an unbounded amount of internal memory. (it is thus able to remember all previous inputs).

iv) to function each process must acquire datons on certain input arcs, if these are not present the process is blocked waiting for input.

These properties are all included in the following formal definition.

F1 Definition: A process with n-input arcs and 1 output arc is a tuple $<\Sigma_0,\ldots,\Sigma_n,Q,q,\delta,R>$

where

$\Sigma_0,\ldots,\Sigma_n$ are possibly infinite alphabets such that Σ_0 is the alphabet of the output arc and Σ_i $1 \leq i \leq n$ is the alphabet of the ith input arc.

Q is a possibly infinite set of states

$q_0 \in Q$ the initial state

$\delta : Q \to \mathcal{P}([n])$ where $[n] = \{ 1, \ldots, n \}$

R is a set of rules

The mapping δ associates with each possible state the set of numbers that denote the input arcs on which the process must receive inputs. Thus mapping δ has a similar function to the input status flags of Adams.

The set of rules describes for each state: the datons necessary to enable the node to compute and also the activity that should be performed when computation is enabled. This activity may involve changing state and outputing one or more datons on the output arc. The following is a formal definition of a rule:

F2 Definition: each rule is of the form

$<q; u_1, \ldots, u_n> \to <u; q'>$

with $q, q' \in Q$

and $u \in \Sigma_0 \hat{} \, \Sigma_0^*$

(Σ_0^* is the set of finite sequences over Σ_0)

$u_i \in \Sigma_i$ if $i \in \delta(q)$

$u_i = \Lambda$ (empty sequence) if $i \notin \delta(q)$

The formal operational semantics given by Arnold are extremely precise. In particular he gives precise definitions to concepts such as nodes, nets, computation sequences, .. etc. As can be seen from the last definition Arnold allows nodes to be non-deterministic in that more than one rule may have the same left hand side. Although non-deterministic nodes are allowed in this model they are never used to describe nodes that compute history functions. Thus the nodes which compute history functions are sequential and deterministic.

The substantial error referred to earlier is to do with Arnold's definition of a node. To be more precise Arnold requires his nodes to produce at least as much output as they receive input. This means that essential filter nodes like NEXT and WHENEVER that produce less output than they receive input cannot be realised as Arnoldian processes. These simple filters can certainly be realised as Kahnian processes and thus Arnold's claim that his operational model is equivalent to Kahn's is certainly false. The reason that Arnold requires that his nodes compute at least as much output as they receive input is that he requires his nodes to compute history functions with a special property. To understand this property we must describe a metric space over infinite sequences of natural numbers in which the distance between two objects is defined to be 1/N where N is the amount of agreement over the initial segments of the two objects. The fact that Arnold requires his nodes to produce at least as much output than they receive input corresponds to a node computing a contraction mapping on the metric space described above. It is a well known result that under certain conditions contraction mappings give unique fixed points. Arnold is able to use these facts in some of his proofs.

Let us now relate Arnold's model to the topics set out in the introduction to this chapter.

Already we have seen that there are many useful functional nodes that cannot be realised as Arnoldian processes. Thus Arnold's model cannot be thought of as a completely general model of pipeline dataflow.

As a result of the restriction to the definition of node we find that the definition of what it means for a node to compute a history function is correspondingly restricted. The following is the definition of what it means for a node to compute a function overs histories. This definition is important because it is the first to formally define what it means for a node to be functional. Arnold uses two versions of this definition. The first is for processes that are sequential and deterministic, in this case the function $\hat{P}$ is a history function. The second is for processes that are non-determinate and sequential, in this case the function $\hat{P}$ is a set valued function and this is the version of the definition shown below.

F3 Definition: The function

$\hat{P}:\Sigma_1^{\omega} \times \ldots \times \Sigma_n^{\omega} \to \mathcal{P}(\Sigma_0^{\omega})$

associated with P (P a process) is such that $\hat{P}(V_1,\ldots,V_n)$ is the set of results of the ω-derivations starting from the initial configuration $\langle \Lambda, q_0, V_1, \ldots, v_n \rangle$.

Note that if the process is deterministic then there is only one possible ω-derivation starting from the initial state. In our opinion this definition is restricted in that it assumes that a node always begins computation with infinite amounts of data on all its input arcs. We feel that this assumption is not realistic because in any real system the input arcs may be empty to begin with and even at intermediate stages in the computation. We are thus led to conclude that Arnold's definition of node functionality is not completely general since he does not allow non-deterministic nodes to compute functions.

The third topic listed in the introduction to this chapter is concerned with proving the Kahn principle. Of all the published proofs of the Kahn principle the most precise is that of Arnold. However since Arnold's model is unable to describe nodes that produce less output than they receive input the proof of the Kahn principle is only a simple case of the more general Kahn Principle.

Of all the models of dataflow we have surveyed the model of Arnold is the closest to ours, although we developed our models separately they are surprisingly similar in many ways. For this reason we conclude this section by describing a few of the important definitions and ideas contained in Arnold's work. We begin by looking at an example of a non-deterministic process: Example Let P be a process with n-input arcs and 1 output arc defined by

$<\Sigma_0, \Sigma_1, Q, q_0, \delta, R>$

where $\Sigma_0 = \Sigma_1 = \{ a,b \}$

$Q = \{ q_0, q_1, q_2 \}$

$\delta (q) = \{ 1 \} \ \forall \ q \in Q$

and the rules are:

$<q_0;a> \to <a;q_1>$

$<q_0;a> \to <b;q_2>$

$<q_0;b> \to <b;q_2>$

$<q_1;a> \to <a;q_1>$

$<q_2;a> \to <a;q_1>$

$<q_2;a> \to <b;q_2>$

$<q_2;b> \to <b;q_2>$

This is an example of a non-deterministic process that produces different results depending on what infinite input sequence is supplied. For example if a ω is supplied as input then either a ω or b ω will be output. If b ω is supplied as input then b ω is produced as output. Of all the work we have reviewed Arnold's is the only one to give a formal definition for a dataflow net.

(F4) Definition: Let $P_1,\ldots,P_k$ be k processes P_i having n_i input arcs and m_i output arcs.

Let $p \in \omega$

Let $n = n_1+n_2+\ldots+n_k$

Let $m = m_1+m_2+\ldots+m_k$

A dataflow net is given by the k processes the natural number p and two injections:

$\eta : [n] \rightarrow [p]$

$\sigma : [m] \rightarrow [p]$

The intuitive meaning of this definition is the following:

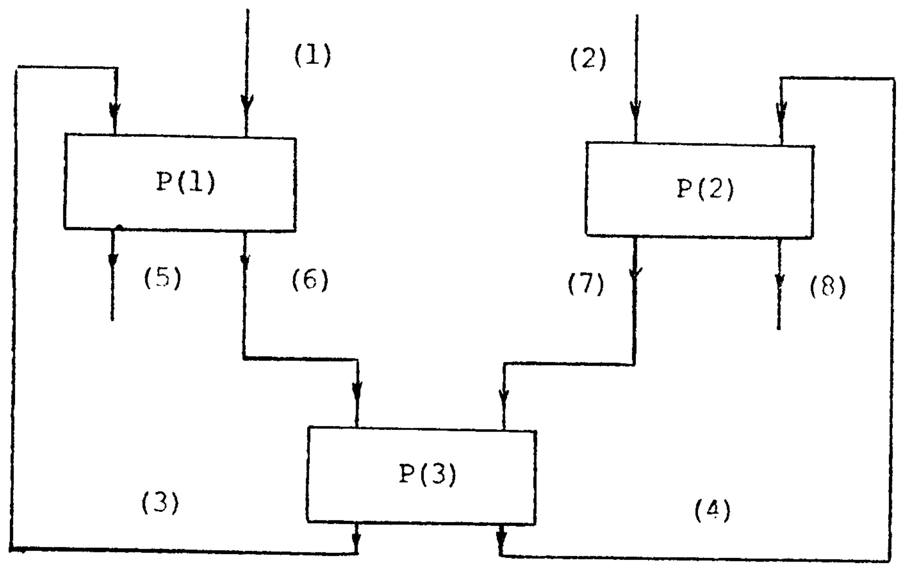
The j^{th} input port of process P_i ($1 \leq j \leq n_i$) is given the number $n_0+n_1+\ldots+n_{i-1}+j$ where $n_0 = 0$. In the same way the j'^{th} output port of process P_i is given the number $m_0+m_1+\ldots+m_{i-1}+j$. The arcs of the net are numbered from 1 to p. The natural number $\eta(1)$ is thus the number of the arc connected to input port number 1 and the natural number $\sigma(1)$ is the number of the arc connected to output port number 1. We therefore suppose that $\eta([n]) \cup \sigma([m]) = [p]$

Figure F5 An example of a data flow net.

The net consists of 3 processes P_1, P_2 and P_3 each having 2 input ports and two output ports .The two injections η and σ from $[6] \rightarrow [8]$ are given by the following table:

	η	σ
1	3	5
2	1	6
3	2	7
4	4	8
5	6	3
6	7	4

The following is a graphical representation of the net.



<u>A modular operational semantics</u> ?

Gilles Kahn gave a lot of importance to the fact that a network of processes could be considered as a single process (even though it was not true in his model). Motivated by this consideration Arnold proves that his operational model has this encapsulation property. However we find Arnold's result rather misleading because it is not true in general. The reason for this is that Arnold assumes that his nodes always receive infinite amounts of input on all input arcs and in reality this may not be the case. For example the DOUBLEID node described earlier can be written as a single process in Arnold's notation and it can replace two seperate single identity nodes in any context provided an infinite amount of input is given to both input arcs of the DOUBLEID node. However if one of the inputs dries up the node is blocked waiting for input and is thus unable to compute the DOUBLEID function. Thus Arnold's encapsulation property is for the limited context in which sequential and deterministic nodes compute in an environment in which they are never starved of input. Thus the Encapsulation property is trivially true for Arnoldian nets. In chapter 2 we prove that our model of dataflow has the encapsulation property in a completely general sense.

G. A language capable of expressing all pipeline Dataflow computations

In this, the last section of the current chapter, we describe a programming language proposed by Robert Keller[29]. Keller extends Kahn's simple language for parallel processing by the addition of two new primitive nodes called poll and choice. Keller is mainly interested in dataflow nets in which nodes may behave non-deterministically, and as a consequence, the new primitives both introduce non-determinism into what is otherwise a sequential and deterministic language. Keller states that non-determinate behaviour can be introduced into a language in two ways. One way is through time dependency, this form of non-determinate behaviour is caputred by the poll primitive. The other form of non-determinate behaviour is randomness within a computing station and this form is captured by the choice primitive. Keller introduces these primitives into Kahn's language in the following way:

The poll primitive checks to see if the fifo queue associted with a particular arc is empty or not. We can think of this in terms of a boolean procedure in ALGOL which returns true if the queue it is looking at is non-empty and false otherwise. This can be expressed with a primitive of the following form:

NONEMPTY(<arc name>) The inclusion of this primitive into Kahn's language enables processes to detect an empty input arc and thus avoid being blocked waiting for input. This enables Keller's processes to realise nodes like the DOUBLEID node mentioned earlier.

The choice primitive randomly chooses between different activities. We can think of this in terms of a boolean procedure in ALGOL which returns a random truth value and when used in conjunction with a conditional primitive such as if-then-else allows a process to non-deterministically choose between different computations. This primitive may be expressed in the following form:

```
        CHOICE()
```

Although we do not provide a proof we claim that the addition of these two primitives to Kahn's language means that the extended language is able to describe any pipeline dataflow computation. For example here is the DOUBLEID node:

```
    process doubleid(integer in x,y; integer out p,q);
           begin
             repeat
               if NONEMPTY(x) then PUT(GET(x),p);
               if nonempty(y) then PUT(GET(x),q);
             end;
           end; Another example is the unfair MERGE node:
    process merge(integer in x,y; integer out z);
          begin
             repeat
             if NONEMPTY(x) and NONEMPTY(y) then
                if CHOICE() then PUT(GET(X),z)
                            else PUT(GET(y),z)
                fi;
             fi;
             if NONEMPTY(x) then PUT(GET(x),z) fi;
             if NONEMPTY(y) then PUT(GET(y),z) fi;
             end;
          end;
```

Keller, like many others, assumes that Kahn gave a proof of the Kahn principle in [26] and thus does not address the issue. As a consequence Keller is not interested in non-deterministic and non-sequential nodes that are functional. Rather he is intersted in nodes with non-determinate input/output behaviour.

In the following chapter we develop our own formal operational semantics for pipeline dataflow. The reason we do this rather than adopt Keller's language is that we want to be able to reason formally about dataflow nets and dataflow computations and this is very difficult to do from a language like Keller's. For example in Chapter 5 we prove that nodes with a partcularly simple property, namely the 1 step Church-Rosser like property, are functional. A corresponding proof for processes defined in Keller's language would be extremely difficult.

Chapter 2

A Formal Operational Semantics for Pipeline Dataflow

The main result of this chapter is a precise formulation of a completely general operational semantics for pipeline dataflow. In section A we present a formal definition of a pipeline dataflow node in terms of a non-deterministic automaton. In section B we examine networks of these nodes. The two main result of section B are: (1) a formal definition of closed pipeline dataflow nets and (2) a formal definition of closed net computations. In Section C we show how the formalism of section B can be re-used to describe open nets and open net computation.

To support our claim that our operational semantics is completely general in section E, we prove that our model has the encapsulation property in a completely general sense.

A. Nodes and non-deterministic automata

The nodes in our model of pipeline dataflow will be continuously operating automomous computing stations connected to one another via pipes which are unbounded fifo queues and along which they endlessly exchange information.

To save notational complexity and without loss of generality we assume that the discrete units of data that travel along the pipes (i.e. the datons) represent only natural numbers.

A computing station in our model consumes datons one by one from its input arcs and outputs datons one by one on the computing stations output arcs. In terms of input/output behaviour our model is not sequential in that each node computation may simultaneously produce and consume datons on all its input and output arcs. Although we do not provide a proof we claim that all input/output behaviours can be expressed by our model.

A simple node in our model usually consumes datons at the same rate at which it produces them. However, we allow more general nodes that may produce output at a different to that at which they consumes inputs.

The justification for reasoning about computing stations as continuously operating black boxes is that many applications such as operating systems and database systems are best thought of in terms of continuously operating autonomous processes. A practical example of this is a network of UNIX processes connected to each other via UNIX pipes.

In our formal operational semantics each computing station (node) is associated with a set of internal states and at any given moment a node is in one of these states. When a node is first "activated" it moves automatically into a known initial state. Thereafter it may move to other internal states depending upon what the node is to compute. We can think, informally, of the internal state of a node as having two distinct roles.

One role is as a "marker", marking the current step in the algorithm that specifies a node's behaviour. The initial state is a marker to the first step in such an algorithm. For example consider a node with two internal states, one input arc and one output arc. In its initial state the node consumes its first input, produces no output and moves into a second state. In its second state the node consumes its next input and produces as output a copy of the consumed input. If we require this last step to be repeated ad infinitum we arrange for the node to remain permanently in its second state. As you have probably realised the node just described is our old friend the NEXT node.

A second role or use of internal states is as memory. We feel that is is not unreasonable to think of a node requiring access to all of its previous inputs in order to produce its next output. As nodes are continuously operating this may require a possibly unbounded amount of internal memory. (Some authors such as Arvind and Gostelow[3] restrict themselves to a subset of pipeline dataflow in which nodes have only one state (i.e. no memory). As a result of these restrictions these models are certainly not general models of pipeline dataflow. In addition they lack the encapsulation property- subnets have memory, in the form of daton queues, but nodes have none). An example of a node that uses internal state as memory is the node that produces on its one output arc the running total of the datons it has consumed through its one input arc. In the case of this node we think of the initial state as initialising the running total to zero. The node repeatedly awaits the arrival of a daton on its input arc and as soon as one becomes available it is consumed, the value of the consumed daton (a natural number) is added to the internal state and a daton representing the new running total is output. Since the sum of two natural numbers is always a natural number, a countably infinite number of internal states enables the node to record succesive running totals and thus the node works correctly for any input history.

Although we can informally think of internal states as having two distinct functions this does not mean that nodes need separate internal states for each of these functions. On the contrary, our nodes may 'code up' both of these functions within a single internal state.

We think of our computing stations as black-boxes connected to their outside world through input and output pipes through which they communicate with one another. Now it is certainly possible for a node that consumes datons at a very slow rate to receive inputs from a node that produces datons very quickly. If this is the case then the pipe connecting the two nodes should be able to store the surplus datons in the order in which they arrived. This is the justification for the earlier decision to take pipes as unbounded fifo queues.

In our model we require our nodes to consume datons one by one from their associated input arcs. To formalise this reasoning we associate a one place input buffer with each of a nodes input arcs. This one place buffer is empty if the fifo queue assoicated with the buffers input arc is empty, otherwise it holds the daton at the head of the input arcs associated fifo queue. Our nodes are able to consume a daton from an input arcs by issuing a command to erase the corresponding input buffer.

The contents of each one place input buffer together with the internal state gives a snapshot description a node which we call the cause of computation. With every possible cause our nodes (are required) to associate some effect. An effect may be to erase some or all of the nodes input buffers; it may be to change the internal state, or it may be to output a daton on some or all of the output arcs, or a combination of these 3 activities. Therefore unlike the model of Arnold[4], which becomes blocked if an unexpected input arrives, our model is able to cope with all possible input situations.

To illustrate the idea of causes and effects we turn to our old friend the NEXT node. This node has two internal states q_0 (the initial state) and q_1. A snapshot of this node computing may reveal that the node is in its initial state q_0 with a 5 in its input buffer. We denote this cause of computation by the ordered pair $<5,q_0>$. A later snapshot may reveal that the node is in state q_1 with an empty input buffer. We denote this cause by the ordered pair $<nil,q_1>$, the nil meaning that the input buffer is empty. As we are assuming that datons are all of type natural number the following is the set of all possible causes for the NEXT node:

$\{ <nil,q_0>,<nil,q_1>,<0,q_0>,<0,q_1>,<1,q_0>,<1,q_1>.... \}$

If our nodes are to compute for any input then it is essential that for each possible cause we associate at least a single effect. For the NEXT node we could associate causes with effects in the following way. For all those causes in which the state component is q_1 and the input buffer is non-empty we would associate an effect which is to consume the contents of the input buffer, not to change internal state and to output the consumed input. For example the cause $<2,q_1>$ is associated with the effect <tt, nil, 2>. The tt meaning erase the input buffer, the nil meaning do not change state and the 2 meaning output a daton whose value is 2. Similarly for all those causes in which the state is q_0 and the input buffer is non-empty we would associate an effect which is to erase the contents of the input buffer, to change internal state to q_1 and not to output anything. For example the cause $<2,q_0>$ is associated with the effect $<tt, q_1, nil>$. The tt meaning erase the contents of the input buffer, the q_1 meaning move to the new state q_1 and the nil meaning do not produce any output. The two remaining causes <nil,q_0> and <nil,q_1> are both associated with the busy wait <nil, nil, nil>. The first nil meaning the do not erase the input buffer, the second meaning do not change internal state and the third meaning do not produce any output.

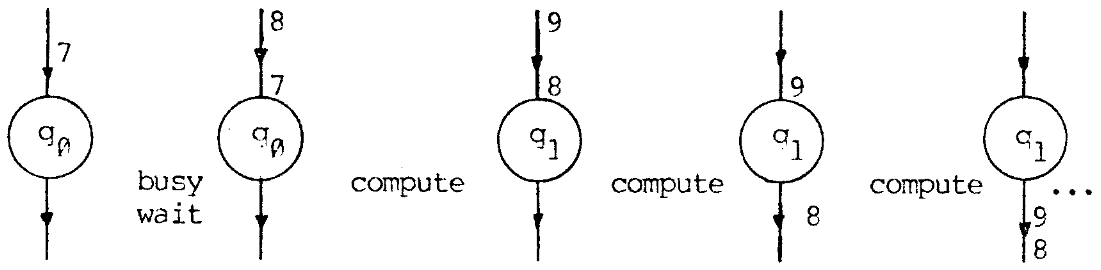
Provided the node just described computes indefinitely it will compute the following history function:

next: Ka -> Ka

s.t. $\forall \alpha \in$ Ka next(α) = < $\alpha_1, \alpha_2, \alpha_3, \ldots$ >

note: α = < $\alpha_0, \alpha_1, \alpha_2, \ldots$ >

The following diagram illustrates a possible computation sequence for the node just described:



As we saw in the previous example if one or more of the input buffers associated with a cause is empty then it is still possible to associate an effect with that cause. In some cases the effect may be to do nothing, we call this busy waiting.

On the other hand the effect may be to cause some activity, this is called computing on empty buffers. It is possible for sequential nodes (e.g. Arnold's[4]) to compute when some of their buffers are empty, but only if they completely ignore the contents of these buffers. Using Kahn's GET primitive, for example, it is possible to wait for the appearance of a daton down the first arc and output it when it arrives even if the second buffer is empty. But when a GET is invoked, the node must do just that and has no way of knowing whether or not anything has arrived in the other buffer.

The more general nodes which we allow, however, are capable of performing other activities (such as output) while waiting for input on certain arcs - in other words, they are essentially able to do more than one thing at the same time. It is precisely these more general nodes that have been ignored in previous models. A very simple example of such a node is the DOUBLEID node. This node has two inputs and two outputs and echos the first input on the first output, and the second input on the second output. Such a node cannot be sequential because it cannot allow both outputs to 'run dry' when only one of the inputs does so. This ability to compute while waiting is essential if our model is to be in any sense general. In fact any model that is unable to compute in this way will be deprived of the encapsulation property.

The following formal definition of a node is based on the informal ideas presented above. A node is specified by: the number of input and output arcs; the initial internal state; the set of all possible internal states, and the collection of all possible cause/effect pairs.

(A1) Definition A node is a sequence $<Q,q,n,m,T>$

where

Q is a countable set with $nil \notin Q$

(the set of all possible internal states)

$q \in Q$

(the initial internal state)

$n,m \in \omega$

(the number of input/output ports resp.)

$T \subseteq (B^n \times Q) \times (E^n \times \hat{Q} \times B^m)$

(the transition relation)

such that

$BW \subseteq T$

where

$BW = \{ <c,nil^{n+m+1}> \mid c \in (B^n \times Q) \}$

$B = \omega \cup \{ nil \}$

$\hat{Q} = Q \cup \{ nil \}$

$E = \{ tt, nil \}$

To facilitate later work we introduce the following auxiliary functions:

Let N be a node $<Q,q,n,m,T>$

(i) States(N) is Q

(ii) Initialstate(N) is q

(iii) Inportarity(N) is n

(iv) Outportarity(N) is m

(v) Transitions(N) is T

Let $t \in$ Transitions(N)

(i) Buffers(t) is $t_0 | $Inportarity(N)

(ii) Newstate(t) is t_1(Inportarity(N))

(iii) Prod(t) is $<$ t_1(Inportarity(N)+1+j) | j $\in$ Outportarity(N)$>$

(iv) Erase(t) is $t_1 |$ Inportarity(N)

Node transitions

Let us examine in more detail the concept of a transition relation. Given a node n (= <Q,q,n,m,T>), the transition relation T is a subset of $(B^n \times Q) \times (E^n \times Q \times B^m)$ i.e. a set of cause effect pairs.

A typical cause is of the form $\langle b_0,\ldots b_{n-1},q\rangle$ where b_i denotes the status of the i^{th} input buffer. If b_i is nil then the i^{th} input buffer is empty, if b_i is some natural number say 7 then the i^{th} input buffer contains the natural number 7. The q component of our typical cause is a node state.

A typical effect is of the form $\langle e_0,\ldots,e_{n-1},z,b_0,\ldots,b_{m-1}\rangle$ where the e_i denotes the activity associated with the i^{th} input buffer. If e_i is nil then the contents of the i^{th} input buffer is not to be erased. If the e_i is tt then the contents of the i^{th} input buffer is to be erased. The z component denotes the activity associated with the node's state. If z is nil then this means that the node's state should remain unchanged. If z is q_{new} then this means the node's state should be changed to the state q_{new}. The b_i component denotes the acitivity associated with the node's output arcs. If b_i is nil then nothing is to be sent along the i^{th} output arc. If b_i is some natural number say 5 then that means that the natural number say 5 is to be sent along the i^{th} output arc.

At this point let us digress to comment upon the somewhat unconventional 'next state' relation used in definition 2.1. Those familiar with traditional automata theory may feel a little uneasy in the use of nil to specify no change in state. In standard automata theory the next state relation is a relation over States $\times$ States and not over States $\times$ (States $\cup$ { nil }). At this point in the thesis we cannot justify this unconventional next state relation but the reasons for its inclusion will become apparent when in Chapter 5 we define a one step Church-Rosser like property for node's.

To familiarise the reader with our definition of a node we present several examples of node's. The first few examples are elementary but the later ones are complex and are meant to show the generality of our definition.

Example 1

The following is the formal definition of a node having two input arcs, one output arc and one internal state. The node in this example repeatedly awaits the arrival of datons on both its input arcs and as soon as both inputs arrive they are consumed and a daton representing their sum is output.

```
< { q }, q, 2, 1, T >
   where T = {  <<nil,nil,q>,<nil,nil,nil,nil>>,


               <<  0,nil,q>,<nil,nil,nil,nil>>,
               <<  1,nil,q>,<nil,nil,nil,nil>>,
               <<  2,nil,q>,<nil,nil,nil,nil>>,
                   .            .
                   .            .
                   .            .
               <<nil,  0,q>,<nil,nil,nil,nil>>,
               <<nil,  1,q>,<nil,nil,nil,nil>>,
               <<nil,  2,q>,<nil,nil,nil,nil>>,
                   .            .
                   .            .
                   .            .
               <<  0,  0,q>,< tt, tt,nil,  0>>,
               <<  0,  1,q>,< tt, tt,nil,  1>>,
               <<  0,  2,q>,< tt, tt,nil,  2>>,
                   .            .
                   .            .
                   .            .
               <<  1,  0,q>,< tt, tt,nil,  1>>,
               <<  1,  1,q>,< tt, tt,nil,  2>>,
               <<  1,  2,q>,< tt, tt,nil,  3>>,
                   .            .
                   .            .
                   .            .
               <<100,  0,q>,< tt, tt,nil,100>>,
               <<100,  1,q>,< tt, tt,nil,101>>,
                  .             .
                  .             .
                  .             .
                 }
```

<u>Example 2</u>

This example is the formal defintion of our old friend the NEXT node. This node has two states, one input arc and one output arc. Unlike the last example the node does not process datons it simply manipulates them. To be more precise the node dicards its first input but thereafter passes on any future inputs.

$< \{ q_0, q_1 \}, q_0, 1, 1, T >$

where T = {

$$
\begin{array}{l}
<<nil, q_0>, <nil, nil, nil>>, \\
<< 0, q_0>, < tt, q_1, nil>>, \\
<< 1, q_0>, < tt, q_1, nil>>, \\
\quad \vdots \qquad\qquad \vdots \\
<<nil, q_1>, <nil, nil, nil>>, \\
<< 0, q_1>, < tt, nil, 0>>, \\
<< 1, q_1>, < tt, nil, 1>>, \\
<< 2, q_1>, < tt, nil, 2>>, \\
\quad \vdots \qquad\qquad \vdots
\end{array}
$$

}

Example 3

The following is the formal definition of a node that has a countably infinite number of states, one input arc and one output arc. This is a naive example of a functional node that computes on empty buffers. In fact the node computes the following history function:

first: Ka -> Ka

s.t $\forall \alpha \in$ Ka first(α) = $< \alpha_0, \alpha_0, \alpha_0, \ldots >$

The formal definition of this node is:

$< \{ q_i \mid i \in \omega \} \cup \{ q \}, q, 1, 1, T>$

where $T = \{$ <<nil,q>,<nil,nil,nil>>,

```
            << 0,q>,< tt,q_0,  0>>,
            << 1,q>,< tt,q_1,  1>>,
                .        .
                .        .
                .        .
            <<nil,q_0>,<nil,nil,  0>>,
            << 0,q_0>,< tt,nil,  0>>,
            << 1,q_0>,< tt,nil,  0>>,
                .        .
                .        .
                .        .
            <<nil,q_1>,<nil,nil,  1>>,
            << 0,q_1>,< tt,nil,  1>>,
            << 1,q_1>,< tt,nil,  1>>,
                .        .
                .        .
                .        .
            <<nil,q_99>,<nil,nil, 99>>,
            << 0,q_99>,< tt,nil, 99>>,
            << 1,q_99>,< tt,nil, 99>>,
                .        .
                .        .
                .        .
                   }
```

Up to this point in the thesis we have defined the set of transitions associated with a node by writing down every individual member of that set. We chose such a verbose method of defining the set of transitions so as to avoid any misunderstanding of what we mean by a set of transitions. It is now time to introduce a more concise notation. To do this we have defined a notation similar to that of Rodriguez[43] .Instead of writing transitions as a countable set of ordered pairs such as

$\{ \quad \langle\langle nil, q_0\rangle, \langle nil, nil, nil\rangle\rangle,$

$\langle\langle\ 0, q_0\rangle, \langle\ tt, nil,\ 0\rangle\rangle,$

$\}$

we shall write a table of transition rules such as the following:

$\langle nil, q_0\rangle \rightarrow \langle nil, nil, nil\rangle$

$\langle\ 0, q_0\rangle \rightarrow \langle\ tt, nil,\ 0\rangle$

This does not in itself simplify the notation it simply gives the transitions more of an operational flavour.

To simplify the notation we shall not include in our transition tables those transitions associated with busy waiting (i.e. transitions whose right hand side are all nil's). Thus we adopt the convention that any cause not included in the transition table is assumed to be paired with a busy wait.

In addition we simplify our notation by use of transition schemas. For example the schema

$\forall x \in \omega$

$\langle x, q_0\rangle \rightarrow \langle\ tt, nil, x\rangle$

corresponds to the transition table

$\langle 0, q_0\rangle \rightarrow \langle\ tt, nil, 0\rangle$

$\langle 1, q_0\rangle \rightarrow \langle\ tt, nil, 1\rangle$

$\langle 2, q_0\rangle \rightarrow \langle\ tt, nil, 2\rangle$

Using our new notation the PLUS node described earlier becomes

$< \{ \; q \; \}, q, 2, 1, T >$

where T is given by the following transition schema

$\forall x,y \in \omega \;\; <x,y,q> \; -> \; < tt, tt, nil, x+y>$

More General nodes

All the examples of nodes given in the previous section possess the property that each cause determines a single effect. Nodes with this property we call determinate nodes. In addition the nodes of the previous section were all deterministic with respect to their input/output behaviour. In all previous models of dataflow with the exception of Wiedmer[49], all determinate nodes are deterministic with respect to their input/output behaviour. However in our model a determinate node's input/output behaviour may be non-deterministic. These nodes we call time sensitive or time dependent nodes and they are characterized by the ability to compute on empty buffers. In Keller's language[29] it is nodes that use the empty buffer test that may be time senstive. In the previous section we saw an example of a determinate node that computed on empty buffers, namely example 3. the following is an example of a node that computes on empty buffers and is time sensitive:

Example 4

This node has one internal state, one input arc, one output arc and if its input buffer is full it erases the buffer and outputs a copy of the erased daton. However if the input buffer is empty it outputs a zero (i.e. it computes on an empty buffer). Thus if this node receives inputs faster than it can process them, then it will compute the identity function. However if there are delays to the input they will cause spurious zeros to appear as a part of the node's output. Formally we have:

$< \{ \; q \; \}, q, 1, 1, T >$

where T is given by

$\forall x \in \omega$

$<nil,q> \; -> \; <nil,nil, \; 0>$

$< \; x,q> \; -> \; < tt,nil, \; x>$

<u>Non-sequential nodes</u>

All of the example nodes we have given so far can all be written up using Gilles Kahn's simple language for parallel programming. However the following two examples both of which compute history functions cannot be programmed in Kahn's language.

<u>Example 5</u>

The first of these examples is the DOUBLEID node. This node has one internal state, two input arcs and two output arcs and echos on its first output arc the datons that it receives on its first input arc and echos on its second output arc the datons that it receives on its second input arc. There is no sequential process (i.e. Kahnian or Arnoldian) that corresponds to this node. The DOUBLEID node has the following formal definition:

$< \{ q \}, q, 2, 2, T >$

where T is

$\forall x,y \in \omega$

$< x,nil,q> \rightarrow < tt,nil,nil, x,nil>$

$<nil, y,q> \rightarrow <nil, tt,nil,nil, y>$

$< x, y,q> \rightarrow < tt, tt,nil, x, y>$

A more complex example of a non-sequential node is the following:

Example 6

The following is an example of a non-sequential node with determinate input/output behaviour. The node we define is called 'parallel or' but first let's look at the 'simple or' whose formal definition is

$< \{ q \}, q, 2, 1, R >$

where R is all the transitions of the form

$<x,y,q> \rightarrow < tt, tt,nil, x \text{ or } y >$

$\forall x,y \in \{ 1,0 \}$

(1 and 0 denote True & False respectively)

The simple 'or' awaits for a daton to arrive on both input arcs and on arrival they are both consumed and their logical 'or' is output. The parallel version of this node takes advantage of the following equalities:

$1 \text{ or } y = 1$

$x \text{ or } 1 = 1$

$\forall x,y \in \{ 1,0 \}$.

Our parallel version awaits the arrival of a daton in either buffer as soon as a 1 arrives on either input, it outputs a 1. For the sake of argument let us assume that a 1 arrives in the left buffer; the node erases the 1 and outputs a copy without waiting for the corresponding right input. Our node then records using internal memory that it is one ahead on the left input. If another 1 arrives in the left input and still nothing arrives in the right input then another 1 is output and the node records that it is two ahead on the left input. The node can carry on like this indefinitely or until a 0 arrives in the left input in which case it must allow right hand input to catch up. This is only half of the explaination, the other can be extracted from the following formal definition

$$\langle \{ B_i \mid i \in \omega \}, B_0, 2, 1, R \rangle$$

where R is

$$\forall x,y \in \{ 1,0 \}$$

$$\begin{array}{llll}
\langle x, & y, & B_0\rangle & \rightarrow \langle tt, tt, nil, x \text{ or } y\rangle \\
\langle 1, & nil, & B_0\rangle & \rightarrow \langle tt, nil, B_1, 1\rangle \\
\langle nil, & 1, & B_{2i}\rangle & \rightarrow \langle nil, tt, B_{2i+2}, 1\rangle \\
\langle 1, & nil, & B_{2i+1}\rangle & \rightarrow \langle tt, nil, B_{2i+3}, 1\rangle \\
\langle 1, & y, & B_{2i+1}\rangle & \rightarrow \langle tt, tt, nil, 1\rangle \\
\langle x, & 1, & B_{2i+2}\rangle & \rightarrow \langle tt, tt, nil, 1\rangle \\
\langle x, & nil, & B_{2i+2}\rangle & \rightarrow \langle tt, nil, B_{2i}, nil\rangle \\
\langle x, & 0, & B_{2i+2}\rangle & \nearrow \\
\langle nil, & y, & B_{2i+3}\rangle & \rightarrow \langle nil, tt, B_{2i+1}, nil\rangle \\
\langle 0, & y, & B_{2i+3}\rangle & \nearrow \\
\langle nil, & y, & B_1\rangle & \rightarrow \langle nil, tt, B_0, nil\rangle \\
\langle 0, & y, & B_1\rangle & \nearrow
\end{array}$$

Note that the even states b_2, b_4,... code up the deficit of the left input whilst the odd states b_1, b_3,... code up the deficit of the right input.

In a similar way we could define other non-sequential nodes such as 'parallel and', and 'wise' if-then-else.

Non-deterministic nodes

So far all the nodes we have defined have been deterministic in that each cause has been associated with a single effect. However our formal defintion of node (A1) also allows us to describe nodes that are non-deterministic. The following is a classic example of a non-deterministic node with non-determinate input/output behaviour.

Example 7

The node we shall describe is called the unfair MERGE node. It is used in various other forms in many models of dataflow. For example Davis[17] defines a more general form of this node and calls it an Arbiter cell. Our MERGE node has one internal state, two input arcs and one output arc. The node continuously awaits the arrival of a daton in either of its input buffers, outputing the first daton to arrive in either buffer. If datons arrive simultaneously a random choice is made as to which daton is to be output.

A formal defintion of the unfair MERGE node is the following:

$< \{ \ q \ \}, q, 2, 1, T >$

where T is

$\forall x, x \in \omega$

$< x, nil, q> \rightarrow < tt, nil, nil, x>$

$<nil, y, q> \rightarrow <nil, tt, nil, y>$

$< x, y, q> \rightarrow < tt, nil, nil, x>$

$< x, y, q> \rightarrow <nil, tt, nil, y>$

We usually write the last two entries in the above schema as

$$\begin{tikzcd} <x,y,q> \arrow[r] \arrow[dr] & <nil, tt, nil, y> \\ & < tt, nil, nil, x> \end{tikzcd}$$

The reason for this is that they both have the same <u>cause</u>

The MERGE node is a classic example of a non-deterministic node that has non-determinate input/output behaviour. Unlike other authors we allow our functional nodes to be non-deterministic. However, we shall have more to say about this in the next chapter. The following is an example of a non-deterministic functional node.

<u>Example 7</u>

The following node computes the identity function but the transitions code up different internal activities. One activity is to build up an internal memory (queue) of inputs, and the other is to output stockpiled datons. The node is non-deterministic because each cause has associated two possible effects, one stockpiling the other outputing. A computation in which all but finitely many operations are stockpiling would be 'unfair' and would fail to produce the required outputs. Formally we have the following definition:

$< \{ q_i \mid i \in \omega \} \cup \{ q \}, q, 1, 1, T >$

where T is

$\forall x \in \omega, l \in Sq(\omega)$

$<nil, q> \;\to\; <nil, \quad nil, \quad nil>$

$< x, q> \;\to\; < tt, \quad q_{<x>}, \quad nil>$

$\qquad\qquad\quad < tt, \quad nil, \quad x>$

$<nil, q_l> \;\to\; <nil, q_{tail(l)}, hd(l)>$

$<x, q_l> \;\to\; < tt, \quad q_{l \hat{} x}, hd(l)>$

$\qquad\qquad\quad <nil, q_{tail(l)}, hd(l)>$

B. Closed nets and closed net computation

In this section we develop a precise formal definition for closed dataflow nets. In addition we formally define dataflow computation for the closed nets. A closed net is roughly speaking a directed graph in which every arc has a source and destination node. Since all arcs have a source node the net is unable to receive inputs from its external environment (i.e. there are no input arcs) and in a similar way there are no output arcs. The more complex nets which allow input and output are dealt with in the next section. The following diagram illustrates a

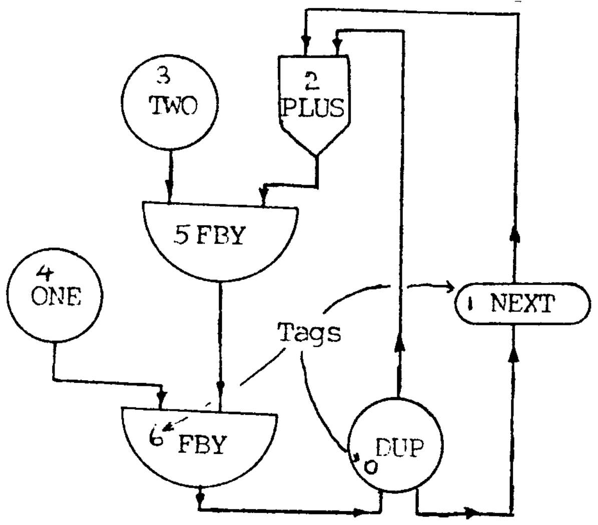


Figure E, A closed dataflow net in which nodes are tagged by natural numbers

typical example of a closed net. In this thesis we deal exclusively with dataflow nets built up from pipes (the arcs) and automata (the nodes). Another possible approach is that of structured nets in which the nodes may themselves denote dataflow nets. This approach would be a generalisation of the named nets used by Adams. As illustrated in the above diagram the nodes of our unstructured net can be labelled with distinct natural numbers. This enables us to uniquely identify each node in a net even if two or more nodes are identical as automata. Moreover if the natural numbers used as tags come from the set n, where n is the number of nodes in a net, then we can denote the nodes in a net as a sequence. For example the sequence associated with the net in the above diagram would be

<DUP, NEXT, +, 2, 1, FBY, FBY>

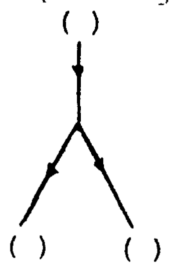
Note in this sequence DUP refers to the automaton that computes the copy function.

In the previous section we saw that within the context of each node it is possible to assign a unique label to each of a nodes input ports and a unique set of labels to each of a nodes output ports. Thus within the context of an entire network the ports of each node are uniquely labelled by an ordered pair <x,y>, where x denotes the node number and y the port number. The set of ordered pairs that uniquely determine each input port is called the set of destinations (denoted by **D**) and the corresponding set for output ports is called the set of sources (denoted by **S**).

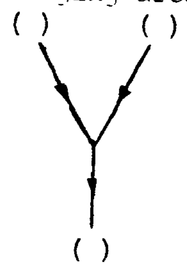
We can now use the set of sources and destinations in an elegant way to define an arc in terms of a source/destination pair. The source being the output port of a node (i.e. an element of **S**) and the destination the input port to a node (i.e. an element of **D**).

Now that we have some idea as to what we mean by an arc and a node we must look at the way in which we want to combine them. Since we are defining closed subnets we must ensure that every source is associated with a destination and that every destination is associated with some source. To ensure that our closed nets are well formed we require that within the set of source/destination pairs that describe the interconnections of a net,no source or destination is included more than once. Thus to ensure that our net is well formed we require that the set of source/destination pairs define a bijective map from **S** to **D.** If this is the case then the following net constructs will be excluded:

i) A splitting arc

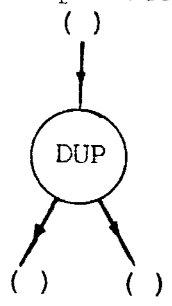


ii) Merging arcs

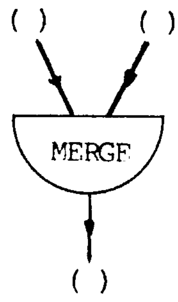


However we get the same effect as i) and ii) by using a) and b) respectively:

a) A duplicator node



b) An unfair merge node



Thus we define a closed net to be a sequence of nodes and a set of source destination pairs such that the number of source pairs is equal to the number of destination pairs. In addition each source determines a unique destination pair and vice versa. This leads us to the following definition of a closed net:

(B1) Definition A closed net is an ordered pair $< \mathbf{F}, \mathbf{A} >$

where

$\mathbf{F} : k \to \mathbf{N}$ where $k \in \omega$ and $\mathbf{N}$ is the set of all nodes.

$\mathbf{A} : S \to D$ is a bijection

where

$S = \{ \langle i,j \rangle \mid i \in \mathrm{dom}(\mathbf{F})$ and $j \in \mathrm{Outportarity}(\mathbf{F}_i) \}$

$D = \{ \langle i,j \rangle \mid i \in \mathrm{dom}(\mathbf{F})$ and $j \in \mathrm{Inportarity}(\mathbf{F}_i) \}$

<u>Example</u>

The diagram below illustrates the closed net described by the following formal definition:

$< \mathbf{F}, \mathbf{A} >$

where

$\mathbf{F} = \langle$ONE, FBY, PLUS, ONE$\rangle$

where

$i \in 4 \quad \mathbf{F}_i \in \mathbf{N}$ (the set of all nodes)

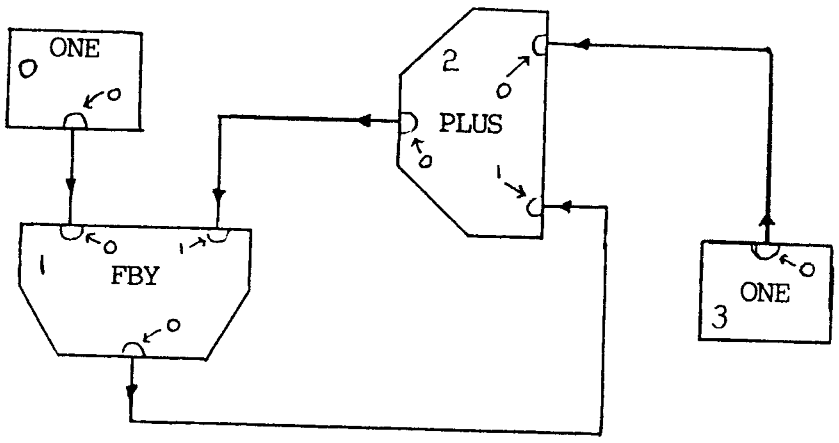
The following table gives the input/output arity of the various nodes:

i	Inportarity($\mathbf{F}_i$)	Outportarity($\mathbf{F}_i$)
0	0	1
1	2	1
2	2	1
3	0	1

$\mathbf{A}$ = { <<0,0>,<1,0>, <<1,0>,<2,1>>,

<<2,0>,<1,1>>, <<3,0>,<2,0>> }

Figure F, A closed Dataflow net showing node and port numbering



The following are auxiliary functions that will facilitate later formalism:

Let M be a closed net < $\mathbf{F}$,$\mathbf{A}$ >

(i) Nodes(M) = $\mathbf{F}$

(ii) Arcs(M) = $\mathbf{A}$

(iii) Size(M) = dom($\mathbf{F}$) (the number of nodes in the net)

Let a (= <<s,n>,<d,m>>) be an element of Arcs(M)

(i) Tonode(a) = d

(ii) Toport(a) = m

(iii) Fromnode(a) = s

Closed Net Computation

Now that we have defined closed nets formally let us turn our attention to closed net computation.

The current state of a closed net is roughly speaking made up of two objects. Firstly a vector of states that records for each node in the net the current state of each node. The second is a function that associates with each arc in the net the current contents of the arcs fifo queue (the queue function). For a closed net in its initial state the node state vector records each node as in its initial state and the queue function records all arcs as being empty. A closed net computation involves the closed net computes in moving from one net state to another via a net transition. A net transition is a vector of node trasnsitions such that the vector contains one transition for each node in the net. A net computation is then a countable sequence of net transitions, a finite sequence defining a partial net computation and an infinite sequence defining a complete net computation. Formally we have the following definitions:

(B2) Definition Let N be a closed net

The state of N is an ordered pair $< \mathbf{S},\mathbf{A} >$

where

$\mathbf{S} \in X_{i \in \mathrm{Size}(N)} \mathrm{States}(\mathrm{Nodes}(N)_i)$

$\mathbf{A} : \mathrm{Arcs}(N) \to \mathrm{Sq}(\omega)$

The initial state of N is a state of N ($< \mathbf{S},\mathbf{A} >$) such that

$\forall a \in \mathrm{Arcs}(N)\ A(a) = \Lambda$

$\forall j \in \mathrm{Size}(N)\ S_j = \mathrm{Initialstate}(\mathrm{Nodes}(N)_j)$

We will also find the following auxiliary functions of use later:

Let N be a closed net

Let T (= < **S**,**A** >) be a state of N

(i) Queues(T) = **A**

(ii) States(T) = **S**

Net computation involves all the nodes in a net performing a node computation. Since each node computes by choosing a compatible node transition we define a net transition to be a sequence of node transitions. This leads to the following formal definition:

(B3) Definition Let N be a net

T is said to be an N-transition iff

$T \in X_{i \in \mathrm{Size}(N)} \mathrm{Transitions}(\mathrm{Nodes}(N)_i)$

A net cannot compute using a random N-transition it must use the transitions available to it in a particular context. For example the Øth node of a net has a state and buffer contents which define a <u>cause</u> it must use a transition that is compatible with that <u>cause</u>, this applies to all nodes in the net. This gives us the following definition of compatibility

(B4) Definition Let N be a net

Let S be a state of N

Let t be an N-transition

t is said to be N-compatible with S

iff

(i) $\forall a \in \text{Arcs}(N)$

$$\text{Buffer}(t_{\text{Tonode}(a)})_{\text{Toport}(a)} = \begin{cases} \text{nil} & \text{if Queue}(S)(a) = \Lambda \\ \text{Queue}(S)(a)|1 & \text{otherwise} \end{cases}$$

(ii) $\forall i \in \text{Size}(N)$

$$\text{State}(t_i) = \text{States}(S)_i$$

We are now in a position to define formally what it means for a node to move from one state to another.

(B5) Definition Let N be a closed net

Let C and D be states of N

D is said to be t-derivable from C over N

$(C \xrightarrow[N]{t} D)$

iff

(i) t is N-compatible with C

(ii) D is such that

a) $\forall$ i $\in$ Size(N)

$$\text{States}(D)_i = \begin{cases} \text{States}(A)_i \text{ if Newstate}(t_i) = \text{nil} \\ \text{Newstate}(t_i) \text{ otherwise} \end{cases}$$

b) $\forall$ a $\in$ Arcs(N)

$$\text{Queues}(B)(a) = \begin{cases} \text{tail}(\text{Queues}(A)(a)\hat{\ }\text{Prod}(t_{\text{Fromnode}(a)})_{\text{Fromport}(a)} \\ \qquad \text{if Erase}(t_{\text{Tonode}(a)})_{\text{Toport}(a)} = \text{tt} \\ \text{Queue}(A)(a)\hat{\ }\text{Prod}(t_{\text{Fromnode}(a)})_{\text{Fromport}(a)} \\ \qquad \text{otherwise} \end{cases}$$

Now that we have formally defined what it means for a node to move from one state to another we can easily define what it means for a node to make a finite or infinite sequence of moves or computation steps.

(B6) Definition Let $k \in \omega$

Let N be a closed net

Let C and D be states of N

D is said to be finitely derivable from C

$(C \xrightarrow[N]{*} D)$

iff

$\exists$ a sequence $\langle E_0, \ldots, E_{k-1} \rangle$ of

N-states and a sequence $\langle t_0, \ldots, t_k \rangle$ of

N-transitions such that

$$C \xrightarrow[N]{t_0} E_0 \xrightarrow[N]{t_1} \ldots \xrightarrow[N]{t_{k-1}} E_{k-1} \xrightarrow[N]{t_k} D$$

(B7) Definition Let N be a closed net

Let C be an ω-sequence of N-states

C is said to be an N-state chain

iff

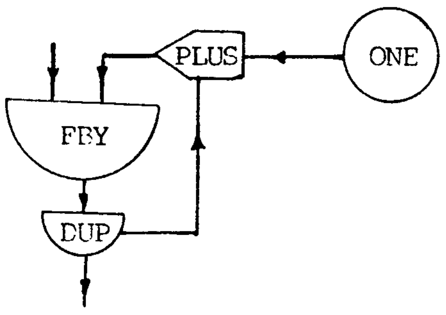
$\exists$ an ω-sequence of N-transitions T

such that $\forall i \in \omega$

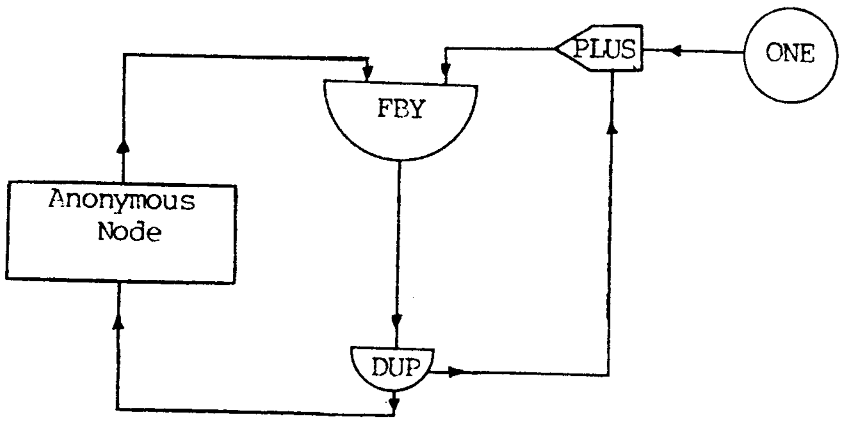
$$C_i \xrightarrow[N]{T_i} C_{i+1}$$

C. Open Nets and Open Net Computation

In this section we will show how the definitions developed in the last section can be applied to open nets and open net computation. The essential difference between an open net and a closed net is that the open nets have arcs which have no source node called the input arcs and arcs which have no destination called the output arcs. A simple open net is illustrated in the following diagram:



To enable us to think of an open net in terms of a closed net we borrow the idea of an environment from Minsky[38]. For our application we can think of the environment as an anonymous dataflow node. Thus any open net when connected to its environment becomes a closed net. For example when we connect the simple net shown above an environment node with one input arc and one output arc we get the closed net illustrated in the following diagram:



The environemnt node turns out to be very useful for two reasons. The first is that it converts an open net into a closed net and thus allows us to define open net computation in terms of closed net computation. The second is that the addition of an environment node allows us to define an open net through a simple extension to the definition of a closed net. The only problem we are faced with is what is an environment node. We shall deal with this problem in the next section and for the moment we think of the environment node as an unknown or anonymous node.

Let us now look at the technical details of using an environment node in the definition of an open net. In the above diagram we see that the anonymous node that was used to close the open net has been tagged with the natural number 4 (i.e. the size of the open net). Now if we recall in definition B1 a closed net is defined to be a sequence of nodes and a set of source/destination pairs (a bijective mapping). The problem with using this definition for open nets is that the source destination pairs do not usually form a bijection. The reason for this is that some of the arcs have no source node and others have no destination node. However, we can use the idea of an environment node to over come this technical difficulty. The above diagram shows that with the addition of an environment node all arcs once again have unique sources and destinations. Although we have used the environment node to allow us to describe input and output arcs in terms of sources and destination pairs we need not consider the environment node as being a part of the open net. These ideas give rise to the following formal definition:

(C1) Definition An open net (subnet) is a sequence $< \mathbf{F}, \mathbf{A}, n, m >$

such that

$n,m \in \omega$

$\mathbf{F} : k \rightarrow \mathbf{N}$ $k \in \omega$ and

$\mathbf{N}$ is the set of all nodes

$\mathbf{A} : S \rightarrow D$ a bijection

where

$S = \{ \; <i,j> \mid i \in dom(\mathbf{F}) \text{ and } j \in Outportarity(\mathbf{F}_i) \; \}$

$\cup \; \{ \; <dom(\mathbf{F}),k> \mid k \in m \; \}$

$D = \{ \; <i,j> \mid i \in dom(\mathbf{F}) \text{ and } j \in Inportarity(\mathbf{F}_i) \; \}$

$\cup \; \{ \; <dom(\mathbf{F}),k> \mid k \in m \; \}$

The formal definition for the simple open net illustrated above is:

$< \mathbf{F}, \mathbf{A}, 1,1>$

where $\mathbf{F} = < 1, +, FBY, DUP >$

	$Inportarity(\mathbf{F}_i)$	$Outportarity(\mathbf{F}_i)$
$i = 0$	0	1
$i = 1$	2	1
$i = 2$	2	1
$i = 3$	1	2

$\mathbf{A} = \{ \; <<0,0>,<1,1>>,<<1,0>,<2,1>>,$

$<<4,0>,<2,0>>,<<3,0>,<4,0>>,$

$<<2,0>,<3,0>>,<<3,1>,<1,0>> \; \}$

We now introduce some auxiliary functions that we make use of later.

Let ρ be an open net $< \mathbf{F},\mathbf{A}, n,m>$

(i) $\text{Inportarity}(\rho) = n$

(ii) $\text{Outportarity}(\rho) = m$

(iii) $\text{Size}(\rho) = \text{dom}(\mathbf{F})$

(iv) $\text{Arcs}(\rho) = \mathbf{A}$

(v) Let a $(= <<s,n>,<d,m>>) \in \mathbf{A}$

(a) $\text{Fromnode}(a) = s$

(b) $\text{Fromport}(a) = n$

(c) $\text{Tonode}(a) = d$

(d) $\text{Toport}(a) = m$

(vi) $\text{Internalarcs}(\rho) = \{ \ a \in \mathbf{A} \ | \ \text{Fromnode}(a) \neq \text{Size}(\rho)$ and $\text{Tonode}(a) \neq \text{Size}(\rho) \ \}$

(vii) $\text{Transitions}(\rho) = \mathbf{X}_{i \in \text{Size}(\rho)} \text{Transitions}(\text{Nodes}(\rho)_i)$

(viii) $\text{Nodestates}(\rho) = \mathbf{X}_{i \in \text{Size}(\rho)} \text{States}(\text{Nodes}(\rho)_i)$

(ix) $\text{Inputarcs}(\rho) = \{ \ a \in \mathbf{A} \ | \ \text{Fromnode}(a) = \text{Size}(\rho) \ \}$

(x) $\text{Outputarcs}(\rho) = \{ \ a \in \mathbf{A} \ | \ \text{Tonode}(a) = \text{Size}(\rho) \ \}$

The fact that any open net can be attached to an environment node of compatible input/output arity means that the problem of defining open net computation has been reduced to that of closed net computation. As we already have a formal definition of closed net computation the only remaining problem is to define precisely what we mean by an environment node. However before we do this we introduce one more definition which we shall make use of later.

(C2) Definition Let ρ be an open net

The internal state of ρ is an ordered pair

$< \mathbf{S}, \mathbf{A} >$

where $S \in \text{Nodestates}(\rho)$

$A: \text{Internalarcs}(\rho) \rightarrow \text{Sq}(\omega)$

The initial state of ρ (Initialstate(ρ))

is a state of ρ (= <S,A>) such that

$\forall a \in \text{Arcs}(\rho) \quad A(a) = \Lambda$

$\forall j \in \text{Size}(\rho) \; S_j = \text{Initialstate}(\text{Nodes}(\rho)_j)$

D. Environment Nodes and Test Beds

In this section we complete the definition of open net functionality by giving a precise meaning to the notion of an environment node. We know roughly speaking that the behaviour of an open net's real environment is non-determinate. By this we mean that the real environment can behave in an unpredicatable manner. The only possible description of the real environment is that it supplies datons to the input arcs of an open net at an unknown rate and removes datons from the output arcs of an open net at an unknown rate. However since our operational semantics is supposed to be completely general we should be able to simulate, using one of our nodes, the behaviour of an open net's real environment. The node we require is the one that is able to simulate all possible input/output behaviours. In fact using the notation we developed in section A we have no trouble in defining such a node. None of the other models we have described is able to define the environment node.

For example an open net with two input arcs and one output arc is closed by an environment node with 1 input arc and two output arcs. the following is a formal definition of an environment node with one input arc and two output arcs:

$< \{ \ q \ \}, q, 1, 2, T >$

where T is

$\forall x,y,z \in \omega \cup \{ \text{nil} \}$

$\forall a,b \in \{ \text{tt}, \text{nil} \}$

$<x,y,q> \ \text{->} \ <a,b,\text{nil},z>$

The environment node just described can be used with any open net with 2 input arcs and one output arc. In general an open net may have n input arcs and m output arcs and so if we want to close any given open net we shall require the following family of environment nodes,

Environment(n,m):

$\forall n,m \in \omega$

$< \{ \; q \; \}, q, n, m, T >$

where T is

$\forall x_0,\ldots,x_{n-1},z_0,\ldots,z_{m-1} \in \omega \cup \{ \text{nil} \}$

$\forall a_0,\ldots,a_{n-1} \in \{ \text{nil},\text{tt} \}$

$<x_0,\ldots,x_{n-1},q> \;\texttt{->}\; <a_0,\ldots,a_{n-1},\text{nil},z_0,\ldots,z_{m-1}>$

Thus given any open net we can close it by connecting the net to a compatible environment node. As a consequence we are able to use the definition of closed net computation to describe open net computation.

To make this idea more precise we introduce the concept of a test bed function. A test bed function has as its domain the set of all open nets and as its range the set of all closed nets. To be more precise the test bed function takes an open net and a compatible environment node and maps it to the corresponding closed net. The test bed function is formally defined as follows:

(D1) Definition Let $\mathbf{O}$ be the set of all open nets

Let $\mathbf{C}$ be the set of all closed nets

Let $\rho \in \mathbf{O}$

Let $n = \mathrm{Inportarity}(\rho)$

Let $m = \mathrm{Outportarity}(\rho)$

$\tau : \mathbf{O} \rightarrow \mathbf{C}$ (the test bed function)

such that

$\tau(\rho)$ is a closed net with

$\mathrm{Nodes}(\tau(\rho)) = \mathrm{Nodes}(\rho) \wedge \mathrm{Environment}(n,m)$

$\mathrm{Arcs}(\tau(\rho)) = \mathrm{Arcs}(\rho)$

E. The Encapsulation Property

In this section we prove that any open net can be replaced in any pipeline dataflow context by a black box equivalent single node. Informally speaking, two compatible open nets are black box equivalent iff given the same inputs at the same rate they produce and consume the same datons at the same rate.

(E1) Theorem (the encapsulation property)

Let ρ be an open net

There exists a single node (N) which is black box equivalent to ρ.

Proof We can construct N as follows:

(i) The node N and the open net ρ have the same number of input and output arcs. Inportarity(N) = Inportarity(ρ) Outportarity(N) = Outportarity(ρ)

(ii) The states of node N are the internal states of the open net ρ
States(N) = States(ρ)

(iii) The initial state of N is the initial state of ρ
Initialstate(N) = Initialstate(ρ)

(iv) The transitions of N are the internal transitions of ρ
Transitions(N) = Transitions(ρ) (i.e. via some coding)

Now that we have shown that N exists we prove that it is black box equivalent to ρ.

Let ρ' be the open net consisting of the node N and its
input/output arcs.

For any ω-configuration sequence for $\tau(\rho)$ there exists an ω-configuration sequence $\tau(\rho')$ in which the behaviour of the environment is equivalent to that of $\tau(\rho)$ and such that ρ and ρ' are indistinguishable as black boxes.

This must be the case since ρ' is always able to simulate (via the transition coding) any possible transition of ρ.

Thus the two open nets are black box equivalent.

Note: since the two nets are
indistinguishable as black boxes within the context of a testbed (hence the τ's in the above) they are indistinguishable within the context of any pipeline dataflow context.

QED

Chapter 3

Relating our operational semantics for pipeline dataflow to Kahn's Denotational semantics for pure dataflow

In the last chapter we formulated a completely general operational semantics for pipeline dataflow and thus achieved one of our goals. In this chapter we will formulate a completely general definition of what it means for an open net to compute a history function thus achieving another of our goals. Thus the single most important result of this chapter is the characterization of open net functionality in terms of a two player infinite game of perfect information. This definition of functionality we claim is completely general and we use it to prove several important results. The first result is that the history functions computed by functional open nets are continuous in the sense of Kahn, the second is that every continuous history function is the function computed by some node and finally that every open net can be replaced in any pure dataflow context by some node. However before we do this we include two brief sections in which we briefly explain some of the important ideas underlying our results. The first section describes some elementary background information on the fixed point theory of recursion and the second gives more details of Kahn's denotational semantics for pure dataflow.

A. Some Mathematical Preliminaries

To understand Kahn's denotational semantics for pure dataflow it is necessary to have at least some elementary knowledge of the fixed point theory of recursion. In this section we present some of the basic ideas underlying this theory. For a more complete presentation the reader is referred to Bird[10] or Manna[50].

We begin with some elementary definitions.

(A1) Definition A binary relation $\leq$ over a set S

is a partial ordering of S

iff

$\forall x,y,z \in S$

(i) $x \leq x$ (reflexive)

(ii) $x \leq y$ and $y \leq x$ implies $x = y$ (anti-symmetric)

(iii) $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)

The structure $< S, \leq >$ is called a partially ordered set or poset for short.

For the sake of brevity and when the context is unambiguous we will denote the poset $< S , \leq >$ by its set i.e. S.

(A2) Definition Let $< S , \leq >$ be a poset,

Let x be an infinite sequence of elements of S,

x is said to be an increasing chain

iff

$\forall i \in \omega \; x_i \leq x_{i+1}$.

(A3) Definition The structure $< S , \leq >$ is a domain or countable chain complete partial order (C3PO)

iff

(i) $< S, \leq >$ is a poset,

(ii) S contains a minimal element,

(iii) Any increasing chain in S has a lub in S,

In the introduction to this thesis we defined the set Ka of countable sequences of natural numbers. In Ka we included the empty sequence $\wedge$. The following is a binary relation which we will make great use of in this and subsequent chapters.

(A4) Definition $\subseteq$ is a binary relation over Ka such that

$\forall x,y \in Ka \quad x \subseteq y$

iff

x is an initial segment of y.

For example $<1,2,3,4> \subseteq <1,2,3,4,5,6>$.

(A5) Proposition The structure $< Ka, \subseteq >$ is a poset

Proof Straightforward.

In this and subsequent chapters we shall also use an extended form of the binary relation $\subseteq$. The extension we have in mind is the following:

(A6) Definition Let $n \in \omega$,

$\subseteq_n$ is a binary relation over Ka^n such that

$\forall x,y \in Ka^n \quad x \subseteq_n y$

iff

$\forall i \in n \; x_i \subseteq y_i$

(A7) Proposition For any $n \in \omega$

the structure $< Ka^n, \subseteq_n >$ is a C3PO

Proof (i) $< Ka^n, \subseteq_n >$ is a poset, straightforward,

(ii) $\{ \Lambda \}^n$ is the least element,

(iii) Every increasing chain in Ka^n has a lub in Ka^n, straightforward.

In the introduction to this thesis we stated that in the denotational semantics nodes would be associated with functions over countable sequences of natural numbers (Ka). We will thus make use of the following definitions:

(A8) Definition Let $< A, \leq_A >$ and $< B, \leq_B >$ be C3PO's,

Let $f : A \rightarrow B$,

f is said to be monotonic

iff

$\forall a,b \in A$

$a \leq_A b$ implies $f(a) \leq_B f(b)$.

(A9) Definition Let $< A, \leq_A >$ and $< B, \leq_B >$ be C3PO's,

Let $f : A \rightarrow B$,

f is said to be continuous

iff

(i) f is monotonic,

(ii) For every increasing chain x in A

$$f(\bigsqcup_{i \in \omega} x_i) = \bigsqcup_{i \in \omega} f(x_i)$$

The concept of continuous function, C3PO and fixed point (i.e. a fixed point of a function h is any value x for which $h(x) = x$) are all brought together in the following important theorem due to Kleene.

(A10) Theorem (The first recursion theorem, Kleene)

Let $\langle D, \leq_D \rangle$ be a C3PO,

Let f: D -> D be a continuous function,

f possesses a fixed point x given by

$x = \bigsqcup_{n \in \omega} f^n(\delta_d)$

where δ_D is the least element in D and

$f^n(\delta_d) = \underbrace{f(\ldots.f(f(\delta_D))\ldots)}_{\text{n-times}}.$

Moreover x is the least fixed point of f under the partial ordering $\leq_D$. This theorem is used by Gilles Kahn[26] to give a denotational semantics for pure dataflow.

B. Kahn's Denotational Semantics for Pure Dataflow

In our opinion there is no doubt that one of the single most important contributions to the study of dataflow was the Kahn principle. For completeness we include, in this section, a brief description of Kahn's work [26].

Kahn noted that it is possible to associate a countable sequence, called a history with each arc in a pipeline dataflow net. The history of an arc denotes the entire sequence of objects to have travelled upon that arc. The order of the objects in the sequence corresponds to the order in which the objects travelled along the arc. In Kahn's model different arcs can carry objects of different types and thus the type of the objects in a history is determined by the type of the arc (e.g. integer, matrix, real ...etc). However in our model the only objects that flow along arcs are natural numbers and thus a history is simply a countable sequence of natural numbers or in other words an element of Ka. We should point out that it is essential not to confuse a queue with a history. A queue is a dynamic (operational) object that grows and shrinks as a result of the computational activity of the nodes attached to either end of an arc. On the other hand a history is a static (denotational) object that is the entire record of all the datons to have travelled along an arc.

A further observation of Kahn was that it is possible for computing stations with memory of their own to compute functions from the histories of their inputs to the history of their output. In other words a node with n input arcs and 1 output arc could compute a function from Ka^n to Ka. Kahn refers to these functions as history functions.

We should point out that in the operational model we developed in the last chapter it is possible to describe nodes that do not compute history functions; one such node is the unfair MERGE node. The history of the MERGE node's inputs determines a large set of possible outputs, some would say it computes a relation. Thus the goal we pursue in the remainder of this chapter, is that of choosing a subset of our model in which the nodes all compute history functions. In Kahn's restricted operational model all the nodes compute continuous history functions. Kahn explains the restriction to functional pipeline dataflow nodes in the following terms:

(i) Monotonicity means that receiving more input at a computing station can only provoke it to send out more output. Indeed it is a crucial property since it allows parallel operation. A machine may not need all input to start computing, since future inputs concern only future outputs.

(ii) Continuity prevents any station from deciding to send some output only after it has received an infinite amount of input.

We call dataflow nets in which all the nodes compute history functions <u>Pure Dataflow nets</u>.

<u>Fixpoint equations</u>

Rather than study the behaviour of a complex machine, Kahn wanted to study the properties of the solution to a set of equations. To achieve this he associated with each parallel program schema p (dataflow graph) a set Σ_p of equations over sequence domains, in such a way that a set of sequences is a possible solution to Σ_p iff it is a possible set of histories for the arcs of the dataflow graph. Kahn gives the following rules for constructing Σ_p:

(i) to each arc h in the net associate a variable x_h

(ii) if $x_0,\ldots,x_{n-1}$ are the variables associated with the input arcs and $i_0,\ldots,i_{n-1}$ are the set of sequences fed in as inputs include the equations

$$\begin{aligned} x_0 &= i_0 \\ &\vdots \\ x_{n-1} &= i_{n-1} \end{aligned}$$

(iii) for each node f, with n input arcs $(x_0,\ldots x_{n-1})$ and m output arcs

$(y_0,\ldots,y_{m-1})$ include the m equations

$$\begin{aligned} y_0 &= f_0(x_0,\ldots,x_{n-1}) \\ \vdots\ \ & \qquad \vdots \\ y_{m-1} &= f_{m-1}(x_0,\ldots,x_{n-1}) \end{aligned}$$

Clearly, the histories of the arcs of the graph p have to satisfy the system Σ_p. Moreover since Σ_p is a set of fixpoint equations over a C3PO where the operators are continuous, such a system admits a unique minimal solution, the least fixed point solution (see A10).

The minimum solution $\{\ y(x_0),\dots,y(x_{n-1})\ \}$ of the system

$\Sigma_p = \{\ x_i = \tau_i(x_0,\dots,x_{n-1})\, i \in n\ \}$

where the τ_i terms are continuous operators

is given by

$\bigsqcup_{i \in \omega} (x_0,\dots,x_{n-1})$

where

$x_i = \Lambda \quad i \in n$

$x_i = \tau_i(x_0,\dots,x_{n-1}) \quad i \in n \quad j \in \omega$

An alternative and more consise way of writing this expression is by using the μ operator. If f is a function from a C3PO D into itself then the least fixed point of f is given by the following expression

$\mu\, x\, f(x) \quad (= \bigsqcup_{i \in \omega} f^i(\Lambda_D)\ ).$

Thus the solution for the system of equation Σ_p becomes:

$\mu x\ \tau(x).$

The following is an example of a dataflow net and its associated set of equations. Note that the Upper case letters in the graph denote the names of nodes and the corresponding lower case letters in the set of equations denote the function(s) computed by the node.

$$\begin{tikzcd}
 & {} \arrow[d, "x_\emptyset"] & & \\
 & F \arrow[dl, "x_1"'] \arrow[dr, "x_2"] & & \\
G \arrow[rr, "x_3"] \arrow[dr, "x_4"'] & & H \arrow[r, "x_7"] \arrow[dl, "x_5"] & {} \\
 & K \arrow[d, "x_6"] & & \\
 & {} & &
\end{tikzcd}$$

Open net p

$$\left.\begin{array}{ll}
x_\emptyset & = i \\
x_1 & = f(x_1,x_2) \\
x_2 & = g_\emptyset(x_1) \\
x_3 & = g_1(x_1) \\
x_4 & = k_\emptyset(x_4) \\
x_5 & = k_1(x_4) \\
x_6 & = h_1(x_3,x_5) \\
x_7 & = h_\emptyset(x_3,x_5)
\end{array}\right. \quad \Sigma\ p$$

The principle that the operational behaviour of a pure dataflow net is exactly described by the least fixed point solution to the nets equation we call the Kahn Principle. Many computer scientists believe that the Kahn principle was proved by Kahn. However Kahn never published a formal proof of this principle nor did he define precisely the concepts of "node", "net" etc. In the remainder of this chapter we formulate a precise definition of what it means for a pipeline dataflow node to compute a history function.

iii Using 'where' notation together with certain transformation rules to massage expressions involving fixed point operators

The first to use 'where' notation in Computer Science was P.J. Landin in his renowned paper "The next 700 Programming Languages" (see Landin[33]). In this thesis we use 'where' notation as a meta-language to prove certain results about the operational behaviour of nets. Our 'where' notation is a minor variant of Landin's 'where', namely our 'where' clause corresponds to Landin's 'where rec'. In his paper Landin shows how 'where' expressions can be thought of as a formal system in which precise proofs can be carried out using a number of transformation rules. A simple example of a 'where' expression is the following:

```
 a where
          a = c where
                      c = 1
                end
   end
```

In this example it is obvious that the value of the variable 'a' is 1. The following example is more complex:

```
g  where
         g = F(a) where
                       a  = 1
                   end
         a = 2
   end
```

If we were using Landin's 'where' the value of the above expression would be **F**(2) since Landin's 'where' associates the outermost occurence of the variable 'a' with the 'a' in **F**(a). However, our 'where' corresponds to Landin's 'where rec' and so it would associate the innermost 'a' with **F**(a). Thus the value of the our 'where' expression is **F**(1). In A.ii we saw how to associate a set of equations with a pure dataflow net. In the chapter 4 we will prove that the operational behaviour of a pure dataflow net is exactly the least fixed point of the net's associated set of equations. To do this we will need to prove that different expressions involving the least fixed point operator μ are equivalent. We saw in section A.ii how the expression

$\mu\ x\ \tau(x)$

corresponds to the least fixed point solution to a set of equations.

In this thesis we use the transformations defined below to massage expressions which involve μ (the fixed point operator). A simple example of such an expression is

μ x f(x)

See section A.i for more details about C3PO's. The transformation rules given below can all be verified using standard fixed point theory (e.g. see De Bakker[8]).

<u>Our transformation rules</u>

(a0) Given the fixed point expression:

μ a **F** (a)

we can express this, without changing the meaning, using the following 'where' notation:

a where a = **F** (a)

We call this transformation μ elimination.

(a1) Conversley, given a 'where' expression:

a where a = **F** (a)

we can express this, without changing the meaning, as the following fixed point expression:

μ a **F** (a)

We call this transformation μ introduction.

(b0) Given a 'where' expression we can add a new equation to the right hand arm of the 'where' without changing the meaning of the expression so long as the new equation defines a variable not already used. For example we can add the equation z = 10 to

```
a where
        a = 7
  end
giving us
a where
       a = 7
       z = 10
  end
```

Both 'where' expression have the same value.

(b1) Conversely, we can remove an equation provided the variable defined by the equation is not used anywhere else in the 'where' expression.

(c0) Another transformation is the substitution of a variable for its definition. This does not change the meaning of the 'where' expression.

(c1) Conversely, given an expression which is equal to some variable, it is possible to change the expression for the variable without changing the meaning of the 'where' expression.

(d0) Finally, as our 'where' expressions are defined over a C3PO any variable defined in terms of itself corresponds to the recursive definition and so the fixed point theorem (A10) applies. Hence we can take a fixed point without changing the meaning of the 'where' expression.

For example:

```
a where
      a = b
      b = F(b)
 end
```

can be transformed into:

```
a where
      a = b
      b = μ b F(b)
 end
```

without changing the meaning of the 'where' expression.

(dl) Conversley, given an equation which contains a fixed point operator in the normal form (i.e $x = \mu x f(x)$), we can remove the fixed point operator, leaving a recursive definition. This transformation does not change the meaning of the 'where' expression.

The following is an example of how we prove that

$\mu A\ \mathbf{F}(A) = \mu A\ \mathbf{F}(\ \mathbf{F}(A)\ )$

$\mu A\ \mathbf{F}(A)$

$\downarrow$ (μ elimination)

A where A = $\mathbf{F}(A)$

$\downarrow$ (substituition of a variable for its definition)

A where A = $\mathbf{F}(\ \mathbf{F}(A)\ )$

$\downarrow$ (μ introduction)

$\mu A\ \mathbf{F}\ (\ \mathbf{F}\ (A)\ )$

QED

C. Open Net Funcionality

In this section we examine what it means to say that an open net computes a history function. To be more precise, what does it mean to say that an open net, with n input arcs and one output arc, computes a function $f:Ka^n \rightarrow Ka$.

Functionality is an extemely important property as it allows us to reason about an open net in terms of a simple mathematical object, namely a history function. This is in contrast to the complex operational nature of an open net (i.e. internal net state, net transitions etc.).

To begin with let us look at functionality in nodes. Remember that a node together with its input and output arcs is an an open net. Given a node we know, roughly speaking, that the node computes a history function if and only if the node's entire output activity is determined only by the node's entire input activity. As we stated earlier there are two ways in which a node may fail to be functional:

(i) if there is randomness within the node itself

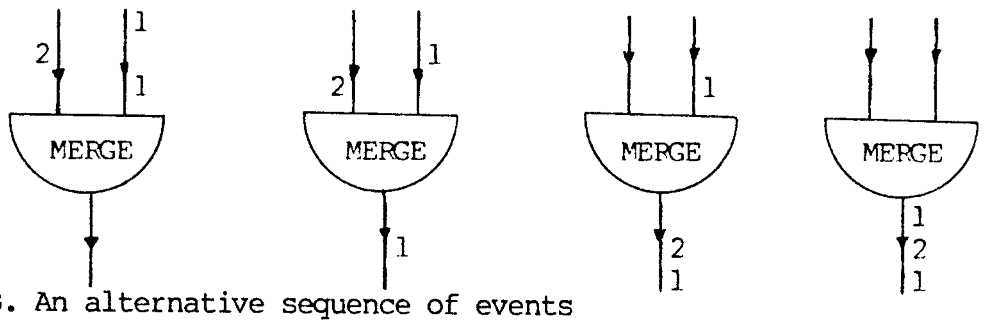
(ii) if the rate of input of datons effects more than the rate of output of datons (i.e. the node is time-sensitive)

The classic example of a node that is both time-sensitive and random is the unfair MERGE node. For convenience we restate its formal definition:

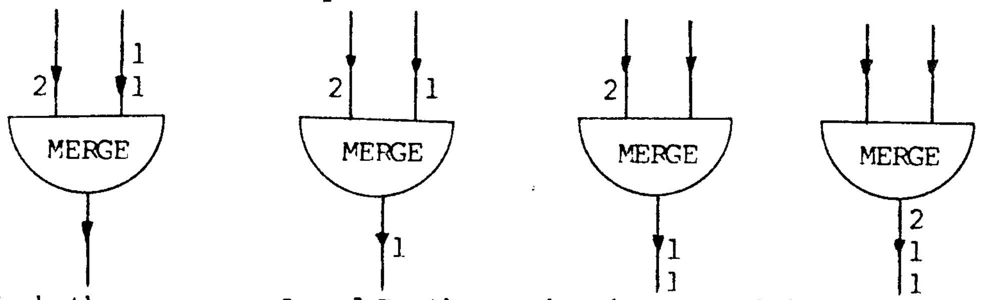
```
< { q },q,2,1,T>
   where T is
      ∀ x,y ∈ ω
      <  x,nil,q> -> < tt,nil,nil,  x>
      <nil,  y,q> -> <nil, tt,nil,  y>
      <  x,  y,q> -> < tt,nil,nil,  x>
                     <nil, tt,nil,  y>
```

The merge node is time sensitive in that the node's output activity is dependent upon the rate of arrival of the node's inputs. The following diagrams illustrate this time dependency:

A. A possible sequence of events

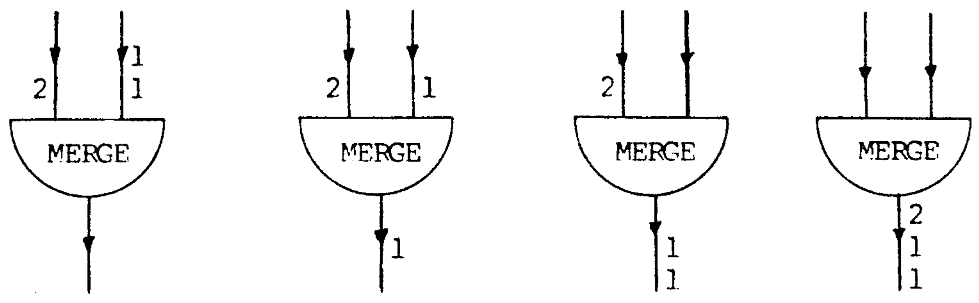


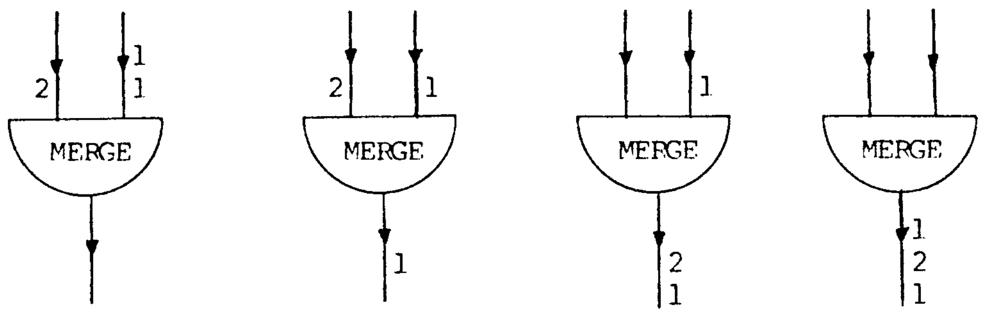
B. An alternative sequence of events



In both sequences A and B, the entire input activity of the merge node is (<2>,<1,1>). In sequence A the input datons arrive in the following order (Λ,<1>), (<2>,<1>) and (<2>,<1,1>). This results in the node outputing the sequence <1,2,1>. In sequence B the same inputs arrive in a different way namely (<2>,Λ), (<2>,<1>) and (<2>,<1,1>). This results in the node outputing the sequence <2,1,1>. Thus the example show that the merge node is indeed time-senstive.

On the other hand if the sequences (<2>,<1,1>) are placed on the node's input arcs at the beginning of the computation, hence preventing any time sensitive behaviour, we see the node's randomness at work. Here are two possible sequences of events





In both cases the node starts with the same inputs but produces a different output for each sequence. The reason for this is that the node had to choose non-deterministically between the following two transitions:

$\forall x,y \in \omega$

$$\begin{aligned} < x, y, q> \;->\; & < tt,nil,nil,x> \\ & <nil, tt,nil,y> \end{aligned}$$

Other authors who have given a definition of node functionality have usually assumed the following naive definition of node functionality.

A node is said to compute a history function f iff given α as input there exists an infinite computation sequence which accumulates f(α) on the nodes output arcs.

In addition these authors insist that the node begins computation with all the inputs it is ever to receive already queued up on the nodes input arcs.

For a variety of reasons this "obvious" definition turns out to be both unrealistic and incorrect.

To begin with a node will never (when in use) have an infinite sequence of datons on its input arcs. On the contrary the input arcs are usually empty to begin with and even at intermediate stages in the computation. The contents of the input arcs of any 'real' node is determined by the rate at which the node's environment supplies inputs and the rate at which the node consumes inputs. The proposed definition fails because it requires that a node functions correctly (i.e. computes f) only when supplied datons at a faster rate than that at which the node consumes input. It could be argued that the above definition is adequate if nodes are incapable of computing on "empty buffers" and we would agree with this. As our nodes may compute on "empty buffers" we require a more general definition of functionality.

There is an interesting way to repair the inadequacy of the above definition and although we do not use the repaired version it will be of interest to examine how we could repair the definition.

The above definition requires that all the datons that a node ever receives are placed on the nodes input arcs before compu$\overset{t}{\wedge}$ation begins. This usually means placing an infinite sequence of natural numbers on each of the nodes input arcs. Our main objection to this is that it is unrealistic because datons arrive one by one and often with pauses between successive datons. These pauses mean that those nodes that can compute on an "empty buffer" would do so and often with disasterous effect. For example consider the node that produces a zero when its buffer is empty and outputs a copy of its buffer otherwise. This node has the following formal definition:

< { q }, q, 1, 1, T >

where T is

$\forall x \in \omega$

<nil,q> -> < nil,nil, 0>

< x,q> -> < tt,nil,x>

If we initialise this node's input arc with the infinite sequence <1,2,3,4,5,...> , then according to the above definition this net would compute the identity function. If convinced of this by the above definition we incorporated this "identity node" into a 'real' data flow net then it could use its "empty buffer" computation with disastrous effect. For example if there was a delay before the arrival of the first daton (say a six) then one or more zeros may have been output before the six was output.

To repair our hypothetical definition of functionality we extend the notion of a history to include timing information. To do this we introduce the notion of a unit of delay called a hiaton (Greek for pause). This term was coined by E. A. Ashcroft and W. W. Wadge and is denoted by '*'. Now instead of initialising an input arc with a sequence from Ka (the set of histories) we initialise an input arc from the set $(\omega \cup \{ * \})^{\omega}$ (the set of hiatonic sequences). We can then restate the definition of functionality as follows:

A net is said to compute a history function f iff given any hiatonic sequence α as input there exists a computation sequence that produces f(del(α)) as output.

Where del is a function that removes all hiatons from a hiatonic sequence thus producing a pure history.

It is possible to formalise functionality using hiatonic, however, we shall not do so prefering to stay with pure histories, at least util we have proved the Kahn principle. However we shall use a a denotational semantics based on hiatonics in chapter 5 when we look at ways to extend Kahn's denotational semantics to handle a broader class of operational behaviours. Let us now return to the main theme of this section namely the formulation of a precise definition for node functionality.

A second problem with the obvious definition of node functionality proposed earlier is that it requires only that f(α) be possible as the output history, but not necessary. Since in our operational model functional nodes may be non-deterministic, the distinction between possible and necessary is crucial. We can certainly define a node that can output a random sequence of datons and in fact the environment node with n input arcs and 1 output arc would, according to our hypothetical definition, compute every history function $f:Ka^n \rightarrow Ka$.

The fact that our functional nodes may be non-deterministic means that we cannot repair the last problem by requiring that every sequence of transition produce f(α) as output. This requirement would be too restrictive because it rules out any sort of control or direction of the activity of a node. Such control is however necessary because our nodes are non-deterministic devices capable of doing more than one activity (e.g. input and output) at the same time. If computation proceeds at random one vital activity may be neglected even though the computation as a whole never stops. We call such a situation "livelock" a term coined by E. A. Ashcroft[54].

A good example of a node which is non-deterministic and yet functional is the node that computes the identity function but whose transitions code up different internal activities. One activity is to build up an internal stockpile (a queue), another is to output stockpiled datons. The formal definition of this node was given in chapter 2 (example 7 p.75). The node is obviously non-deterministic and any computation sequence in which all but finitely many operations are stockpiling would be in livelock and hence fail to produce the required output.

We could of course avoid all these problems by restricting ourselves to deterministic and sequential nodes, but we reject this course since we would like to define the class of all dataflow nodes that compute history functions, including non-sequential and non-deterministic nodes.

With this in mind we formulate a more "dynamic" version of the obvious "static" definition above.

The problem with the above definition of functionality is that it assumes that the underlying computation is sequential. We can think of sequentiality in terms of our node transtions as associating with each "cause" only one "effect".

However in this thesis we are looking for a more general definition, one which makes sense even when the underlying computation is non-sequential and non-deterministic. In other words our definition should also be applicable to nodes which allow more than one possible effect to be associated with each cause. Example 7 on page 75 defines a non-deterministic node that computes the identity function and in which each cause is associated with two effects. One effect is to internally stockpile datons and the other is to output internally stockpiled datons. If we choose transitions using a strategy that chooses only finitely many output effects then the node will not compute the identity fuction because of livelock. (i.e. it will spend most of its time building up an internal stockpile of datons that it may never be able to output).

From this example it is obvious that the definition of functionality that we require must be one which allows a node to be used in conjunction with a "fair" strategy for avoiding livelock.

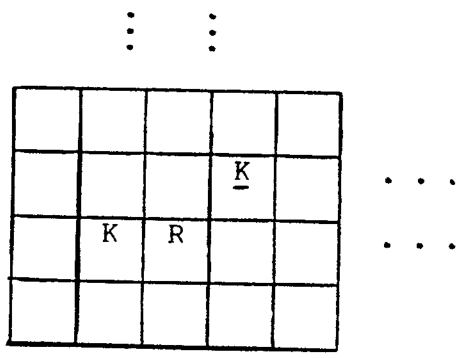
Our definition of node functionality is defined in terms of a two player infinite game of perfect information. However before we deal with infinite games or our definition of functionality we should point out that whatever we have said in this section about nodes is also applicable to open nets in general.

D. Infinite Games

In this section we present some background information on the theory of infinite games. However for more detail the reader is referred to Gale and Stewart[18] and Wadge[47].

An infinite game is a game in which a particular 'round' need not terminate, so that the outcome of the contest can only be determined by examining the entire history of the contest.

Simple but interesting examples of infinite games can be constructed by extending the standard chessboard infinitely in one or more directions, and by suitably modifying the rules. Consider, for example, the position shown in the diagram (the board extends infinitely in the direction of the dots).



White's goal is to checkmate Black's king, and Black's goal is to avoid checkmate. This game is genuinely infinite because Black can win, but cannot achieve certain victory after any finite number of moves, i.e. we cannot in general conclude that Black has won without examining the entire record of the game.

This game nevertheless has a finite aspect in that one of the players (White) cannot win without terminating the game: we might call such a game "half-finite". But it is easy to devise games which are not even half-finite. For example we could retain the above board and position but change the rules so that White's goal is instead to get arbitrarily far away from the Black king, i.e. to play so that no matter how large an integer n is, there will be a point in the game after which White's king will never be less than n moves away from the Black king. Then clearly neither player can ever win in any finite number of moves, and it will always be necessary to look at the entire history of the game to determine the winner.

It is not difficult to give a precise definition of 'infinite game' provided we restrict ourselves to games in which

(i) there are only 2 players.

(ii) each player on each move has only countably many choices for his next move.

(iii) there are no infinite stages in the game i.e. all rounds are of length ω.

The history of a particular 'round' of such a game can be 'coded-up' as a pair $<\alpha, \beta>$ of elements of ω^{ω} and thus the game is completely determined by the subset $\mathbf{W}$ of $\omega^{\omega} \times \omega^{\omega}$ consisting of all histories of codes of bouts in which 'II' is the winner. We therefore assume for simplicity that each player on each move plays a natural number, and define the game to be the set $\mathbf{W}$ itself.

(D1) Definition An (infinite) game is a subset of $\omega^\omega \times \omega^\omega$

We can now formalise the notion of a strategy. It is clear that a strategy for 'II' in one of these games is a function from Sq to ω which takes as its argument the sequence $\langle\alpha(\emptyset),\alpha(1),\ldots,\alpha(n-1)\rangle$ for each n of 'I's first n moves and gives as its result player 'II's nth move $\beta(n)$. For our purposes it is more convenient to have a startegy for 'II' yield the entire history $\langle\beta(\emptyset),\beta(1),\ldots,\beta(n-1)\rangle$ of 'II's moves up to that point. Strategies for player 'I' are defined in a similar manner.

(D2) Definition

(i) A strategy for 'II' is a monotonic function γ from Sq to Sq s.t. $|\gamma(s)| = |s|$ for every s in Sq

(ii) A strategy of 'I' is a monotonic function ρ from Sq to Sq s.t. $|\rho(s)| = |s|+1$ for every s in Sq.

note we often abbreviate Sq(ω) to Sq.

It should be noted that this defintion implies that the games we are studying are games of perfect information, i.e. games in which each player has complete knowledge of his opponents moves up to that point.

Now if γ is a strategy for 'II' , we let γ denote the corresponding function from ω^ω to ω^ω which takes as its argument the entire history of 'I's moves and gives as its result the entire history of 'II's moves. Thus γ is a winning strategy for 'II' for the game **W** iff $\langle\alpha,\gamma(\alpha)\rangle$ is in

W for every α, and the notion of a winning strategy for 'I' is similarly defined.

(D3) Definition For any monotonic function γ from Sq to Sq and any $\alpha \in \omega^\omega$

$$\tilde{\gamma}(\alpha) = \bigcup_{k \in \omega} \gamma(\alpha|k)$$

note that if γ is a strategy (either for 'I' or for 'II') then $\gamma(\alpha)$ will be in ω^ω for every α.

(D4) Definition For any game **W**

(i) A winning strategy for 'II' for **W** is a strategy γ for 'II' such that $\langle\alpha,\tilde{\gamma}(\alpha)\rangle \in$ **W** for every α in ω^ω.

(ii) A winning strategy for 'I' for **W** is a strategy $\bar{\sigma}$ for 'I' such that $\langle\tilde{\bar{\sigma}}(\beta),\beta\rangle \in -$ **W** for every β in ω^ω.

The study of the infinite almost always concerns in some way or another, the question of determinedness. A game is determined iff one of the players has a winning strategy (i.e. if the game determines a winner) Since every finite game is determined, and since also draws are not possible in infinite games (as we have defined them) it might be plausible to conclude that every infinite game is determined. This conclusion is however not justified. The question of determinedness is also beyond the scope of this thesis and the reader is refered to Wadge[47].

Now it is certainly true that it cannot be the case that both players 'I' and 'II' have winning strategies for a game **W.** Given two strategies σ and γ for 'I' and 'II' respectively, we can 'play them off' against each other and form a unique element $\langle\alpha,\beta\rangle$ of $\omega^\omega \times \omega^\omega$ called the clash of σ and γ. This is a term introduced by the logician J. Addison.

If $\langle\alpha,\beta\rangle$ is the clash of σ and γ then $\alpha = \tilde{\sigma}(\beta)$ and $\beta = \tilde{\gamma}(\alpha)$. If σ and γ are both winning strategies the clash $\langle\alpha,\beta\rangle$ would have to be in both $\mathbf{W}$ and $-\mathbf{W}$, impossible. Thus given two strategies for 'I' and 'II' respectively, one must be 'superior' to the other.

This argument does not however, imply that every game is determined. It may be that given any strategy for 'I', player 'II' has a strategy which is superior, but that given, any strategy for player 'II', player 'I' has a strategy which is superior. In fact it is possible using the unrestricted axiom of choice, to construct (by 'diagonalising' over strategies) a game which is not determined.

E. Infinite Games and Open Net Functionality

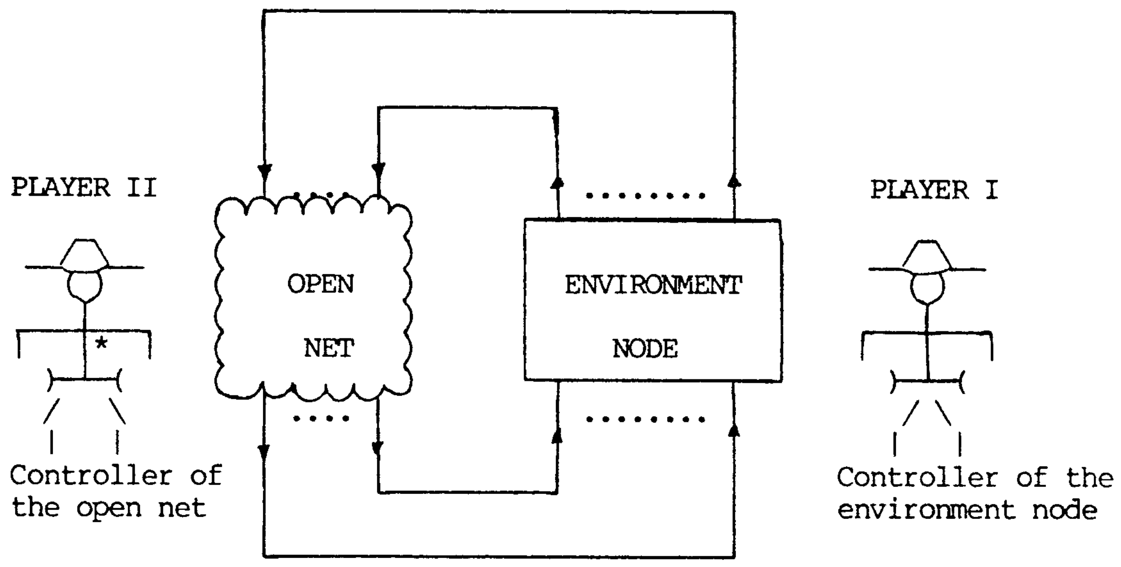
In this section we relate certain kinds of infinite games and open net functionality. The use of infinite games allows us to express in a natural way, the dynamic nature of net computation. As a consequence we are able to give a natural and completely general definition of open net functionality.

In the customary manner we begin by giving an informal, anthropomorphic explanation of the connection between infinite games and open net functionality. Let us assume that an argument has arisen between two individuals whom we shall call 'I' and 'II'. The argument is whether a certain open net computes a particular history function. Indivitual 'II' claims that the open net computes a history function. However individual 'I' demands that he be allowed to test out the open net on some sample inputs to see if it really does compute a history function.

Individual 'II' accepts 'I's challenge and produces an open net which is packaged to look like a black box with the open net's input and output arcs protruding. Player 'II' then invites player 'I' to make a move by placing a daton on some or all the black boxes input arcs. After 'I' has placed inputs on the black boxes input arcs player 'II' moves by choosing a compatible open net transition. The effect of this transition may cause a daton to fall out of some or all of the black boxe's output arcs. The contest between these two individuals then continues in this fashion ad infinitum.

The goal of 'I' in this contest is to discredit 'II's black box and so 'I' may try all kinds of activity to test out whether 'II's black box really does compute a particular history function. For example 'I' may try to supply datons at different rates hoping there is a time sensitive node within 'II's black box, he may completely stop placing datons on the input arc hoping that 'II's black box will move into a state that over produces output. If at any finite stage the accumulated outputs of 'II's black box are not an initial segment of the particular history function applied to the accumulated inputs then 'I' will be able to discredit 'II's black box. If 'I' is unable to discredit 'II's black box at any finite stage he may be able to do it by examining the entire history of the contest. To do this player 'I' examines the entire input to 'II's black box and if this input was α and player 'II' claimed that his black box computed the history function f then the entire output of 'II's black box must be $f(\alpha)$. If this is the case then 'II's claim is totally correct and 'I' is the loser. However if 'II's black box produces as output only an initial segment of $f(\alpha)$ then 'II's claim is only partially correct and thus 'I' is able to discredit 'II's black box.

Let us now make some mathematical sense of this anthropomorphic description. The situation just described fits in very naturally with the formalism we developed in the last chapter. In particular the context in which the game is played is well suited to the idea of open nets computing within test beds. If we place this game in the context of a test bed we can think of player 'II' as a controller of the open net and player 'I' as the controller of the environment node. This situation is illustrated in the following diagram:


PLAYER II
Controller of the open net
OPEN NET
ENVIRONMENT NODE
PLAYER I
Controller of the environment node

(Note: it is customary in infinite games to think of player'II' as the 'good' player and 'I' as his 'evil' opponent) In this context player 'II' makes a move by choosing a compatible open net transition and player 'I' makes a move by choosing an environment node transition.

The two players are now able to engage in their contest, an infinite game. Player 'II' (the open net controller) must use a strategy which ensures that the open net produces the correct output. This strategy must be totally correct no matter how the input arrives from the environment node (i.e. no matter what rate the controller of the environment node decides to produce datons). The fact that an open net computes a history function does not mean that player 'II' succeeds no matter what choices he makes (remember open nets may be non-determinate allowing a choice of open net transitions); it means only that he has a strategy to ensure success in his battle against a hostile opponent (player 'I').

Our definition of open net functionality formalises this anthropomorphic view in terms of winning strategies for two player infinite games. The idea that nodes require controlling strategies in order to choose transitions suggests an infinite game with the following rules

Let δ be an open net with n input arcs and m output arc.

Let f: $(Ka^n \to Ka)^m$

The rules of the infinite game $G(f,\delta)$ are:

(i) The game begins with the open net in its initial state (i.e. all the nodes in their initial internal state and all the internal arcs empty.). and the input/output arcs of the open net empty. In other words $\tau(\delta)$ (the testbed for δ) is in its initial state.

(ii) The two players alternate in making moves 'I' playing first.

(iii) On each of his moves 'I' places a daton on some or all of the open nets input arcs (possibly none). What 'I' really does is choose an environment node transition which has the effect of producing datons on some or all of the open nets input arcs. In addition 'I's choice of transtion may result in some datons being consumed from the open net's output arcs.

(iv) Each of 'II's moves involves choosing one of a nonempty set of possible open net transitions. In other words the controller of the open net chooses a compatible open net transition. There will always be at least one possible transition to choose from and usually more than one.

(v) Given that 'I's infinite sequence of moves produced an input history $\alpha \in Ka^n$. Player 'II' wins iff 'II's infinite sequence of moves produce for each output arc i ($\in$ m) the output history

$f_i(\alpha)$.

Note that in conventional infinite game theory a move for a player is usually associated with a natural number (see section D). However, in the our infinite game we have said that a move for player 'II' is a net transition (similarly for player 'I'). To be precise we should have labelled each possible net transition with a natural number (notice that the set of net transitions is countable). In this way Player 'II's choice of natural number would be associated with some open net transition (similarly for Player 'I'). To avoid this coding we choose to describe a move directly in terms of an approppriate transition. Thus the following definitions of strategy take a move as being some net transition, in the case of player 'II' and an environment node transition in the case of player 'I'.

In our game a strategy for player 'II' is a monotonic function γ that takes a finite sequence of moves for 'I' and produces a finite sequence of (responses) moves for 'II'.

(E1) Definition Let δ be an open net

Let n = Inportarity(δ)

Let m = Outportarity(δ)

A strategy for 'II' for G(f,δ) is a monotonic function

γ: (Transitions(Environment(n,m)))^{Sq} -> (Transitions(δ))^{Sq}

such that

$\forall \alpha \in$ (Transitions(Environment(n,m)))^{Sq}

$|\gamma(\alpha)| = |\alpha|$

(E2) Definition Let σ be an open net with

n input arcs and m output arcs

Let $f:(Ka^n \to Ka)^m$

Let **O** be the set of all finite and infinite sequences of moves for 'II' in $G(f,\sigma)$

Let **I** be the set of all finite and infinite sequences of moves for 'I' in $G(f,\sigma)$

(i) Output: $(\mathbf{O} \to Ka)^m$

such that $\forall\, o \in \mathbf{O}\; j \in m$

$$\mathrm{Output}_j(o) = \prod_{i \in \mathrm{dom}(o)} \mathrm{Prod}(o(i)_{\mathrm{Fromnode}(y(j))})(\mathrm{Fromport}(y(j)))$$

where $y(j) = \langle \mathrm{Arcs}^{-1}(\sigma)(\mathrm{Size}(\sigma),j), \langle \mathrm{Size}(\sigma),j\rangle\rangle$

(ii) Input: $(\mathbf{I} \to Ka)^n$

such that $\forall\, t \in \mathbf{I}\; j \in n$

$$\mathrm{Input}_j(t) = \prod_{i \in \mathrm{dom}(t)} \mathrm{Prod}(t_i)_j$$

Note $\prod_{i \in n} x_i = x_\emptyset \hat{}\, x_1 \hat{} \ldots \hat{}\, x_{n-1}$ n finite

$x_\emptyset \hat{}\, x_1 \hat{}\, x_2 \hat{} \ldots$ n infinite

(note $a\hat{}\,nil = nil\hat{}\,a = a$)

(E3) Definition Let $\bar{\sigma}$ be an open net with n input arcs and m output arcs

Let $f:(Ka^n \rightarrow Ka)^m$

Let γ be a strategy for 'II' for $G(\bar{\sigma},f)$

Let $\mathbf{I}$ be the set of all infinite sequences of moves for 'I'.

Let $A = \{ a \in \text{Environment}(n,1) \mid \text{Input}(a) = \alpha \}$

(i) γ is said to be a totally correct strategy for 'II'

iff

$\forall a \in A \ \text{Output}(\tilde{\gamma}(a)) = f(\alpha)$

(ii) γ is said to be a partially correct strategy for 'II'

iff

$\forall a \in A \ \text{Output}(\tilde{\gamma}(a)) \subseteq f(\alpha)$

The use of infinite games allows the following definition of functionality:

(E4) Definition An open net $\bar{\sigma}$ is said to compute a history function f iff there exists a totally correct strategy for player 'II' in $G(f,\bar{\sigma})$ and any other strategy γ for 'II' is a partially correct strategy.

F. Some Results

In this section we prove some results about open nets that compute history functions. For notational simplicity we prove the results with respect to open nets that compute functions of the form:

$f:Ka^n \to Ka$. The methods used in proving these results extend readily to the more general case in which nodes compute functions of the form

$f: (Ka^n \to Ka)^m$ where $n,m \in \omega$.

(F1) Theorem Every history function computed by an open net is monotonic.

Proof Let $n \in \omega$

Let ρ be an open net such that

Inportarity(ρ) = n

Outportarity(ρ) = 1

Let $f : Ka^n \to Ka$ be the function computed by ρ

Let γ be a totally correct strategy for 'II' in $G(f,\rho)$

Let $\alpha , \beta \in Ka^n$ such that $\alpha \subseteq \beta$

Let $\mathbf{B} = \{ B \in \text{Environment}(n,1)^\omega \mid \text{Input}(B) = \beta \}$

Let $\mathbf{A} = \{ A \in \text{Environment}(n,1)^\omega \mid \text{Input}(A) = \alpha \}$

(see E2 for a definition of the function Input)

In what follows we prove that $f(\alpha) \subseteq f(\beta)$.

To begin with we know that $\alpha \subseteq \beta$ and thus for any sequence $A \in \mathbf{A}$ we can find a sequence $B \in \mathbf{B}$ such that B simulates the moves of A up to any finite number of moves.

i.e. $\forall n \in \omega \forall A \in \mathbf{A} \exists B \in \mathbf{B} \quad A|n = B|n$

If we apply γ to this we get that

$\forall n \in \omega \forall A \in \mathbf{A} \exists B \in \mathbf{B}$

$\mathrm{Output}(\gamma(A|n)) = \mathrm{Output}(\gamma(B|n)) \subseteq \mathrm{Output}(\tilde{\gamma}(B)) = f(\beta)$

(note $\tilde{\gamma}(A) = \bigcup_{k \in \omega} \gamma(A|k)$ and thus

$\tilde{\gamma}(A)|n = \gamma(A|n)$)

Hence $\mathrm{Output}(\tilde{\gamma}(A)|n) \subseteq f(\beta)$

Since Output is continuous we have that

$\exists n' \in \omega\, n' \leq n \quad \mathrm{Output}(\tilde{\gamma}(A))|n' = \mathrm{Output}(\tilde{\gamma}(A)|n) \subseteq f(\beta)$

thus $\quad f(\alpha) \subseteq f(\beta)$

QED

(F2) Theorem Every history function computed by an open net is continuous.

Proof Let $n \in \omega$

Let ρ be an open net such that

Inportarity(ρ) = n

Outportarity(ρ) = 1

Let $f : Ka^n \rightarrow Ka$ be the function computed by ρ

Let γ be a totally correct strategy for 'II' for $G(f,\rho)$

Let C be any increasing chain in Ka^n

Let $\mathbf{A}^i = \{ A^i \in \text{Environment}(n,1)^\omega \mid \text{Input}(A^i) = C_i\}$ for all $i \in \omega$

Let $\mathbf{A} = \{A \in \text{Environment}(n,1)^\omega \mid \text{Input}(A) = \bigsqcup_{j \in \omega} C_j\}$

We prove that $\bigsqcup_{i \in \omega} f(C_i) = f(\bigsqcup_{i \in \omega} C_i)$

Since f is monotonic (F1) we have that

$\bigsqcup_{i \in \omega} f(C_i) \sqsubseteq f(\bigsqcup_{i \in \omega} C_i)$

However let us assume that

$\bigsqcup_{i \in \omega} f(C_i) \sqsubset f(\bigsqcup_{i \in \omega} C_i)$

We can express this in terms of our infinite game $G(f,\rho)$ as:

$\forall j \in \omega, A^j \in \mathbf{A}^j, A \in \mathbf{A}$

$\bigsqcup_{i \in \omega} \text{Output}(\tilde{\gamma}(A^i)) \sqsubset \text{Output}(\tilde{\gamma}(A))$.

Thus for any $A \in \mathbf{A}$, $j \in \omega$, $A^j \in \mathbf{A}^j$

$\tilde{\gamma}(A)$ is able to produce at some finite stage

an output that cannot be produced by any

$\tilde{\gamma}(A^k)$ for any $k \in \omega$;

but this cannot be true since

$\forall n \in \omega \ \forall A \in \mathbf{A} \ \exists k \ \exists A^k$ s.t. $A|n = A^k|n$

$$(\text{Input}(A) = C_\emptyset \cup C_1 \cup \ldots = \text{Input}(A_\emptyset) \cup \text{Input}(A_1) \cup \ldots)$$

Thus whatever is output at some finite stage

by $\tilde{\gamma}(A)$ can also be output at some finite

stage by $\tilde{\gamma}(A^k)$ for some $k \in \omega$ and some $A^k \in \mathbf{A}^k$.

To be more precise:

$\forall n \in \omega \forall A \in \mathbf{A} \exists k \in \omega, A^k \in \mathbf{A}^k$

$$\text{Output}(\tilde{\gamma}(A)|n) = \text{Output}(\tilde{\gamma}(A^k)|n)$$

Hence

$$\bigsqcup_{i \in \omega} f(C_i) \not\subset f(\bigsqcup_{i \in \omega} C_i)$$

Thus it must be the case that

$$\bigsqcup_{i \in \omega} f(C_i) = f(\bigsqcup_{i \in \omega} C_i)$$

QED

(F3) Theorem Every continuous history function is the function computed by some node (i.e. some atomic open net).

Proof Let $n \in \omega$

Let $f: Ka^n \to Ka$

We construct a node that when placed together with its input and output arcs into a test bed computes the function f according to E4

Such a node (= N) is defined by the following:

(i) Inportarity(N) = n

(i.e. the function has Ka^n as its domain)

(ii) Outportarity(N) = 1

(i.e. the function has Ka as its range)

(iii) $\text{State}(N) = \{ \langle \alpha,\beta,l \rangle \mid \alpha \in Sq^n \text{ and } \beta \in Sq \text{ and } l \in \omega \}$

The α component records the history read so far; β records what has been output so far; l is the running total of datons output so far.

(iv) $\text{Initialstate}(N) = \langle \{ \Lambda \}^n, \Lambda, \emptyset \rangle$

(v) Transitions(N) are given by the following transition schema:

$\forall a \in (\omega \cup \{ nil \})^n$, $b \in \omega \cup \{ nil \}$,

$k \in \omega$, $\alpha \in Sq^n$, $\beta \in Sq$

$$\langle a, \langle \alpha, \beta, k \rangle \rangle \rightarrow \langle E, \langle \alpha\hat{\ } a, \beta\hat{\ }\mathbf{f}(\alpha)(k), k' \rangle, \mathbf{f}(\alpha)(k) \rangle$$

where $i \in n$

$$E_i = \begin{cases} tt \text{ if } a_i \not\in nil \\ ff \text{ otherwise} \end{cases}$$

$$k' = \begin{cases} k+1 \text{ if } |\beta\hat{\ }\mathbf{f}(\alpha)(k)| > |\beta| \\ k \quad \text{otherwise} \end{cases}$$

Let us place this node (i.e. N) together with its input and output arcs (we call this the open net ρ') into a test bed. Once in the test bed we can begin to play the infinite game $G(f,\rho')$. Since the function f is continuous we know that giving the node more input can only cause it to produce more output (i.e. monotonicity) Thus given a few inputs we know that we can safely produce the outputs required by the function definition. In other words future inputs will not require us to recall those datons already output. Secondly we know that the node will never require an infinite amount of input to produce some output.

Informally the strategy for Player 'II' in this game is to consume inputs whenever possible and to produce an output whenever possible. Thus on 'II's n^{th} move he will have produced $f(\alpha|n)|n$ as output where α is the result of the first n moves of 'I'. Since the node defined above is deterministic player 'II' is never in the postion where he has a choice of moves. Thus the formal strategy of 'II' is entirely determined by the moves of 'I'. However no matter what moves are made by 'I' the strategy of 'II' is such that on 'II' n^{th} move he will have output $f(\alpha|n)|n$ where α is the input produced by the first n moves of 'I'. Thus 'II' strategy is a totally correct strategy.

QED

(F4) Theorem Every pure dataflow open net can be replaced in any pure dataflow context by a pure dataflow node

Proof A direct consequence of theorems F2 and F3

QED

The so called 'encapsulation' property described in Arnold[4] is based on a theorem similar to F4.

Chapter 4

A Proof of the Kahn Principle in a Completely General Context

In this chapter we give, for the first time, a proof of the Kahn principle for finite pure dataflow nets. Others who have attempted to prove the Kahn principle have tried to do so directly. This is extremely complicated as it involves direct reasoning about open net computation (e.g. fair ω-sequences of net transitions, etc..). Our proof of the Kahn principle makes use of infinite games and as a result the proof is less complex than it would have been had we used a direct approach.

The Kahn principle has many important consequences one of which is that sets of equations can be thought of as an equational programming language similar to Lucid[7]. This type of programming language has been ignored by many designers of dataflow languages. Another consequence of the Kahn principle is that we can use the simple denotational semantics to reason about our operational ideas. A good example of this is the cycle sum test of Wadge[46]. These consequences are briefly described in section B.

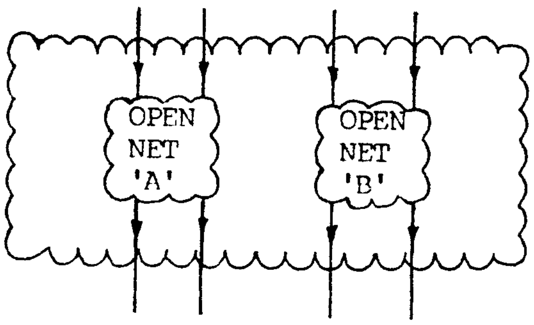
A. A Proof of The Kahn Principle

In this section we show how all well-formed open pure dataflow nets can be build up using two particularly simple operations. One of these operations is the placing side by side of two open nets to form a larger open net, we call this juxtaposing two nets. The second operation is called iteration or looping. The idea here is that a net can be formed from an existing open net by choosing one of the subnets output arcs and bending it back to feed an arbitrary input arc. Some dataflow groups do not allow dataflow nets with loops (e.g. Hankin[24]). The reason for this is that dataflow nets which allow re-cycling (looped arcs) may deadlock. We deal with the problem of deadlock in section B.

Our approach to proving the Kahn principle is in a number of stages. To start with we define what it means for an open net to be Kahnian (i.e. it computes the least fixed point of the net's associated set of equations). Similarly we say that a net is output Kahnian or O-Kahnian iff its output behaviour is as predicted by the least fixed point of the net's associated set of equations. The first results we prove are that the operations of juxtaposing and iteration preserve O-Kahnity. Using these results we prove by induction of the size of a net that all finite pure dataflow nets are O-Kahnian. The Kahn principle is established by the use of a simple lemma which states that an open net is Kahnian iff it is O-Kahnian. We now present the technical details of a proof of the Kahn principle.

Juxtaposition

We begin by defining precisely what it means to juxtapose (i.e. place side by side) two open nets.



The Compostion of two open nets

Let **a** and **b** be two open nets.

c = **a:b** (read **a** juxtaposed **b**)

iff

a:b is the unique net **c** such that

(i) Inportarity(**c**) = Inportarity(**a**) + Inportarity(**b**)

(ii) Outportarity(**c**) = Outportarity(**a**) + Outportarity(**b**)

(iii) Nodes(**c**) = Nodes(**a**)$\hat{}$ Nodes(**b**)

(iv) Arcs(**c**) = $\{ \langle\langle n+S,i\rangle,\langle m,j\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in \mathbf{J} \} \cup$

$\{ \langle\langle n,i\rangle,\langle m+S,j\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in \mathbf{P} \} \cup$

$\{ \langle\langle n+S,i+N\rangle,\langle m+S,j\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in \mathbf{I} \} \cup$

$\{ \langle\langle n+S,i\rangle,\langle m+S,j+M\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in \mathbf{O} \} \cup$

$\{ \langle\langle n+S,i\rangle,\langle m+S,j\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in \mathbf{B} \} \cup$

Internalarcs(**a**)

where S = Size(**a**)

N = Inportarity(**a**)

M = Outportarity(**a**)

J = Inputarcs(**a**)

P = Outputarcs(**a**)

I = Inputarcs(**b**)

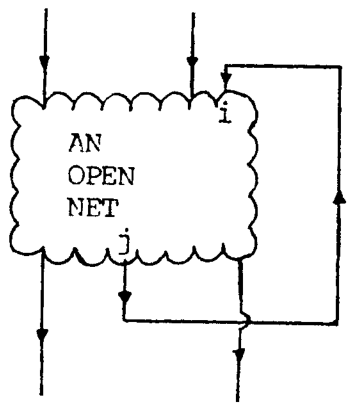
O = Outputarcs(**b**)

B = Internalarcs(**b**)

This large expression simply does re-naming. The internal arcs of **a** need no re-naming so they appear as they are; the input arcs of **a** need to have their source node updated to the size of the new net; similarly for the output arcs of **a.** In the juxtaposed net **b** must be completely re-labelled. This means that the input arcs of **b** must not only have there source node updated (i.e. to the size of the new environment node); but also the port numbers must be re-labelled so that they begin from the Inportarity(**a**). A similar argument applies to the output arcs of **b.** The internal arcs of **b** must be re-labelled to take account of their new position in the node sequence that defines the juxtaposed net. This is also the reason that the nodes associated with the input and output arcs of **b** are re-labelled.

Iteration

Another important operation over open nets is the iteration operation. This involves taking an arbitrary output arc and bending it back to an arbitrary input arc, as illustrated in the diagram:



Bending back the j^{th} output arc to the i^{th} input arc.

We define this operation precisely as follows:

Let **c** be an open net

Let i,j ∈ Inportarity(**c**),Outportarity(**c**) respectively

c' is the new subnet formed by bending back the j^{th} output arc of **c** to feed the i^{th} input arc of **c.**

(i) Inportarity(**c'**) = Inportarity(**c**) - 1

(ii) Outportarity(**c'**) = Outportarity(**c**) - 1

(iii) Nodes(**c'**) = Nodes(**c**)

(iv) Arcs(**c'**) = Internalarcs(**c**) $\cup$ { <i',j'> } $\cup$

{ <<n-1,p>,<m,q>> | <<n,p>,<m,q>> ∈ **I** & (p<i or p>i) } $\cup$

{ <<n,p>,<m-1,q>> | <<n,p>,<m,p>> ∈ **O** & (q<j or q>j) }

where **I** = Inputarcs(**c**)

O = Outputarcs(**c**)

i' = Arcs(**c**)$^{-1}$(size(**c**),i)

j' = Arcs(**c**)(size(**c**),j)

In this case the only re-labelling is to the input/output arcs.

(A1) Lemma Any pure dataflow open net definable in our operational model can be build using a combination of

i) Juxtaposition: the placing side by side of two open nets to form a new open net.

ii) Iteration: the bending back of an arbitrary output arc to an arbitrary input arc within the same open net.

Proof Layout all the nodes in the net using juxtaposition and apply iteration to make neccesary interconnections.

QED

Relating our dataflow nets to sets of fixed point equations

If we are to state precisely what we mean to say that a net is Kahnian then we must be able to relate our dataflow nets with a set of fixed point equations. To formalise this idea we associate two functions with each open net **c**. One function we call $\mathbf{E}^{\mathbf{c}}$ and the other $\mathbf{F}^{\mathbf{c}}$. The function $\mathbf{E}^{\mathbf{c}}$ associates a countable sequence of natural numbers with each internal arc. The actual value of the sequence depends on the history function computed by the source node of the arc (and of cause that nodes input arcs). In a similar way the function $\mathbf{F}^{\mathbf{c}}$ associates a countable sequence of natural numbers with each output arc. Formally we have the following:

Let **c** be a pure dataflow open net

n = Inportarity(**c**)

m = Outportarity(**c**)

C = Internalarcs(**c**)

O = Outputarcs(**c**)

We associate two functions with **c**

(i) $\mathbf{E^C} : Ka^{\mathbf{C}} \times Ka^n \rightarrow Ka^{\mathbf{C}}$

$\forall q \in \mathbf{C} \quad x \in Ka^n$

$\mathbf{E^C}(\mathbf{C},x)(q) = \text{Fnodes}(\mathbf{c})_{\text{Fromnode}(q)}(\mathbf{C},x)$

where Fnodes(**c**) is the sequence of functions that correspond to the sequence Nodes(**c**) of functional nodes.

(ii) $\mathbf{F^C} : Ka^{\mathbf{C}} \times Ka^n \rightarrow Ka^m$

$\forall x \in Ka^n \; q \in \mathbf{O}$

$\mathbf{F^C}(\mathbf{C},x)(\text{Toport}(q)) = \text{Fnodes}(\mathbf{c})_{\text{Fromnode}(q)}(\mathbf{C},x)$

Where Fnodes(**c**) is as above.

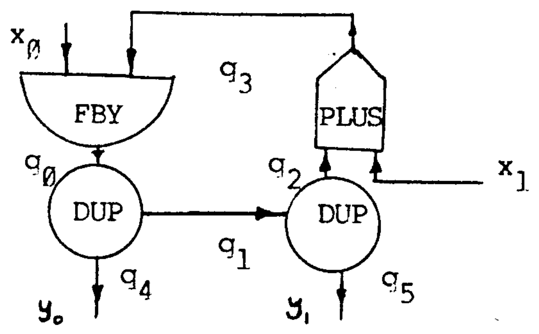
The function $\mathbf{E^C}$ can used to associates an equation with each internal arc in the open net. In a similar way $\mathbf{F^C}$ can be used to associate an equation with each of the output arcs of **c**. For example consider the open net N defined by the following structure:

<<FBY,DUP,DUP,PLUS>, { $q_0,q_1,\ldots,q_5$ }, 2, 2>

The nodes FBY etc. have been formally defined in chapter 2 and for notational convenience, in this example we refer to the nodes by their name. The arcs of N are as follows:

$q_0 = \langle\langle 0,0\rangle,\langle 1,0\rangle\rangle$ $q_1 = \langle\langle 1,1\rangle,\langle 2,0\rangle\rangle$ $q_2 = \langle\langle 2,1\rangle,\langle 3,1\rangle\rangle$

$q_3 = \langle\langle 3,0\rangle,\langle 0,1\rangle\rangle$ $q_4 = \langle\langle 1,0\rangle,\langle 4,0\rangle\rangle$ $q_5 = \langle\langle 2,0\rangle,\langle 4,1\rangle\rangle$



Note that we have labelled the input arcs with a sequence of input variables (the x's) and the output arcs with a sequence of output variables (the y's). For this example the $\mathbf{E}^N$ function is

$$\mathbf{E}^N(c,x) = \{ \langle q_0, \mathrm{fby}(x_0,c_{q_3})\rangle, \langle q_1, \mathrm{dup}_1(c_{q_0})\rangle, \langle q_2, \mathrm{dup}_1(c_{q_1})\rangle, \langle q_3, \mathrm{plus}(x_1,c_{q_2})\rangle \}$$

Note that the fby, dup and plus are not the nodes FBY, DUP and PLUS, they are the functions computed by these nodes. From the above definition of $\mathbf{E}^N$ we see that the set of equations associated with the internal arcs of N are:

$$c_{q_0} = \mathrm{fby}(x_0,c_{q_3})$$
$$c_{q_1} = \mathrm{dup}_1(c_{q_0})$$
$$c_{q_2} = \mathrm{dup}_1(c_{q_1})$$
$$c_{q_3} = \mathrm{plus}(x_1,c_{q_2})$$

For the function $\mathbf{F}^N$ we have the following definition:

$$\mathbf{F}^N(c,x) = \{ \langle \mathrm{Toport}(q_4),\ \mathrm{dup}_0(c_{q_0})\rangle, \langle \mathrm{Toport}(q_5),\ \mathrm{dup}_0(c_{q_1})\rangle \}$$
$$= \langle\ \mathrm{dup}_0(c_{q_0}),\ \mathrm{dup}_1(c_{q_1})\ \rangle$$

From this we see that the equations associated with the output arcs are:

$$y_0 = \mathrm{dup}_0(c_{q_0})$$
$$y_1 = \mathrm{dup}_1(c_{q_1})$$

Thus given any pure dataflow net **c** we can use $\mathbf{E}^{\mathbf{c}}$ and $\mathbf{F}^{\mathbf{c}}$ to formalise precisely what it means for a **c** to compute the least fixed point of its associated set of equations (A3). However, before we give this definition we define what it means for the output activity of a net to compute that predicted by the least fixed point solution to the net's associated set of equations. Nets which behave in this way are called Output Kahnian or more concisely O-Kahnian.

(A2) Definition Let **c'** be a pure dataflow open net

Let $x \in \mathrm{Ka}^{\mathrm{Inportarity}(\mathbf{c})}$

Let $c \in \mathrm{Ka}^{\mathrm{Internalarcs}(\mathbf{c})}$

c' is said to be <u>O-Kahnian</u>

iff

the output activity of the net **c**,

given input x, is

$\mathbf{F}^{\mathbf{c}}(\ \mu c\mathbf{E}^{\mathbf{c}}(c,x),\ x)$

(A3) Definition Let **c** be a pure dataflow open net

Let $x \in Ka^{\text{Inportarity}(\mathbf{c})}$

Let $a \in Ka^{\text{Internalarcs}(\mathbf{c})}$

c is said to be <u>Kahnian</u>

iff

(i) **c** is O-Kahnian

(ii) the activity of the internal arcs of **c**, given input x, is $\mu c\mathbf{E}^{\mathbf{c}}(c,x)$

We now prove that any finite pure dataflow net is O-Kahnian. The proof is by induction on the size of the net and uses the following important lemmas:

(A4) Lemma Let **a**, **b** be O-Kahnian nets

Let **c** be **a:b** (i.e the juxtaposing of **a** and **b**)

c is O-Kahnian

Proof Let $\mathbf{A}$ = Internalarcs(**a**)

$\mathbf{B}$ = Internalarcs(**b**)

$\mathbf{C}$ = Internalarcs(**c**)

n = Inportarity(**a**)

n' = Inportarity(**b**)

m = Outportarity(**a**)

m' = Outportarity(**b**)

$a \in Ka^{\mathbf{A}}$

$b \in Ka^{\mathbf{B}}$

$c \in Ka^{\mathbf{C}}$

$x \in Ka^{n+n'}$

Since **a** and **b** are O-Kahnian we have from

(A2) that the output activity of **a**, given $z \in Ka^n$ is

$\mathbf{F}^{\mathbf{a}}(\mu a\mathbf{E}^{\mathbf{a}}(a,z),z)$.

Similarly the output activity of **b**, given $y \in Ka^m$ is

$\mathbf{F}^{\mathbf{b}}(\mu b\mathbf{E}^{\mathbf{b}}(b,y),y)$.

It is fairly obvious that the juxtaposing of two

nets will not change their operational behaviour

since the nets do not interact.

Thus the output of **c**, given input z y, will certainly be

$\mathbf{F}^{\mathbf{a}}(\mu a\mathbf{E}^{\mathbf{a}}(a,z),z)\hat{}\,\mathbf{F}^{\mathbf{b}}(\mu b\mathbf{E}^{\mathbf{b}}(b,y),y)$ ....(1)

or alternatively we could replace $z\hat{}y$ by x ($\in Ka^{n+n'}$)

with $z = x|n$ and $y = \langle x_n, x_{n+1}, \dots, x_{n+n'-1}\rangle$

If we are to prove that **c** is O-Kahnian then we

must prove that the output activity of **c** given

input x is

$\mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x),x)$ ....(2)

Let us begin be expressing $\mathbf{F}^{\mathbf{c}}$ and $\mathbf{E}^{\mathbf{c}}$ in terms of $\mathbf{F}^{\mathbf{a}}$,

$\mathbf{F}^{\mathbf{b}}$, $\mathbf{E}^{\mathbf{a}}$ and $\mathbf{E}^{\mathbf{b}}$. We can do this because we know

the juxtaposed net relates to the original nets.

However, before we do this we must take into

account the re-naming that is done when the nets are

juxtaposed. To do this we introduce the re-name function:

$rn(D,l) = \{ \langle\langle n+l,i\rangle,\langle m+l,j\rangle\rangle \mid \langle\langle n,i\rangle,\langle m,j\rangle\rangle \in D \}$

Now $\mathbf{F}^{\mathbf{c}}$ can be written as

$\mathbf{F}^{\mathbf{c}}(c,x) = \mathbf{F}^{\mathbf{a}}(c|\mathbf{A},x|n')\hat{}\,\hat{\mathbf{F}}^{\mathbf{b}}(c-c|\mathbf{A},x_{n},\ldots,x_{n+n'-1})$

and $\mathbf{E}^{\mathbf{c}}$ can be written as

$\mathbf{E}^{\mathbf{c}}(c,x) = \mathbf{E}^{\mathbf{a}}(c|\mathbf{A},x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c-c|\mathbf{A},x_{n},..,x_{n+n'-1})$

where

$\hat{\mathbf{F}}^{\mathbf{b}}:Ka^{rn(\mathbf{B},size(\mathbf{a}))} \times Ka^{n} \rightarrow Ka^{rn(\mathbf{B},size(\mathbf{a}))}$

$\hat{\mathbf{E}}^{\mathbf{b}}:Ka^{rn(\mathbf{B},size(\mathbf{a}))} \times Ka^{n} \rightarrow Ka^{m}$

Note: the essential difference between $\hat{\mathbf{F}}^{\mathbf{b}}$, $\hat{\mathbf{E}}^{\mathbf{b}}$ and $\mathbf{F}^{\mathbf{b}}$, $\mathbf{E}^{\mathbf{b}}$ is that the set of equations generated by the first pair differs from the set of equations generated by the second pair only in only in the names of the variables. The reason for this that in the juxtaposed net the arcs the arcs corresponding to **b** have been re-labelled. Thus the least fixed point of both sets of equations is the same.

Now, if we are to prove that **c** is O-Kahnian we must prove that the expression given to use by the definition of O-Kahnity of **c** (2) is equivalent the the expression we obtained from the known operational behaviour (1). In other word we must prove that the following expressions are equivalent:

For the first m outputs of **c** we must prove that

$\mathbf{F}^{\mathbf{a}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x)|\mathbf{A},x|n)$ is equivalent to

$\mathbf{F}^{\mathbf{a}}(\mu a\mathbf{E}^{\mathbf{a}}(a,x|n)$

For the last m' outputs of **c** we must prove that

$\hat{\mathbf{F}}^{\mathbf{b}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x) - \mu c\mathbf{E}^{\mathbf{c}}(c,x)\,|\mathbf{A}, x_n, \ldots, x_{n+n'-1})$

is equivalent to

$\mathbf{F}^{\mathbf{b}}(\mu b\mathbf{E}^{\mathbf{b}}(b, x_n, \ldots, x_{n+n'-1})$

For the first m outputs we only have to prove

that the arguments of $\mathbf{F}^{\mathbf{a}}$ are equivalent:

$\mu c\mathbf{E}^{\mathbf{c}}(c,x)\,|\mathbf{A} = \mu a\mathbf{E}^{\mathbf{a}}(a,x|n)$

lhs $\mu c\ \mathbf{E}^{\mathbf{c}}(c,x)\,|\mathbf{A}$

$\downarrow$ (μ elimination)

$c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{c}}(c,x)$ end

$\downarrow$ (expansion of $\mathbf{E}^{\mathbf{c}}$)

$c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{a}}(c|\mathbf{A}, x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c - c|\mathbf{A}, x_n, \ldots, x_{n+n'-1})$

end

$\downarrow$ (adding new equations)

$c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{a}}(c|\mathbf{A}, x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c - c|\mathbf{A}, x_n, \ldots, x_{n+n'-1})$

$a = c|\mathbf{A}$

end

$\downarrow$ (substitution of expression for variable)

a where $c = \mathbf{E}^{\mathbf{a}}(a, x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c - c|\mathbf{A}, x_n, \ldots, x_{n+n'-1})$

$a = c|\mathbf{A}$

end

$\downarrow$ (replacing a variable by its definition)

a where $c = \mathbf{E}^{\mathbf{a}}(a, x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c - c|\mathbf{A}, x_n, \ldots, x_{n+n'-1})$

$a = [\mathbf{E}^{\mathbf{a}}(a, x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c - c|\mathbf{A}, x_n, \ldots, x_{n+n'-1})]\,|\mathbf{A}$

end

$\downarrow$ (simplification)

a where $c = \mathbf{E}^{\mathbf{a}}(a,x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c-c|\mathbf{A},x_n,\dots, x_{n+n'-1})$

$a = \mathbf{E}^{\mathbf{a}}(a,x|n)$

end

$\downarrow$ (removing unused equation)

a where $a = \mathbf{E}^{\mathbf{a}}(a,x|n)$ end

$\downarrow$ (μ introduction)

$\mu a\ \mathbf{E}^{\mathbf{a}}(a,x)$

This the the rhs

For the last m' outputs of **c** we must prove that:

$\hat{\mathbf{F}}^{\mathbf{b}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x)-\mu c\mathbf{E}^{\mathbf{c}}(c,x)|\mathbf{A},x_n,\dots,x_{n+n'-1})$

is equivalent to

$\mathbf{F}^{\mathbf{b}}(\mu b\mathbf{E}^{\mathbf{b}}(b,x_n,\dots,x_{n+n'-1}),x_n,\dots,x_{n+n'-1})$

Since $\hat{\mathbf{F}}^{\mathbf{b}}$ and $\mathbf{F}^{\mathbf{b}}$ have the same meaning (except for the re-naming of arcs) we must prove that their arguments are equal except for the re-naming of arcs.

Essentially this means proving that

$\mu c\mathbf{E}^{\mathbf{c}}(c,x)-\mu c\mathbf{E}^{\mathbf{c}}(c,x)|\mathbf{A} = \mu b\mathbf{E}^{\mathbf{b}}(b,x_n,\dots,x_{n+n'-1})$

lhs $\mu c\mathbf{E}^{\mathbf{c}}(c,x)-\ \mu c\mathbf{E}^{\mathbf{c}}(c,x)|\mathbf{A}$

$\downarrow$ (μ elimination)

$c-c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{c}}(c,x)$ end

$\downarrow$ (expansion of $\mathbf{E}^{\mathbf{c}}$)

$c-c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{a}}(c|\mathbf{A},x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c-c|\mathbf{A},x_n,\dots,x_{n+n'-1})$

end

$\downarrow$ (add new equations)

$c-c|\mathbf{A}$ where $c = \mathbf{E}^{\mathbf{a}}(c|\mathbf{A},x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(c-c|\mathbf{A},x_n,..,x_{n+n'-1})$

$a = c|\mathbf{A}$

$b = c-c|\mathbf{A}$

end

$\downarrow$ (substitute expression for variable)

b where $c = \mathbf{E}^{\mathbf{a}}(a,x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})$

$a = c|\mathbf{A}$

$b = c-c|\mathbf{A}$

end

$\downarrow$ (remove c)

b where $a = [\mathbf{E}^{\mathbf{a}}(a,x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})]|\mathbf{A}$

$b = [\mathbf{E}^{\mathbf{a}}(a,x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})]-$

$[\mathbf{E}^{\mathbf{a}}(a,x|n) \cup \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})]|\mathbf{A}$

end

$\downarrow$ (simplification)

b where $a = \mathbf{E}^{\mathbf{a}}(a,x|n)$

$b = \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})$

end

$\downarrow$ (removing unused equation)

b where $b = \hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,..,x_{n+n'-1})$ end

$\downarrow$ (μ introduction)

$\mu b\hat{\mathbf{E}}^{\mathbf{b}}(b,x_n,...,x_{n+n'-1})$

This is the rhs except for the re-naming associated with $\hat{\mathbf{E}}^{\mathbf{b}}$.

Thus we have proved that **c** is O-Kahnian.

QED

(A5) Lemma Let **c** be an O-Kahnian pure dataflow open net.

Let $i \in$ Inportarity(**c**)

Let $j \in$ Outportarity(**c**)

Let **c'** be the result of looping back the j^{th} output arc of **c** to feed the i^{th} input arc of **c.**

c' is O-Kahnian

Proof Let **C'** = Internalarcs(**c'**)

Let **C** = Internalarcs(**c**)

Let n = Inportarity(**c**)

Let m = Outportarity(**c**)

Let $x \in Ka^{n}$

Let $x' \in Ka^{n-1}$

Let $c \in Ka^{\mathbf{C}}$

Let $c' \in Ka^{\mathbf{C'}}$

Let q = $\langle$Arcs(**c**)$^{-1}$(Size(**c**),i),Arcs(**c**)(Size(**c**),j)$\rangle$

(i.e the looped arc)

Thus $\mathbf{C'} = \mathbf{C} \cup \{ q \}$

Since **c** is O-Kahnian we know that the output activity of **c** given input x is:

$\mathbf{F}^{\mathbf{c}}(\mu c \mathbf{E}^{\mathbf{c}}(c,x),x)$

We want to prove that the output activity of **c'** given input x' is:

$\mathbf{F}^{\mathbf{c'}}(\mu_{c'} \mathbf{E}^{\mathbf{c'}}(c',x'),x')$

Since we know the relationship between the old net and the iterated net we can express $\mathbf{F}^{\mathbf{c}'}$ and $\mathbf{E}^{\mathbf{c}'}$ in terms of $\mathbf{F}^{\mathbf{c}}$ and $\mathbf{E}^{\mathbf{c}}$:

$\mathbf{F}^{\mathbf{c}'}(c',x') = \mathbf{F}^{\mathbf{c}}(c'/_q, x'\uparrow_i(c'_q))\downarrow_j$

$\mathbf{E}^{\mathbf{c}'}(c',x') = \mathbf{E}^{\mathbf{c}}(c'/_q, x'\uparrow_i(c'_q))/_q\mathbf{F}^{\mathbf{c}}(c'/_q,\ x'\uparrow_i(c'_q))\downarrow_j$

Since $\mathbf{c}$ is O-Kahnian it computes a sequence of functions

$f:(Ka^n \to Ka)^m$

To be more precise it computes

$f = \lambda x\ \mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x),x)$

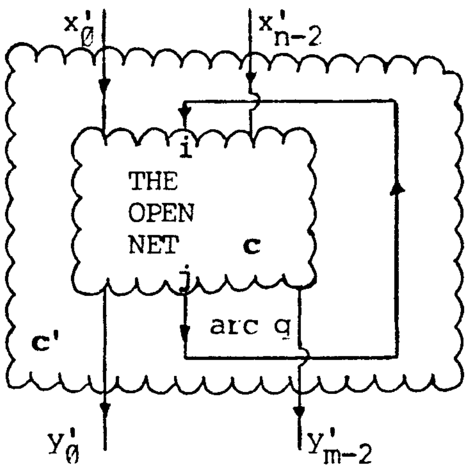
This means we can use infinite games.

Let τ be a totally correct strategy for the game $G(f,\mathbf{c})$.

We prove that there is a totally correct strategy τ' for $G(f',\mathbf{c}')$ such that

$f' = \langle \lambda x'\ f_k(x'_0, \ ,x_{i-1},\mu z f_{m-1}(x'\uparrow_i z),x'_i, \ ,x'_{n-2}) \rangle_{k \in m\downarrow_j}$

$\mathbf{c}'$ is the open net with n-1 input and m-1 output arcs formed by bending back the j th output arc of $\mathbf{c}$ to feed the i th input arc of $\mathbf{c}$.



τ' is a totally correct strategy derived from τ using an auxiliary game in which τ is applied to

x' and the output from the jth output arc. Note that in infinite games is a standard technique to have a strategy play against itself.

Since τ' is derived from τ, the first output of the each of the output arcs of **c'** is

$\forall k \in m \; m \neq j$

$f_k(x'_0,..,x'_{i-1},\triangle,x'_i,..,x'_{n-2})|1$

(since **c'** has no input on its ith input arc).

The second output will be

$f_k(x'_0,..,x'_{i-1},\mathbf{o}_1,x'_i,..,x_{n-2})|2$

where $\mathbf{o}_1 = f_j(x'_0,..,x'_{i-1},\triangle,x'_i,..,x'_{n-2})|1$

(since τ is playing against itself).

If we continue the process we get the following that the rth of **c'** is:

$f_k(x'_0,..,x'_{i-1},\mathbf{o}_{r-1},x'_i,..,x'_{n-2})|r$

where $\mathbf{o}_{r-1} = f_j(x'_0,..,x'_{i-1},\mathbf{o}_{r-2},x'_i,..,x'_{n-2})|r-1$

Thus **c'** certainly does compute

$f' = \lambda x'\; f(x'\uparrow_i \mu z(f(x'\uparrow_i z))_j)\downarrow_j$

Thus the infinite game argument defines for us the operational behaviour of the output arcs of **c'**.

If we want to prove that **c'** is O-Kahnian then we must prove that, given input x', the output activity of **c'** is:

$\mathbf{F}^{\mathbf{c}'}(\mu c'\mathbf{E}^{\mathbf{c}'}(c',x'),x') \qquad (= f'(x'))$

Since we can express $\mathbf{F}^{\mathbf{c}}$ in terms of $\mathbf{F}^{\mathbf{c}}$ and $\mathbf{E}^{\mathbf{c}}$ we have:

$f'(x') = \mathbf{F}^{\mathbf{c}}((\mu c'\mathbf{E}^{\mathbf{c}'}(c',x'))/_q,\; x'\uparrow_j(\mu c'\mathbf{E}^{\mathbf{c}'}(c',x'))_b)\downarrow_j \qquad .....(2)$

Thus we must prove that the expression we have derived from the known operational behaviour (2) is equivalent to the expression we have from the definition of O-Kahnity for **c'** i.e. (1). As we are trying to show the equivalence between two expression each of which is the result of an application of $\mathbf{F}^{\mathbf{C}}$, then all we need do is to show that each expression has equivalent arguments for $\mathbf{F}^{\mathbf{C}}$.

The first argument of $\mathbf{F}^{\mathbf{C}}$ as given by the expanded definition of the O-Kahnity of **c'** is:

$(\mu c'\mathbf{E}^{\mathbf{C'}}(c',x'))_b$

The first argument of $\mathbf{F}^{\mathbf{C}}$ as given by the infinite game solution is:

$\mu z(\mathbf{F}^{\mathbf{C}}(\mu c\mathbf{E}^{\mathbf{C}}(c,x'\uparrow_i z),x'\uparrow_i z))_j$

$(\ \mu c'\mathbf{E}^{\mathbf{C'}}(c',x')\ )_b$

$\downarrow$ (μ elimination)

c'_b where $c' = \mathbf{E}^{\mathbf{C'}}(c',x')$

$\downarrow$ (expansion of $\mathbf{E}^{\mathbf{C'}}$)

c'_b where $c' = \mathbf{E}^{\mathbf{C}}(c'/_b,x'\uparrow_i(a'_b))/_b\mathbf{F}^{\mathbf{C}}(c'/_b,x'\uparrow_i(c'_b))_j$

end

$\downarrow$ (adding new equations)

c'_b where $c' = \mathbf{E}^{\mathbf{C}}(c'/_b,x'\uparrow_i(a'_b))/_b\mathbf{F}^{\mathbf{C}}(c'/_b,x'\uparrow_i(c'_b))_j$

$c = c'/_b$

$z = c'_b$

end

$\downarrow$ (substitution of an expression for a variable)

z where $c' = \mathbf{E}^{\mathbf{C}}(c,x'\uparrow_i z)/_b\mathbf{F}^{\mathbf{C}}(c,x'\uparrow_i z)_j$

$c = c'/_b$

$z = c'_b$

end

$\downarrow$ (eliminate c')

z where $c = [\mathbf{E^C}(c,x'\uparrow_i z)/_b \mathbf{F^C}(c,x'\uparrow_i z)_j]/_b$

$z = [\mathbf{E^C}(c,x'\uparrow_i z)/_b \mathbf{F^C}(c,x'\uparrow_i z)_j]_b$

end

$\downarrow$ (simplification)

z where $c = \mathbf{E^C}(c,x'\uparrow_i z)$

$z = \mathbf{F^C}(c,x'\uparrow_i z)_j$

end

$\downarrow$ (taking a fixed point)

z where $c = \mu c\mathbf{E^C}(\mathbf{c},x'\uparrow_i z)$

$z = \mathbf{F^C}(c,x'\uparrow_i z)_j$

end

$\downarrow$ (replace a variable by its definition)

z where $z = \mathbf{F^C}(\mu c\mathbf{E^C}(c,x'\uparrow_i z)_j$ end

$\downarrow$ (μ introduction)

$\mu z\ \mathbf{F^C}(\mu c\mathbf{E^C}(c,x'\uparrow_i z),x'\uparrow_i z)_j$

This is the output activity predicted by the infinite game i.e. the rhs.

The second argument of (1) in its expanded form is:

$(\ \mu c'\mathbf{E^{C'}}(c',x')\ )/_b$

The second argument of (2) is:

$\mu c\mathbf{E^C}(c,x'\uparrow_i \mu z(\mathbf{F^C}(\mu c\mathbf{E^C}(c,x'\uparrow_i z),x'\uparrow_i z))_j)$

$(\ \mu c'\mathbf{E^{C'}}(c',x')\ )/_b$

$\downarrow$ (μ elimination)

$c'/_b$ where $c' = \mathbf{E^{C'}}(c',x')$

$\downarrow$ (expansion of $\mathbf{E^{C'}}$)

```
$c'/_b$ where $c' = \mathbf{E}^{\mathbf{c}}(c'/_b, x'\uparrow_i(a'_b))/_b\mathbf{F}^{\mathbf{c}}(c'/_b, x'\uparrow_i(c'_b))_j$
        end

                $\downarrow$    (adding new equations)

$c'/_b$ where $c' = \mathbf{E}^{\mathbf{c}}(c'/_b, x'\uparrow_i(a'_b))/_b\mathbf{F}^{\mathbf{c}}(c'/_b, x'\uparrow_i(c'_b))_j$
                $c = c'/_b$
                $z = c'_b$
        end

                $\downarrow$    (substitution of an expression for a variable)

    $c$   where $c' = \mathbf{E}^{\mathbf{c}}(c, x'\uparrow_i z)/_b\mathbf{F}^{\mathbf{c}}(c, x'\uparrow_i z)_j$
                $c = c'/_b$
                $z = c'_b$
        end

                $\downarrow$    (eliminate c')

    $c$   where $c = [\mathbf{E}^{\mathbf{c}}(c, x'\uparrow_i z)/_b\mathbf{F}^{\mathbf{c}}(c, x'\uparrow_i z)_j]/_b$
                $z = [\mathbf{E}^{\mathbf{c}}(c, x'\uparrow_i z)/_b\mathbf{F}^{\mathbf{c}}(c, x'\uparrow_i z)_j]_b$
        end

                $\downarrow$    (simplification)

    $c$   where $c = \mathbf{E}^{\mathbf{c}}(c, x'\uparrow_i z)$
                $z = \mathbf{F}^{\mathbf{c}}(c, x'\uparrow_i z)_j$
        end

                $\downarrow$    (taking a fixed point)

    $c$   where $c = \mu c\mathbf{E}^{\mathbf{c}}(c, x'\uparrow_i z)$
                $z = \mathbf{F}^{\mathbf{c}}(c, x'\uparrow_i z)_j$
        end

                $\downarrow$    (replace a variable by its definition)
```

c where $c = \mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z)$

$z = \mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z),x'\uparrow_i z)_j$

end

$\downarrow$ (taking a fixed point)

c where $c = \mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z)$

$z = \mu z\mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z),x'\uparrow_i z)_j$

end

$\downarrow$ (replacing a variable by its definition)

c where $c = \mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i \mu z\mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z)_j)$

end

$\downarrow$ (μ introduction)

$$\mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i \mu z\mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x'\uparrow_i z),x'\uparrow_i z)_j$$

This is the output activity predicted by the infinite game

Thus we have that **c** is O-Kahnian.

QED

(A7) Theorem All finite pure dataflow open nets are O-Kahnian

Proof By induction on the size of the open net

(i.e. the number of nodes within a net)

base step ($n = 1$) all open nets containing

a single node are O-Kahnian

(all nodes compute history functions)

assume all nets of size $n-1$ are O-Kahnian

i) adding an extra node using juxtaposition

is O-Kahnian (see A3).

ii) applying an arbitrary number of iterations to any net arising from i) preserves O-Kahnianity (see A4).

Hence all pure dataflow open nets are O-Kahnian.

QED

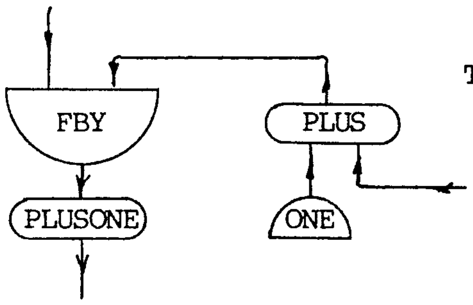
(A8) Definition Let **c** be a open net.

$\hat{\mathbf{c}}$ is the output net of **c**

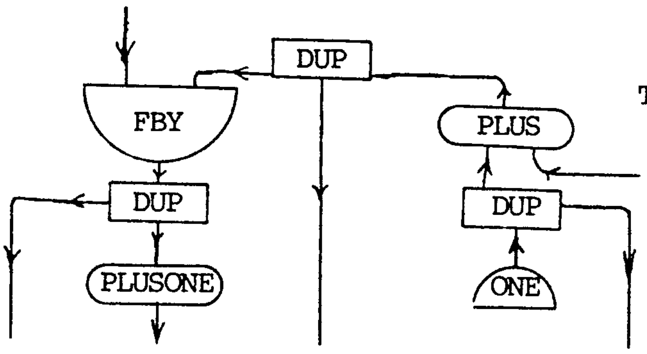
iff

$\hat{\mathbf{c}}$ is an open net derived from **c** such that every internal arc of **c** is an output arc of $\hat{\mathbf{c}}$ (i.e. each internal arc is cut and a duplicator node is used to re-connect the severed arc, the second output of the duplicator becomes an output arc of $\hat{\mathbf{c}}$).

The following diagram shows an open net **c** and its corresponding $\hat{\mathbf{c}}$.



The net **c**



The net $\hat{\mathbf{c}}$

(A9) Theorem Let **c** be any pure dataflow open net.

c is Kahnian iff $\hat{\mathbf{c}}$ is O-Kahnian

Proof **c** Kahnian = > $\hat{\mathbf{c}}$ is O-Kahnian

If **c** is Kahnian then tapping off all internal arcs (using duplicator nodes) will not change the operational behaviour of that part of the net $\hat{\mathbf{c}}$ that corresponds to the original net. The only effect operationally is to duplicate arc activity. Denotationally the net $\hat{\mathbf{c}}$ is associated with a set of equations that differs from the set of equations for **c**, by the addition of extra equations, the rhs of which involve a duplicate function. It would not be difficult using the techniques we used to prove (A7) to show that $\hat{\mathbf{c}}$ is O-Kahnian.

$\hat{\mathbf{c}}$ is O-Kahnian = > **c** is Kahnian

The set of equations associated with **c** differs from the set of equations for $\hat{\mathbf{c}}$ only by the addition of extra equations involving duplicator functions. Removing these extra equations obviously corresponds to the removal of the duplicator nodes used to tap off the internal arcs of **c**. This will have no effect on the operational behaviour of the net **c**. Again using the techniques we used to prove (A7) it is not difficult to prove that **c** is Kahnian.

QED

(A10) Theorem (The Kahn Principle)

All finite pure dataflow open nets are Kahnian.

Proof From (A7) we have that every pure dataflow open net is O-Kahnian.

Given any finite pure dataflow open net **c**, the corresponding output net **c** is O-Kahnian. However, theorem A9 tells us that if **c** is O-Kahnian then **c** is Kahnian. Thus all pure dataflow open nets are Kahnian.

QED

A proof of the Kahn principle for certain kinds of infinite nets is given in Chapter 5.

C. Some Consequences of the Kahn Principle

In this section we examine two interesting consequences of the Kahn principle. The first is that pure dataflow programs are not graphs but rather sets of equations. We can think of these equations as an equational dataflow language. We believe the programs in this language are concise and elegant. The second consequence is that we can reason about operational activity denotationally. An interesting example of this is the cycle sum test for dataflow deadlock.

An equational dataflow language

Great amounts of resources have been poured into research projects all over the world with the sole aim of developing a dataflow language. Jack Dennis at MIT is developing a language called VAL[1]; Arvind and Gostelow a language called ID[51] and Osman, Hankin and Sharp a language called CAJOLE[24].

On the other hand we get an elegant equational dataflow programming language for "free" (i.e. via the Kahn principle). Here are some examples of some simple "standard" programs:

(i) The fibonacci program

```
F = 1 fby ( 2 fby (F + 2))
G = next F
OUTPUT = F
```

(ii) A program to generate the stream of factorials

```
I = 1 fby (I+1)
F = 1 fby (F*(next I))
OUTPUT = F
```

Note: fby appears as an infix operator.

The equational programmer uses circularity to bring about repetition. This idea is far more general than the simple iterations of our two simple programs. (Advanced equational programming is beyond the scope of this thesis and the reader is referred to Ashcroft and Wadge[7]). The equational programs we have written so far are extremely simple; but by adding one or two more powerful operators the language can become quite powerful. An example of one of these advanced operators is called "upon". The following is an operational definition of "upon": As long as O's (representing falses) arrive down its second input arc it sends copies of the last daton it consumed from its first input arc. When a 1 (representing true) arrives on its first input arc, the node consumes the next daton on its first input arc and send copies until the next 1 arrives. For example if the history 4,2,3,9,8,... arrives along the first input arc and O,1,O,1,1,1,O,... arrives along the second input arc the the history of the output arc is 4,4,2,2,3,9,8,8,...

We can now use this node (or rather the history function computed by the node) in the following merge program:

```
AA = A upon AA ≤ BB
BB = B upon BB ≤ AA
 C = if AA ≤ BB then AA else BB fi
OUTPUT = C
```

If A and B are sequences in increasing order then the output will be the ordered merge. If we add to the above program the following equations:

```
D = 1 fby C
A = 2*D
B = 3*D
OUTPUT = D
```

We have a new program, without inputs, that produces as output all numbers of the form 2^i, 3^j in increasing order. Thus even with very few operators we can write interesting programs.

Deadlock and the cycle sum test

In all but trivial dataflow programs we find that variables are defined directly or indirectly in terms of themselves. The fact that a variable is defined in terms of itself means that its corresponding arc in the network is part of some loop. Thus the datons that travel around that loop are endlessly re-cycled. However, this may not always be the case because it is possible that the loop runs dry. The following two programs illustrate the situation:

(i) x = x + 1

The loop is permanently dry.

(ii) x = 1 fby (next (1 fby(next x)))

The loop is able to produce the output 1 before seizing up.

More complex deadlocks may occur which depend on the value of certain variables. For example:

```
x = 9 fby next y
y = 3* x upon p
```

This program dealocks almost immediately unless the first value of P is 0.

Thus we have the situation where certain programs like the one above deadlock and yet others like fibonacci and merge go on forever (given enough inputs).

In [46] Wadge describes how it is possible to decide on what programs will deadlock. He notices that in healthy programs a variable depends on itself in such a way that the present value of the variable depends on at most the previously computed value. On the other hand in programs which deadlock a variable requires a present or future value of itself. This observation suggests some sort of requirement which ensures that the present value of a variable is dependent only on its previous values. To make this idea more precise we need to state in an exact way the ways in which the outputs of various nodes depend on their inputs. For example:

(i) A = B+C

The value of A depends on the present value of B and C.

(ii) A = next B

A depends on the value of B one time step in the future.
i.e. the first 3 values of A require the first 4 values of B.

(iii) A = C fby D

A depends on the present value of C and on the value of D one time step in the past.
i.e. The first 3 values of A depend on the first 3 values of C and the first two values of D.

These dependencies are clearly cumulative and thus given

A = 3 fby (5 fby B)

The first 3 values of A require only the first value of B.

Effects may also cancel each other, thus

A = 9 fby next B

The first n values of A depend on the first n values of B.

This observation lead Wadge to assign the following numbers to the input arcs of the various operators:

(i) O is assigned to each of the data operators such as +,*,..

(ii) O and -1 respectively are assigned to the arguments of fby

(iii) +1 is assigned to the argument of next.

(iv) O and -1 respectively are assigned to the arguments of upon.

When operations are composed their numbers are added; to find the way in which the value of a whole expression can depend on the values of variables occuring in it, we consider the expression as a tree, trace paths from the root of the tree to the variable; and add up the number associated with the operators on the path. For example if the expression is

(9 fby(next B + next C)) upon (next (P fby B))

the path to P goes through the second argument of upon(-1), the argument of next (+1) and the first argument of fby (Ø). The sum of this is O and so we may conclude that in general the present value of the expression could depend on the present value of P.

Given the graph of a program we can tell wether a variable depends on its own present or future values. What we do is to form the path sums of all paths which start at the arc which corresponds to the variable in question. (i.e. all cycles containing the arc). If the cycle sum is negative,the dependency of the variable on itself is healthy. To ensure the whole program is healthy we repeat the cycle sum test for each variable. Equivalently we make sure that every cycle in the graph has a negative cycle sum.

The importance of Wadge's paper is that although the concept of deadlock is operational he goes on (via the Kahn principle) to give a denotational proof of the cycle sum test. Such a proof is beyond the scope of this thesis.

Note: In this thesis we have numbered the time dependencies in the opposite way to which Wadge numbered them in his paper. The reason for this is that it is compatible with the extended version of the cycle sum test (see Faustini and Wadge[55]). The extended cycle sum test allows the cycle sum test to be applied to equational programs which may include recursive defined user functions.

Chapter 5

Possible extensions and refinements

In this chapter we briefly examine ways in which to extend the denotational semantics to handle a broader class of behaviours (i.e. not just pure dataflow). In addition we look at ways describing functional nodes in terms of their internal properties; we see this as a refinement to the operational semantics. However, before we look at these extensions and refinements we prove the Kahn principle for certain kinds of infinite net.

A. A Functional Programming Approach to Extend Pure Dataflow

The Kahn principle as we stated it in chapter 4 was defined only for finite nets. The next question we ask ourselves is does the principle hold for infinite pure dataflow nets? As far as our operational model is concerned there is no problem in defining infinite nets, but this in itself is no justification for proving the extended Kahn principle. However, if we look at the denotational semantics and in particular the related equational dataflow programming language, we immediately find an excellent reason for wanting to prove the extended Kahn principle. The equational dataflow programming language described in (∫4.B) is limited in that the programs can only be defined in terms of a simple finite set of equations. To allow the user to develop programs in a structured way we need to extend this 'simple' language by allowing equations defining functions, including recursive definitions. Some typical user defined functions (UDF's) are :

```
upon(x,p) = first x fby if p then upon(next x ,next p)
                                  else upon(x, next p) fi
whenever(x,p) = if first p then first x fby whenever(next x, next p)
                                else whenever(next x, next p) fi
merge(a,b) = valof
                aa = upon(a, aa ≤ bb )
                bb = upon(b, bb ≤ aa )
                result = if aa ≤ bb then aa else bb fi
               end
```

The implementation of this extended language (which is similar to Lucid[6]) involves either dynamically growing nets or (notionally) infinite nets (but still pure datflow). The methods of this thesis extend naturally to such nets and permit us to give, for the first time, a proof of the correspondingly extended Kahn principle. Notice that even though the nets are required to be infinite they always have a finite number of input and output arcs. Nets with a finite number of input arcs and a finite number of output arcs are said to have finite fan/out an fan/in. The way we prove the extended Kahn principle is to build the infinite net as the limit of a sequence of finite nets. We begin by taking a single node, for convenience we choose the first node in the sequence of nodes associated with the infinite net's formal description. This node together with its associated input/output arcs (which may include iterated arcs) is a finite pure dataflow net. Since the pure dataflow net is finite we know that the net is Kahnian. In addition we know how the zeroth node relates to the infinite net. The reason this is important is that we will place no input on those input arcs that are associated with internal arcs in the infinite net. Thus the behaviour of the single node together with its input/output arcs is an approximation to the behaviour of the infinite net. Next we take the first and second node in the infinite nets node sequence. This time there are two nodes which may themselves be interconnected (the same way as they are interconnected in the infinite net). Again we choose not to give the second net any input on the inputs that correspond to the internal arcs of the infinite net. Thus the second net is an even closer approximation to the behaviour of the infinite net. If we continue this argument ad infinitum we get a seqeunce of net behaviours the limit of the sequence of behaviours being the behaviour of the infinite net.

(A1) Theorem (the extended Kahn Principle)

All infinite pure dataflow nets with finite fan/in and fan/out are Kahnian.

Proof Let $\mathbf{c}$ be an infinite pure dataflow net

with Inportarity($\mathbf{c}$) $\in \omega$

Outportarity($\mathbf{c}$) $\in \omega$

Let $\mathbf{C}$ = Internalarcs($\mathbf{c}$)

Let $c \in Ka^{\mathbf{C}}$

$\forall i \in \omega$

Let $\mathbf{c}_i$ be a open net with finite fan/in and fan/out.

Let $a_i \in Ka^{\mathbf{c}(i)}$

Let $y_i \in Ka^{\text{Inportarity}(\mathbf{c}(i))}$

Let $x \in Ka^{\text{Inportarity}(\mathbf{c})}$

If we want to prove that **c** is Kahnian then we must prove that:

$\mu c\mathbf{E}^{\mathbf{c}}(c,x)$ is the activity of the internal arcs of **c** and that

$\mathbf{F}^{\mathbf{c}}(\mu c\mathbf{E}^{\mathbf{c}}(c,x),x)$ is the activity of the output arcs of **c**.

Let $\mathbf{c}_0$ be the open net formed from node zero of Nodes(**c**) together with the nodes input and output arcs (including iterated arcs) Since $\mathbf{c}_0$ is finite it is Kahnian and thus given inputs $y_0 \in Ka^{\text{Inportarity}(\mathbf{c}(0))}$ its internal arcs compute $\mu a_0\mathbf{E}^{\mathbf{c}(0)}(a_0,y_0)$.

If we arrange for all the input arcs of **c**(Ø) which correspond to internal arcs in **c** to be starved of input then the internal behaviour of **c**(0) is an approximation to the behaviour of the infinite net. Next we take the **c**(1) to be the open net corresponding to the first two nodes in nodes(**c**). Since this net if finite it too is Kahnian and thus given input $y_1 \in Ka^{\text{Inportarity}(\mathbf{c}(1))}$ the internal behaviour of **c**(1) is given by $\mu a_1\mathbf{E}^{\mathbf{c}(1)}(a_1,y_1)$.

If we again require that the input arcs of **c**(1) which coresspond to internal arcs in **c** are again starved of input then we get an even better approximation to the behaviour of the infinite net.

If we continue this process ad infinitum we generate two sequences. The first sequence is

$$\langle \mathbf{c}(i) \mid i \in \omega \rangle$$

The second is

$$\langle \mu a_i \mathbf{E}^{\mathbf{c}(i)}(a_i, y_i) \mid i \in \omega \rangle$$

Obviously the net **c** is the limit of the sequence of finite nets:

$$\lim_{i \in \omega} \langle \mathbf{c}(i) \mid i \in \omega \rangle = \mathbf{c}$$

In a similar way the behaviour of **c** is the limit of the sequence of finite approximations to the behaviour of **c**:

$$\lim_{i \in \omega} \langle \mu a_i \mathbf{E}^{\mathbf{c}(i)}(a_i, y_i) \mid i \in \omega \rangle = \mu c \mathbf{E}^{\mathbf{c}}(c, x)$$

Thus the internal behaviour of **c** is Kahnian.

We can apply a similar argument to prove that the behaviour of the output arcs of **c** is Kahnian

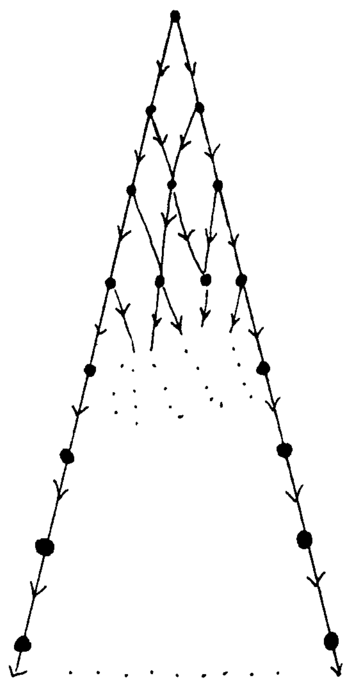
QED

B. Ensuring functionality of nodes

The definition of open net functionality we developed in chapter 3 defines functionality in terms of infinite objects, namely ω-sequences of open net transitions. Thus if we think of an open net as a black box, functionality defines what the black box computes without regard to the black boxes internal behaviour. In this section we examine the internal properties of nodes (i.e. properties of their transition relations) in search of properties that guarantee node functionality.

The one step Church-Rosser like property

Rather than begin looking in detail at the internal behaviour of a node we shall examine the behaviour of an encapsulated net. Since our operational model is truly modular, any property associated with the internal behaviour of an open net, is also associated with the internal behaviour of the corresponding node. We can think of a net computation sequence in terms of a sequence of compatible net transitions. Each net transition includes one transition for each node in the net. Alternatively we can think of net computation in terms of a uni-directed graph. Each node in the graph corresponds to net state and each arc to a net transition. In addition one of the nodes in the graph is distinguished as the initial state. In the diagram below the infinite sequence of arcs on the extreme left corresponds to an infinite sequence of net transitions each of which is a busy wait for each node. Other paths through the graph represent other possible ways of choosing individual node transitions.



From chapter 4 we know that if the nodes in the graph are all functional then any path through the graph will lead to the network computing an approximation to the least fixed point solution to the nets associated set of equations. In particular if a totally correct strategy is used to pick the transitions then exactly the least fixed point solution we be computed. Thus our net computations have a Church-Rosser property. This means that we can set out on any two distinct paths and after travelling along both paths for an arbitrary length of time we would still be able to choose some path that would cause both paths to meet up again and have the same overall behaviour. Since our nets have the encapsulation property then what we have just said about nets applies in a coded form to nodes. In particular we can think of node computation in terms of a uni-directed graph. This time the nodes of the graph denote internal states of a node and the arcs node transitions. One particularly simple way of guaranteeing that a node is functional is to require it to have a

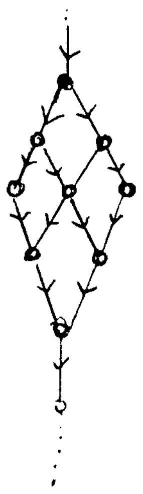
Church-Rosser like property. In other words given a choice of paths through the node computation graph it is always possible at some later stage having taken two distinct paths to join them and have the same overall input/output effect. Nodes with this property are said to be functionally safe. On the other hand a functionally unsafe node is one which according to definition ∫3.E4 is functional but from which it may not be possible to recover from error. An error would be to choose a path which only ever allowed an initial segment of the expected output to be produced.

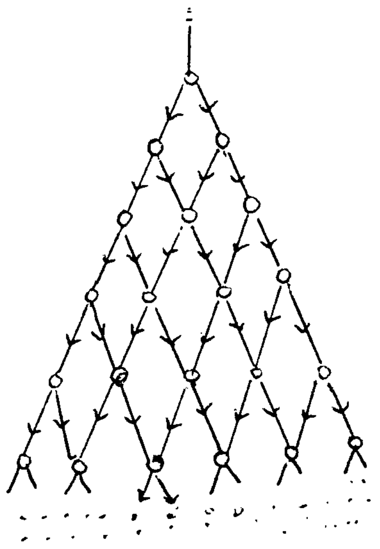
Requiring our nodes to have the Church-Rosser like property is a little too much because it is impossible from looking at a node's transitions to say in general whether the node has this property. We therefore choose the more useful 1-step Church Rosser like property. This property guarantees that when it is possible to go from one internal state to two others in one step, then it is possible for these two paths to meet after one further single step and for the paths to have the same overall

$$\begin{tikzcd}
 & \text{node state} \arrow[dl, "\substack{r1 \\ w1}"'] \arrow[dr, "\substack{r1' \\ w1'}"] & \\
\text{node state} \arrow[dr, "\substack{r2 \\ w2}"'] & & \text{node state} \arrow[dl, "\substack{r2' \\ w2'}"] \\
 & \text{node state} &
\end{tikzcd}$$

$r1 \cup r2 = r1' \cup r2'$ $\qquad\qquad$ $w1 \cup w2 = w1' \cup w2'$

reading and writing effect. This means that nodes with this property are associated with node computation graphs of the following form:





Although we do not give a proof it is not difficult to prove that that nodes with the one step Church-Rosser like property are functional.

Monotone relations and functional nodes

The ideas in this section are in their infancy and should not be thought of as proved results. In fact this section contains the seeds for possible future research. We begin by making a small study of some time dependent nodes to see if we can gain some understanding of time dependency. Let us begin with an example of a node that behaves in a similar way to the identity node except under certain conditions where it produces spurious O's. The formal definition of this node is as follows:

$< \{ \ q \ \}, q, 1, 1, T >$

where T is

$\forall x \in \omega$

$<nil,q> \to <nil,nil,0>$

$<x \ ,q> \to < tt,nil,x>$

So long as this node is supplied with datons at a faster rate than that at which it consumes them it behaves exactly like the identity function. However as soon as we fail to send it datons at a fast enough rate we find the empty buffer computation comes into use and produces spurious 0's.

Another example of a node that computes on empty buffers is the node thst computes the history function "first":

$< \{ \ q_x \mid x \in \omega \ \} \cup \{ \ q \ \} , q , 1, 1, T >$

where $\forall x,y \in \omega$

$<nil,q > \to <nil,nil,nil>$

$<x \ ,q > \to <tt,q_x, \ x>$

$<nil,q_x> \to <nil,nil, \ x>$

$<y, \ q_x> \to <tt, nil, \ x>$

This node computes on empty buffers and yet is still functional. Closer examination of the transitions associated with each node suggests that nodes which compute on empty buffers must be able to perform the same activity no matter what is in the input buffer. We introduce the following definition as a means of capturing the above suggestion.

(B1) Definition Let $\langle D, \leq_D\rangle$ and $\langle E, \leq_E\rangle$ be posets

A relation R over $D \times E$ is said to be monotone

iff

$\forall d^1, d^2 \in D\ e^1 \in E$

$d^1 \leq_D d^2$ and $d^1\ R\ e^1 \rightarrow$

$\exists e^2,\ e^1 \leq_E e^2$ and $d^2\ R\ e^2$

We must now apply this definition to a node transitions relation. Given a node with n input arcs and m output arcs the node's transition relation $T \subseteq \langle\langle B^n \times Q\rangle, \langle E^n \times Q \times B^m\rangle\rangle$ is said to be monotone

iff

$\forall \langle b,q\rangle, \langle b',p\rangle \in T_\emptyset\ \langle e,r,g\rangle \in T_1$

$\langle b,q\rangle \leq_C \langle b',p\rangle$ and $\langle\langle b,q\rangle,\langle e,r,g\rangle\rangle \in T \rightarrow$

$\exists \langle e',r',g'\rangle \in T_1\ \ \langle e,r,g\rangle \leq_E \langle e',r',g'\rangle$ and $\langle\langle b',p\rangle,\langle e',r',g'\rangle\rangle$

where $\langle b,q\rangle \leq_C \langle b',p\rangle$ iff $\forall i \in n\ b_i \subseteq b'_i$ and $q = p$

$\langle e,r,g\rangle \leq_E \langle e',r',g'\rangle$ iff $\forall i \in n\ j \in m$

$e_i \subseteq e'_i$ and $r \subseteq r'$

and $g_j \subseteq g'_j$.

where $x \subseteq y$ iff $x = y$ or $x = nil$

If we examine our two simple examples we will see that "first" node is monotone and the "would be" identity node is not monotone. The reason why this node is non-monotone is that two compatible causes have uncompatible effects:

$\langle\langle nil,q\rangle,\langle nil,nil,0\rangle\rangle$ and $\langle\langle x,q\rangle,\langle tt,nil,x\rangle\rangle$

$\langle nil,q\rangle \leq_C \langle x,q\rangle$ and yet $\langle nil,nil,0\rangle \not\leq_C \langle tt,nil,x\rangle$.

What we would like prove now is that the nodes which have monotone transition relations are functional. Unfortunately this is not the case since the following node has a monotone transition relation and yet is non-determinate:

```
  <{ q,q' },q,2,1,T>
          where T is given by the set
```

$\forall x,y \in \omega$

```
           <nil,nil,q> -> <nil,nil,  q',nil>
           <nil,  y,q> -> <nil, tt,  q',nil>
           <x  ,nil,q> -> <tt ,nil,  q',  y>
           <x  ,  y,q> -> <tt , tt,  q',  y>


           <nil,nil,q'> -> <nil,nil,q,  1>
           <nil,  y,q'> -> <nil, tt,q,  1>
           <x  ,nil,q'> -> <tt ,nil,q,  1>
           <x  ,  y,q'> -> <tt , tt,q,  1>
```

The node just described cleverly disguises its activity between states; and so is able to be monotonic and yet non-determinate. The reason that monotonicity of the transition relation failed to guarantee functionality is that the montone relation fails to detect situations in which the computations associated with two distinct partial causes overlap in activity. A partial cause is one in which one or more of the input buffers is empty. Allowing such overlaps leads inevitable to non-determinate input/output behaviour. Another possible approach to finding a property the guarantees functionality uses the idea of cause chains. A cause chain for the above node is

$<nil,nil,q> \leq_C <nil,y,q> \leq_C <x,y,q>$.

We claim (althought it requires rigorous proof) that if the computing activity associated with each element in a cause chain is non-overlapping and if the total activity for the partial objects in all compatible chains is equal to the activity of the of the complete object which is the limit of these chains then the node is guaranteed to be functional. In the last example above we have a chain of partial causes <<nil,nil,q'>,<nil,y,q'>> and the chain of partial causes <<nil,nil,q'>,<x,nil,q'>> the limit of these chains, within the context of the state q', is <x,y,q'>. The total activity of the chains of partial causes is to erase x and y and to output three 1's. On the other hand the activity associated with the limit cause is to erase x and y and output one 1. Thus there is overlap in the activity of the partial causes and so the node is non-functional. Again we repeat that these ideas are really a topic for further research.

Extending Kahn's denotational semantics

In this section we briefly examine possible extensions to Kahn's denotational model. This is again another area for further research, our aim in this section is to suggest possible extensions and to provide a few references to material that may be of use.

The table below describes the current state (in terms of this thesis) of operational and denotational models.

Opertational Models	Denotational Models		
	Arcs	Nodes	Nets
Pipeline dataflow	?	?	?
F-Nets	?	?	?
T-Nets	?	?	?
Pure Dataflow	Histories	History Functions	Least Fixed Point of a net's associated set of equations

The above table really describes a hierarchy of models, the most general model being pipeline dataflow and the least general pure dataflow. Denotationally we know what the objects are that correspond to arcs, nodes and nets of a pure dataflow net. We now look at the next level up in the hierarchy, namely the T-nets. The T-nets are those pipeline dataflow nets which include all pure datalow nodes with the addition of nodes that are time sensitive. An example of a time senstive node is the node that repeatedly copies its inputs to its outputs but whenever there is no input is outputs a 0. Formally we have

< { q }, q 1, 1, T>

where T is

$\forall x \in \omega$

<nil,q> -> <nil,nil,0>

< x,q> -> < tt,nil,x>

We can see that the zero is included each time there is a break in the supply of inputs. If we assume that the node never receives a daton representing 0 as input, then the occurence of 0's in the output tells us the rate at which the node received its inputs. Now instead of relying on the node never receiving inputs, we introduce a special object called a <u>hiaton</u> which we think of as a unit of delay which (notionally) travels along with the other datons so that a node can produce something regularly even if it has no real output. Using this simple idea we can think of the time dependent nodes such as the example above as a function over hiatonised histories. In the pure dataflow model an arc was associated with a pure history and so denotational we could not reason about the relative rate of arrival on different input arcs. For example the history <1,2,3,4,...> tells us nothing about the way in which the history was formed. On the other hand the sequence <1,*,2,*,3,*,..> tells us the relative time of arrival of the inputs (the * denoting a delay). The example node above does not compute a history function, however, it does compute a function over hiatonised histories:

i:HKA -> HKA where HKA = $Sq(\omega \cup \{ * \}) \cup (\omega \cup \{ * \})^{\omega}$

There is a lot more that could be said about T-nets and hiatonics but this is beyond the scope of this thesis. Incidently the "Brock-Ackerman anomaly" [13] can easily be explained using hiatonics.

An interesting example of the use of hiatonics is to be found in the Ph.D thesis of P.J. Cameron[14]. Camerons thesis is about the design of a non-procedural operating systems language. Hiatonics are used to deal with the time dependent behaviour associated with the scheduling of a resources such as a set of line printers. Although Cameron makes use of hiatonics the user of his language never realises that hiatons are used, they are internal to his system.

Others who have worked with hiatonic are Boussinot[11] and Park[41]. They have been interested in finding a denotational semantics that is able to describe the operational behaviour of the next level in our hierarchy, namely the F-nets. An F-net is roughly speaking pure dataflow plus the addition of a fair merge operator. Without going into details it seems as if both authors have been able to formulate a denotational semantics for F-Nets.

In conclusion we are left with the following table:

Opertational Models	Denotational Models		
	Arcs	Nodes	Nets Behaviour
Pipeline dataflow	?	?	?
F-Nets	Hiatonised Histories	Oracular Hiatonised History Functions	The (de- hiatonised) set of solutions to the recursive set of equations over Hiatonised histories
T-Nets	Hiatonised Histories	Hiatonised History Functions	The (de- hiatonised) solution to a set of recusive equations over hiatonised Functions
Pure Dataflow	Histories	History Functions	Least Fixed Point of the Associated set of equations

To find a denotational semantics for the whole of pipeline dataflow is a difficult problem for which we have not yet found a satisfactory solution.

6 BIBLIOGRAPHY

[1] Ackerman W. B., Dennis J. B. VAL.. A Value oriented algorithmic language, Preliminary Reference Manual.
Computation Structures Group
MIT, Cambridge, Massachusetts.

[2] Adams D. A. A Computation model with Dataflow sequencing
School of Humanities and Science, (Technical Report CS 117), Stanford University, California, Ph.D Thesis, (1968).

[3] Arvind, Gostelow K. P., Some relationships between asynchronous interpreters of a dataflow language,
Formal description of Programming concepts (E. J. Neuhold Editor), pp 95-119,
North Holland Publishing Company, New York, (1977).

[4] Arnold A. Semantique des processus communicants,
RAIRO, Vol. 15, N. 2, pp 103-140, (1981).

[5] Ashcroft E. A., Wadge W., Lucid, a Nonprocedural language with iteration,
Communications of the ACM, Vol. 20, N. 7, pp 519-526, (1977).

[6] Ashcroft E. A., Wadge W., Structured Lucid,
Theory of Computation Report N. 33,
Department of Computer Science, University of Warwick, Coventry, U.K., (1980).

[7] Ashcroft E. A., Wadge W., Lucid, The Dataflow Programming Language
Academic Press, (to be published).

[8] De Bakker J., Mathematical Theory of Program Correctness
Prentice-Hall International Series in Computer Science (1980)

[9] Backus J., Can programming be liberated from the Von Neumann Style? A functional Style and its Algebra of Programs,
Communications of the ACM, Vol. 21, N. 8, pp 613-641 (1978).

[10] Bird R., Programs and Machines,
An introduction to the theory of computation,

Publishers J. Wiley, New York,
Wiley Series In Computing,
(1969).

[11] Bousinnot F., Reseau De Processus Avec Melange Equitable: Un Approche Du Temps Reel,

These de Doctorat D'Etat,
Universite, Paris VII,
(1981).

[12] Brock J. D., Operational Semantics of a Dataflow Language,

S.M. Thesis,
Department of Electrical Engineering and Computer Science, MIT, Cambrigde, Mass.,
(1978).

[13] Brock J. D., Ackerman W. B., An Anomaly in the Specification of Non-determinate Packet Sustems,

Computation Structures Group,
(Note 33-1)
Laboratory for Computer Science, MIT, Cambrigde, Mass.,
(1978).

[14] Cameron P., A Non-procedural Operating Systems Language,

Ph.D Thesis (In preparation),
Department of Computer Science, University of Warwick,
Coventry, U.K.

[15] Conway M. E., Design of a Separable Transition-Diagram Compiler,

Communications of the ACM,
Vol. 6, N. 7, pp 396-408,
(1963).

[16] Davis A. L., The Architecture of and System Method of DDM1: A Recursively Structured Data Driven Machine,

Proceeding of the 5th Annual Symposium on Computer Architecture,
Computer Architecture News,
Vol. 6, N. 7, pp 210-215,
(1978).

[17] Davis A. L., Data Driven Nets: A Maximally Concurrent Procedural Parallel Process Representation For Distributed Computing,

Privately Circulated.

[18] Gale D. & Stewart F.M. Infinite games with perfect information,

Contributions to the theory of games.
Annals of Mathematical studies,
Number 28 Princeton University Press,
Princeton N.J. pp245-266,
(1953)

[19] Dennis J. B., First Version of a Dataflow Procedure Language,

Programming Symposium: Proceedings of Colloque sur la Programmation, (B. Robinet Editor),

Springer-Verlag Lecture Notes in Computer Science, N. 19, pp 362-376, (1974).

[20] Dennis J. B., Misunas D. P., A Preliminary Architecture for a Basic Dataflow Processor,

2nd Annual Symposium on Computer Architecture: Conference Proceedings, pp 126-132, (1975).

[21] Faustini A. A., An Operational Semantics for Pure Dataflow,

9th International Colloquium on Automata, Languages and Programming: Conference proceedings, (1982)

[22] Glauert J., A Single Assignment Language for Dataflow Computing,

M.Sc. Dissertation,

Department of Computer Science, University of Manchester, U.K. (1978).

[23] Gurd J., Watson I., A Multilayered Dataflow Computer Architecture,

Proceedings of the 1977 International Conference on Parallel Processing (J. L. Baer Editor), p 94, (1977).

[24] Hankin C. L., Osman P. E., Sharp J. A., A Dataflow Model of Computation,

Department of Computer Science, Westfield College, Hampstead, London, U.K., (1978).

[25] Henderson P., Functional Operating Systems,

Notes privately circulated, (1981)

[26] Kahn G., The Semantics of a Simple Language for Parallel Programming,

Proceedings of IFIP Congress (J. L. Rosenfeld Editor), pp 471-475, (1974)

[27] Kahn G., McQueen D., Coroutines and Networks of Parallel Processes,

Proceedings of IFIP Congress (B. Gilchrist Editor), pp 993-998, (1977).

[28] Karp R. M., Miller R. E., Properties of a Model of Parallel Computations: Determinacy, Termination and Queueing,
SIAM Journal of Applied Mathematics, Vol. 14, pp 1390-1411, (1966).

[29] Keller R. M., Denotational Models with Undeterminate Operators,
Formal Description of Programming Language Concepts, (E. J. Neuhold Editor), North Holland Publishing Company, pp 337-366, New York, (1977).

[30] Kleene S. C., An Introduction to Metamathematics,
Van Nostrand Company Inc., New York, (1952).

[31] Kosinski P. R., Denotational Semantics of Determinate and Non-Determinate Dataflow Programs,
Ph.D Thesis, Department of Computer Science, MIT, Cambridge, Mass., (1979).

[32] Kosinski P. R., A Straightforward Denotational Semantics for Non-Determinate Dataflow Programs,
Conference Record of the 5th ACM Symposium on Principles of Programming Languages (POPL), pp 214-221, (1978).

[33] Landin P. J., The Next 700 Programming Languages,
Communications of the ACM, Vol. 9, N. 3, pp 157-166 (1966).

[34] Bic L., Protection and security in a dataflow system
Technical report # 26 (Ph.D. Thesis) Department of Information and Computer Science University of California, Irvine Irvine, CA 92717 (1978).

[35] Lindstrom A., Keller, Symposium on Functional Languages and Computer Architecture, Invited paper Goteborg June 1-3, (1981).

[36] Manna Z., Mathematical Theory of Computation,
McGraw Hill Computer Science Series, (1974).

[28] Karp R. M., Miller R. E., Properties of a Model of Parallel Computations: Determinacy, Termination and Queueing,
SIAM Journal of Applied Mathematics, Vol. 14,
pp 1390-1411,
(1966).

[29] Keller R. M., Denotational Models with Undeterminate Operators,
Formal Description of Programming Language Concepts,
(E. J. Neuhold Editor), North Holland Publishing
Company, pp 337-366,
New York,
(1977).

[30] Kleene S. C., An Introduction to Metamathematics,
Van Nostrand Company Inc.,
New York,
(1952).

[31] Kosinski P. R., Denotational Semantics of Determinate and Non-Determinate Dataflow Programs,
Ph.D Thesis,
Department of Computer Science, MIT, Cambridge, Mass.,
(1979).

[32] Kosinski P. R., A Straightforward Denotational Semantics for Non-Determinate Dataflow Programs,
Conference Record of the 5th ACM Symposium on Principles
of Programming Languages (POPL), pp 214-221,
(1978).

[33] Landin P. J., The Next 700 Programming Languages,
Communications of the ACM,
Vol. 9, N. 3, pp 157-166
(1966).

[34] Bic L., Protection and security in a dataflow system
Technical report # 26
(Ph.D. Thesis)
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(1978).

[35] Lindstrom A., Keller, Symposium on Functional Languages
and Computer Architecture,
Invited paper
Goteborg June 1-3,
(1981).

[36] Manna Z., Mathematical Theory of Computation,
McGraw Hill Computer Science Series,
(1974).

[37] McIlroy M. D., Coroutines,
Unpublished Memo,
Bell Telephone Laboratories, Murray Hill,
New Jersey,
(1968).

[38] Minsky M., Finite and Infinite Machines,
Prentice-Hall
Series in Automatic Computation
George Forsythe, Editor
(1972)

[39] Misunas D. P., Error Detection and Recovery in a Dataflow Computer,
Proceedings of the 1976 International Conference
on Parallel Processing, pp 117-122,
(1976).

[40] Pilgram P. T., Translating dataflow into message passing actors.
Ph.D Thesis, (in preparation).
Department of Computer Science,
University of Warwick,
Coventry CV4 7AL

[41] Park D. M. R., Private Communication
(1981)

[42] Ritchie D. M., Thompson K., The UNIX Operating System,
Communications of the ACM,
Vol. 17, N. 7, pp 365-375,
(1974).

[43] Rodriguez J. E., A Graph Model of Parallel Computation,
Ph.D Thesis,
Laboratory for Computer Science,
MIT, Cambridge, Mass.,
(1969).

[44] Rumbaugh J. E., A Dataflow Multiprocessor,
IEEE Transactions on Computers,
C-26, 2, pp 138-146
(1977).

[45] Weng K. S., Stream Oriented Computation in Recursive Dataflows Schemas,
Laboratory for Computer Science, TM 68, MIT,
Cambridge, Mass.,
(1975).

[46] Wadge W. W. An Extensional Treatment of Dataflow Deadlock,
Proceedings of Conference on Semantics of Concurrent
Coputation, Evian,
Sringer-Verlag Lecture Notes on Computer Science, N. 70
pp 285-299,
(1979).

[47] Wadge W. W., Reducibility and Determiness in the Baire Space,
Ph.D Thesis,
Mathematics Faculty,
U.C. Berekley

[48] Wiedmer E., Computing with Infinite Objects,
Theoretical Computer Science, N. 10, pp 133-155,
(1980).

[49] Wiedmer E., Exaktes Rechen Mit Reellen Zahlen und Anderen Unendlichen Objekten,
Ph.D Thesis,
ETH 5975, Zurich,
(1977).

[50] Manna Z., Mathematical Theory of Computation
McGraw-Hill
Series in Computer Science
(1974)

[51] Arvind, Gostelow K.P., Plouffe w., An asynchronous Programming Language and Computing machine",
TR-114,
Department of Information and Computer Science,
University of California , Irvine
(1978)

[52] Yourdon E., Constantine L., L., Structered Design: Fundamentals of a discipline of Computer Programming and systems design.
Prentice-Hall
(1979)

[53] Ritchie D.M., Thompson K., The UNIX time-sharing System
Communications ACM
V. 17 N.6 pp365-375
(July 1974).

[54] Ashcroft E. A., Proving assertions about parallel programs
Journal Computer Systems Sci
V.10 N.1 pp110-135
(1975).

[55] Faustini A.A., Wadge W.W. The cycle sum test for Recursive User Defined Function
Theory of Computation Report
(in Preparation)
Department of Computer Science
University of Warwick
Coventry