

# A Fully Abstract Functional Model for Real Time Systems

Peter Kearney,  
School of Computing and Information Technology, Griffith University,  
Nathan, Queensland. 4111. Australia.

**Abstract.** We introduce a model for real time systems which is based on an extension of the data flow model. The extension is to introduce a special data item (the 'hiaton' or 'wait'). This allows the identification of the index value of a data item in a history with the discrete time at which the data item appeared. To allow convenient abstraction from precise timing requirements we develop a model of nondeterministic real time processes based on functional modelling processes with additional oracle inputs. We show that the resulting model is fully abstract in the sense that processes which induce identical timed input-output behaviour in all contexts are equal.

## 1. Introduction

In this paper we present a model for real time processes which is based on an extension of the data flow model of asynchronous networks.

Recall that a data flow network is a directed graph in which the nodes represent asynchronous processes and the edges represent unidirectional communication lines along which data items flow. In the pure data flow model, which we consider here, edges are assumed to have unbounded buffering capacity. If desired, suitable processes which model bounded buffering capacity may be defined. Each process in a network has a finite number of *input ports* and a finite number of *output ports*.

Data items are received on input ports and sent on output ports. In the study of reactive systems we are often interested in the ongoing behaviour of a system over all time. Thus in general we are interested in the entire sequence of data items generated by a process, over all time, in response to the entire sequence of data items received, over all time. It follows that both input and output sequences may be infinite. For a process with  $m$  input ports and  $n$  output ports, we will refer to an  $m$ -tuple of input sequences as an input history of the process and an  $n$ -tuple of output sequences as an output history of the process.

A *nondeterministic* data flow process may give rise to a number of distinct output histories in response to a given input history. In contrast a *deterministic* (or *functional*) data flow process produces just one output history for each input history. Deterministic data flow provides the underlying computational model for the data flow language LUCID (Wadge and Ashcroft[1985]).

Kahn[1974] presented a successful fixed point semantics for deterministic data flow. In Kahn's approach a deterministic data flow process is characterised by its input-output function which specifies the complete output history for each input history. The Kahn semantics is elegant and provides a convenient basis for specifying and reasoning about deterministic data flow. It is possible, for example, to axiomatise a first order theory, based on the Kahn semantics. Such an axiomatisation can be used to provide a basis for computer assisted verification in a system such as LCF (Paulson[1987]).

## 2. 'Hiatomic' data flow

Our model for real time processes is based on the introduction of a special distinct data item which flows along channels. This special data item has been called a 'hiaton'. The term 'hiaton' is attributed to Wadge and Ashcroft by Faustini[1982]. In Park[1983], the hiaton is used as a basis for a semantics of nondeterministic data flow. Kearney and Staples[1988] refine and extend the work of Park in the semantics of nondeterministic data flow. The use of hiatons for modelling real time processes has been suggested by Faustini and Lewis[1986]. Here we call the hiaton a 'wait' and we denote it by  $\tau$ .

The introduction of waits allows us to use deterministic data flow processes to model real time processes as follows.

First, in common with a number of approaches to real time semantics, we restrict ourselves to considering a discrete time scale, which we model by the positive integers. This model is appropriate for dealing with digital systems.

Next we identify the index value of a data item in a history with the discrete time at which that data item appeared in that history. For example, consider the history

$$h = \tau.\tau.a.\tau.\tau.b.\tau.c.\tau^\omega$$

where the notation  $\tau^\omega$  means an infinite stream of waits. Using our identification we regard the data item  $a$  as being produced at time 3, the data item  $b$  as being produced at time 6 and the data item  $c$  at time 8. Note that  $h$  is of infinite length. We call a history *total* if it has infinite length; otherwise it is called *partial*. An  $n$ -tuple of histories is called total if each of its components is total otherwise it is called partial. We use only total histories in our model, but our analysis uses partial sequences as approximations to total sequences.

We say that a sequence is *defined up to time  $t$*  if it is of length  $t$ . Let  $h = (h_1, \dots, h_n)$  be a partial history such that for each  $i$  between 1 and  $n$ ,  $h_i$  is defined up to time  $t_i$ . Then we say that  $h$  is defined up to time  $t = \min_i(t_i)$ . For example if  $h = (\tau.1.2, \tau.3.\tau.4.\tau.5)$ , then  $h$  is defined up to time 3. Intuitively, we do not have complete information about  $h$  for time 4.

Our model does not define any necessary delay on edges. We assume that the time at which a data item is produced (transmitted) is also the time at which the data item is



available for processing by receiving processes. If desired, communication lines with delays may be modelled by suitable processes.

*Example 1.* A unit delay process. This process delays all input items by one time step:

$$\text{ud}(X) = \tau.X.$$

Here we have used the notation '.' for sequence concatenation.

We require our modelling processes to satisfy the following modelling condition. A formal definition is given in section 5.

*Modelling condition.*

- (1). Output at time 1 is defined before any input is read. (The *initialising* condition).
- (2). Whenever input becomes defined for a longer time, output becomes defined for a longer time. (The *t-monotonicity* condition).

The modelling condition implies that for each time  $t$ , output at times less than or equal to  $t$  is independent of input at times greater than or equal to  $t$ . Also note that the causality condition implies that if the input history is total then so is the output history.

*Example 2.* A real time merge. This process merges items from two input streams onto a single output stream, in the order in which they appear. The process is defined so as to be biased towards its left input, in the sense that if a data item  $x$  is available on the left input at time  $t$  and a data item  $y$  is available on the right input also at time  $t$ , then the data item  $x$  appears before  $y$  in the output.

$$\text{rtm}(X, Y) = \tau.m(X, Y)$$

$$m(X, \perp) = \perp$$

$$m(\perp, Y) = \perp$$

$$m(a.X, b.Y) = a.b.m(X, Y).$$

### 3. Abstraction and Nondeterminism

As discussed so far, our model allows us to specify and verify properties of real time processes using standard and powerful induction techniques. However our specifications are very strict: a real time process specified by a modelling function must produce the correct values at the precisely specified times. In practice such strictness may not be appropriate. For specification purposes we may wish to specify that data values be produced in a certain time range (sometimes called time windows). To allow such

abstraction, while retaining the convenience of functional modelling, we extend our model so that each modelling function has an extra *oracle* input which provides at the functional level the precise information from which we wish to abstract at a higher level.

*Example 3.* The following process  $f$  outputs infinitely often the data item  $x$  at times  $t_1, t_2, \dots$ , where the times  $t_1, t_2, \dots$  are supplied in an oracle sequence  $o = t_1.t_2.\dots$

$$f(o) = \tau.f(o, 1).$$

$$f(t.o, T) = \tau^{t-T-1}.x.f(o, t).$$

By ranging over a set of oracle sequences, a corresponding set of timed output behaviours will result. However our analysis may be conducted at the functional level, using oracle sequences as parameters.

Over what set of oracle values is it appropriate to range? In practice this will depend on the process being specified and the information from which abstraction is being made. In Example 3 above we may wish to consider ranging over all strictly increasing sequences  $t_1.t_2.\dots$  or over oracle sequences  $t_1.t_2.\dots$  such that each  $t_i$  is within a time window  $[t_{i,l}, t_{i,r}]$ .

For simplicity in theory, however, it is convenient to fix a set  $\Delta$  of histories over which oracles may vary, and use variation over  $\Delta$  in all modelling functions. For practical use, oracles from  $\Delta$  may be transformed or filtered to other histories for consumption by the modelling function, as illustrated in Figure 1 below.

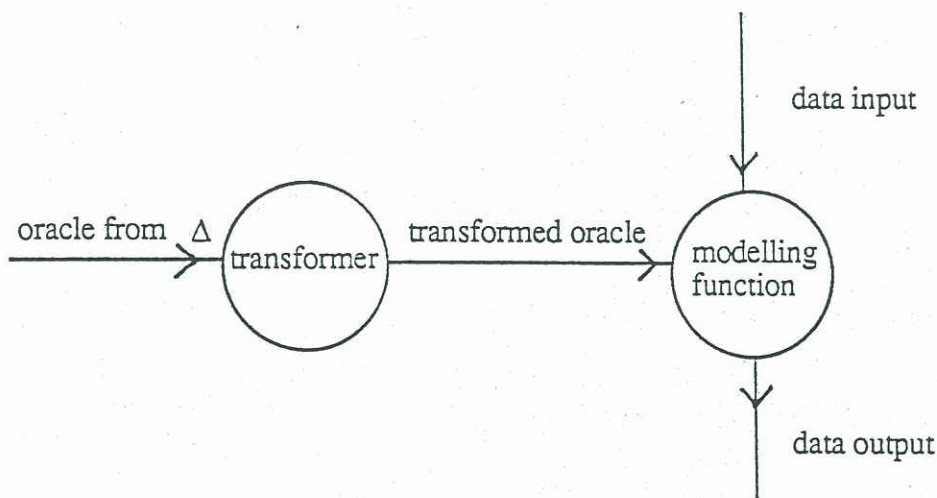


Figure 1.



In choosing the oracle set  $\Delta$  we must balance the desire for theoretical simplicity against the need to use  $\Delta$  to produce practically useful classes of transformed oracles. An extreme but commonly encountered example of abstracting from timing information is the requirement that some data item appear *eventually* but placing no apriori bound on the time at which it will appear. A requirement that an event occur eventually but with no bound placed on the length of time before the event occurs is often called a *fairness* requirement. To model such fairness requirements it is sufficient to have available *fair* oracles: A fair oracle is an infinite sequence of 0's and 1's which contains an infinity of 0's and an infinity of 1's.

*Example 4.* A process which imposes arbitrary finite delays on its input stream may be defined as follows,

$$d(X, \delta) = \tau.d'(X, \delta).$$

$$d'(\perp, \delta) = \perp.$$

$$d'(a.X, 0.\delta) = a.d'(X, \delta)$$

$$d'(a.X, 1.\delta) = \tau.d'(a.X, \delta),$$

assuming that  $\delta$  is a fair oracle.

For practical purposes it seems that using fair oracles we may define oracle transformers to produce desired classes of transformed oracles.

For the sake of modularity, we desire that all networks and processes in our theory have just *one* oracle input. This requirement creates the need, within a network, to convert the network oracle stream into individual oracle streams, one for each network component.

Our basis for that is an oracle distribution process **Dist** which has a single oracle stream as input, and two oracle streams as outputs. Like oracle streams, the **Dist** process has a special role in our model.

The essential requirements for **Dist** and oracle streams are:

- (i). When the input to **Dist** ranges over all possible oracle streams, each output of **Dist** ranges over all possible oracle streams.
- (ii). It must be possible to define a process which has an oracle input and a data output, such that when the oracle input ranges over all possible oracle streams the output ranges over all possible fair sequences of 0's and 1's. Such a process would be used as illustrated in Figure 1.

It is unimportant how these requirements are satisfied. To be definite, we proceed as follows.

The left output stream of the process **Dist** is the sequence of values occurring at odd positions in the input oracle stream. The right output stream of **Dist** is the sequence of values occurring at even positions in the input. Formally, writing  $s$  for the input stream of

the oracle distributor, its left and right output streams  $\lambda(s)$ ,  $\rho(s)$  are oracle sequences defined by :

$$\lambda(s)(n) = s(2n - 1) , n \geq 1.$$

$$\rho(s)(n) = s(2n) , n \geq 1.$$

To provide more than two oracles for use by processes in a network, we use a binary tree of distribution processes. Figure 2 shows an example network with three process components and illustrates a tree of oracle distributors which can be used to provide the three oracle sequences required.

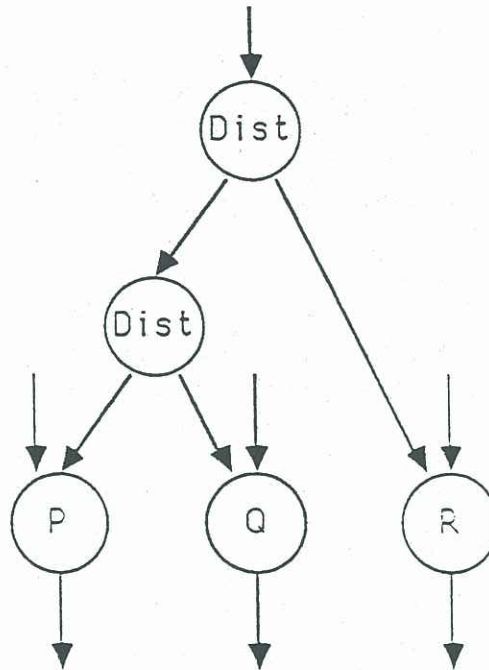


Figure 2.

To ensure that all oracle sequences used in a network are fair, we require that network oracle inputs are *strongly fair*. An oracle sequence  $s$  is strongly fair if, for every binary tree  $T$  of distribution processes, and for every node  $N$  in  $T$ ,  $N$  outputs fair oracle streams when  $s$  is the input to  $T$ .

An oracle sequence can be used in one of two ways. It can be used for the special purpose of being distributed into two oracle sequences. Alternatively, it could be used for some specific computational purpose, such as controlling an arbitrary delay, as in example 4.

We do not claim that the use of strongly fair instead of fair oracle sequences is appropriate for specific computational purposes. Rather, oracle streams are intended to be converted to fair streams before being used for specific computational purposes. Our



strongly fair oracles allow us to model processes which have access to an unbounded number of independent sources of fairness.

#### 4. Full Abstraction and equivalence classes of modelling functions

We desire that our model of real time processes be fully abstract. Informally, *full abstraction* means that if two real time process networks  $N_1$  and  $N_2$  are characterised by different models then this distinction should make some practical difference in some context in which  $N_1$  or  $N_2$  may be used. By a practical difference we mean a difference in timed input-output behaviour.

We have incorporated a form of nondeterminism in our model as an abstraction device: our modelling functions map (timed) input histories and oracles to (timed) output histories and the range of possible timed results is obtained by ranging over all strongly fair oracle inputs. The result of including this abstraction is that modelling a real time process by a single function does not lead to a fully abstract theory. Two different functions may behave in all contexts in ways which are indistinguishable when we range over all possible oracle inputs.

To achieve full abstraction, we define an equivalence relation ('behaviour equivalence') on modelling functions. Our model of a nondeterministic real time network is a behaviour equivalence class of modelling functions.

Let  $f$  and  $g$  be modelling functions. We say  $f$  is *behaviour sub-equivalent* to  $g$  if for every total history  $X$  and every strongly fair oracle  $\delta$ , there exists a strongly fair oracle  $\delta'$  such that  $f(X, \delta) = g(X, \delta')$ . Further, we say that two processes are *behaviour equivalent* if each is sub-equivalent to the other. Thus two processes  $f$  and  $g$  are behaviour equivalent if their history relations on complete histories are equal. It is easy to see that if  $f$  is, for example, *not* sub-equivalent to  $g$  that will result in an observable difference in timed input-output behaviour, since  $f$  is capable of a behaviour in response to  $X$  which  $g$  cannot match.

Of course, we must show that behaviour equivalence is a congruence with respect to network construction, that is, that if we build two networks in identical fashion from equivalent components then we obtain equivalent networks.

In the remainder of this paper we give precise definitions and results formalising the ideas we have introduced above. Space restrictions preclude the inclusion of any proofs.

#### 5. Deterministic modelling processes

For simplicity we assume that all data streams carry data of the same type  $C$ . For our models, as discussed in 2 above, we adjoin a new value  $\tau$ , the *wait*, to give a data type  $D = C \cup \{\tau\}$ .

$D^\infty$  is the set of finite and denumerably infinite sequences of elements of  $D$ , including the empty sequence, written  $\perp$ . For  $n \geq 1$ , elements of  $(D^\infty)^n$  may be called *histories*. We may write  $\perp$  for any history all of whose elements are  $\perp$ .

We write  $\Delta$  for the set of strongly fair oracles.

Extending [Kahn, 74], we characterise deterministic processes as continuous functions of some type  $(D^\infty)^m \times \Delta \rightarrow (D^\infty)^n$ ,  $m, n \geq 0$ . A function of this type characterises a deterministic data flow process with  $m$  input data ports, a single oracle input port and  $n$  output ports. Note that modelling functions are continuous on total oracle inputs, with respect to the standard prefix ordering on total oracle sequences. The splitting process *Dist*, already defined, is clearly continuous with respect to that standard partial order.

For brevity we will write  $[m \rightarrow n]$  for  $(D^\infty)^m \times \Delta \rightarrow (D^\infty)^n$ . For  $\underline{h} \in (D^\infty)^m$ ,  $\delta \in \Delta$  and  $f \in [m \rightarrow n]$  we may write  $f(\underline{h})$  instead of  $f(\underline{h}, \delta)$  when the abbreviation is not ambiguous in context. We may also write  $\perp$  for the function with constant value  $\perp$ .

As discussed in 2 above, the processes used in our deterministic modelling satisfy a *modelling* condition. In our technical development we use a condition based on the domination relation.

Intuitively, we say that  $\underline{h}$  dominates  $\underline{k}$  when  $\underline{k}$  is a prefix of  $\underline{h}$  and, in the case that  $\underline{k}$  is partial, the maximum time for which  $\underline{h}$  is totally defined is greater than the maximum time for which  $\underline{k}$  is totally defined. Here is a more precise definition.

**5.1. Definition.** For history vectors  $\underline{h} = (h_1, \dots, h_m)$ ,  $\underline{k} = (k_1, \dots, k_m)$ , say  $\underline{h}$  dominates  $\underline{k}$ , write  $\underline{h} \text{ d } \underline{k}$ , if for all  $i$ ,  $1 \leq i \leq m$ , either  $h_i = k_i$  and  $h_i, k_i$  are infinite or  $h_i > k_i$ .

For example,  $\tau \text{ d } \perp$ , but not  $\perp \text{ d } \perp$ .

The following extension of domination to continuous functions is a key concept for our analysis, and also leads directly to our *t-monotonicity* condition.

**5.2. Definition.** Let  $f, g \in [r \rightarrow s]$ . Say  $f$  dominates  $g$  if for all  $\underline{h} \text{ d } \underline{k}$ , and all  $\delta \in \Delta$ ,  $f(\underline{h}, \delta) \text{ d } g(\underline{k}, \delta)$ .

We write  $f \text{ d } g$  when  $f$  dominates  $g$ .

**5.3. Definition.** Let  $f \in [r \rightarrow s]$ . Say  $f$  is *t-monotonic* if for all  $\underline{h} \text{ d } \underline{k}$ , and all  $\delta \in \Delta$ ,  $f(\underline{h}, \delta) \text{ d } f(\underline{k}, \delta)$ .

Note that  $f$  is *t-monotonic* if and only if  $f \text{ d } f$ .

We extend the above definitions to vectors of functions as follows.

**5.4. Definition.** Let  $\underline{f} = (f_1, \dots, f_n)$ ,  $\underline{g} = (g_1, \dots, g_n)$  be  $n$ -tuples of functions. Say  $\underline{f}$  dominates  $\underline{g}$  if  $f_i \text{ d } g_i$  for all  $i$ ,  $1 \leq i \leq n$ . Write  $\underline{f} \text{ d } \underline{g}$ . Say  $\underline{f}$  is *t-monotonic* if  $\underline{f} \text{ d } \underline{f}$ .

The initialising condition which completes the concept of modelling is also easily expressed in terms of domination.

**5.5. Definition.** Say  $f \in [r \rightarrow s]$  is *initialising* if  $f(\perp, \perp) \text{ d } \perp$ . A vector  $\underline{f}$  of functions is *initialising* if each component of  $\underline{f}$  is *initialising*.



**5.6. Definition.** A process  $f$  is a *modelling process* if it is initialising and  $t$ -monotonic.

The modelling functions  $(D^\infty)^m \times \Delta \rightarrow (D^\infty)^n$  form a complete subset of the larger cpo of continuous functions. However it is not appropriate to focus attention purely on this subset, intuitively because the modelling functions represent limiting behaviours which are approximated by non-modelling processes. For example there is no least modelling function.

As outlined in the next section, we restrict the class of operators modelling network construction schemes so that the least fixed points of such operators are modelling functions. Those least fixed points may be calculated as least upper bounds of, in general, non-modelling functions.

## 6. Modelling Operators and Network Construction

A fundamental requirement for the discussion of processes is to have a vocabulary for describing the contexts into which a process may be placed. Intuitively, such a context is a network of processes which has a 'hole' into which the process under discussion may be placed. We formalise this requirement by considering (continuous) operators on the continuous functions which model deterministic processes. Intuitively, when a process  $f$  is placed in the 'hole' in a context  $F$ , the resulting network is  $F(f)$ .

More generally, operators can also be used to define functions recursively as discussed below.

It is convenient for our analysis to allow contexts with several 'holes'. That in turn makes it natural to consider operators for which the values, as well as the arguments, are vectors of functions. Accordingly we begin as follows.

**6.1. Definition.** An *operator* is a continuous function from  $m$ -tuples to  $n$ -tuples of the continuous functions on histories. A typical operator will be a continuous function from  $[p_1 \rightarrow q_1] \times \dots \times [p_m \rightarrow q_m]$  to  $[r_1 \rightarrow s_1] \times \dots \times [r_n \rightarrow s_n]$ . Intuitively, this operator represents an  $n$ -tuple of contexts. Each of the contexts has  $m$  'holes'. For each context, the  $i$ -th 'hole' can be filled by a deterministic process of type  $[p_i \rightarrow q_i]$ . For the  $j$ -th context, when all 'holes' are filled the resulting network defines a deterministic process of type  $[r_j \rightarrow s_j]$ ,  $j = 1, \dots, n$ .

**6.2. Definition.** For operators  $F$  and  $G$  of compatible types we denote the composite operator  $F \ G$ . That is, for all function vectors  $\underline{f}$  in the domain of  $G$ ,

$$(F \ G)(\underline{f}) = F(G(\underline{f})).$$

The modelling condition on functions extends naturally to operators as follows.

**6.3. Definition.** An operator  $F$  is *dominance preserving* if for all  $\underline{f}, \underline{g}$  in the domain of  $F$ ,  $\underline{f} \sqsubseteq \underline{g}$  implies  $F\underline{f} \sqsubseteq F\underline{g}$ .

**6.4. Definition.** An operator  $F$  is *initialising* if  $F(\underline{\perp})$  is initialising.

**6.5. Definition.** An operator is a *modelling operator* if it is initialising and dominance preserving.

It should be noted however that we shall use only a specific class MO of modelling operators, defined below.

**6.6. Lemma.** If  $F$  is a modelling operator and  $\underline{f}$  is a modelling function in the domain of  $F$ , then  $F(\underline{f})$  is also a modelling function.

Since we are interested in modelling processes, we show that recursive definitions using modelling operators define modelling processes. This shows that the class of modelling processes is closed under network construction using modelling operators. It also shows that the class of modelling processes is closed under recursion using modelling network schemes.

**6.7. Theorem.** If  $F$  is a modelling operator on  $[r_1 \rightarrow s_1] \times \dots \times [r_n \rightarrow s_n]$  and  $f$  is the least fixed point of

$$\underline{f} = F(\underline{f})$$

then  $\underline{f}$  is modelling.

The class MO of operators considered here is the class of operators to be used for modelling network constructions. First we describe the basic classes of operators to be used in the definition of MO. Note that most of these basic operators are not themselves in MO.

From this point we make occasional use of  $\lambda$  notation in the description of functions.

**Modelling Constant Operators.** Let  $g$  be a modelling function of type  $[r \rightarrow s]$  and let  $f_i$  be variables of type  $[p_i \rightarrow q_i]$ ,  $i = 1, \dots, m$ .

Then  $\lambda(f_1, \dots, f_m).g$  is a modelling constant operator from  $[p_1 \rightarrow q_1] \times \dots \times [p_m \rightarrow q_m]$  to  $[r \rightarrow s]$ , whose value everywhere is  $g$ .

**T-monotonic Constant Operators.** Let  $c$  be a t-monotonic function of type  $[r \rightarrow s]$  and let  $f_i$  be variables of type  $[p_i \rightarrow q_i]$ ,  $i = 1, \dots, m$ .

Then  $\lambda(f_1, \dots, f_m).c$  is a t-monotonic constant operator from  $[p_1 \rightarrow q_1] \times \dots \times [p_m \rightarrow q_m]$  to  $[r \rightarrow s]$ , whose value everywhere is  $c$ .

**Projection Operators.** Let  $f_i$  be of type  $[p_i \rightarrow q_i]$ ,  $i = 1, \dots, m$ . Then  $\lambda(f_1, \dots, f_m).f_i$  is a projection operator from  $[p_1 \rightarrow q_1] \times \dots \times [p_m \rightarrow q_m]$  to  $[p_i \rightarrow q_i]$ . We shall denote this operator by  $P_i$ , the types being implied by the context.

**Identity Operator.** We denote the identity operator by  $\text{Id}$ . It is defined by  $\text{Id}(\underline{f}) = \underline{f}$ .

**Disjoint Union Operator.** Disjoint union models the operation of forming a process from two component processes by grouping them together without interconnection. Two oracle sequences are obtained from the single oracle sequence of the disjoint union by the oracle distribution process  $\text{Dist}$ . Thus the disjoint union operation is as depicted in Figure



3. The input streams  $OS_1$  and  $OS_2$  are the oracle streams for each of the component processes.

For each pair  $[p \rightarrow q]$  and  $[r \rightarrow s]$  of function types there is a disjoint union operator  $DU$  from  $[p \rightarrow q] \times [r \rightarrow s]$  to  $[(p+r) \rightarrow (q+s)]$ .

$(DU(f_1, f_2))(h_1, \dots, h_p, h_{p+1}, \dots, h_{p+r}, \delta)$  is defined to be  $(f_1(h_1, \dots, h_p, \lambda(\delta)); f_2(h_{p+1}, \dots, h_{p+r}, \rho(\delta)))$ .

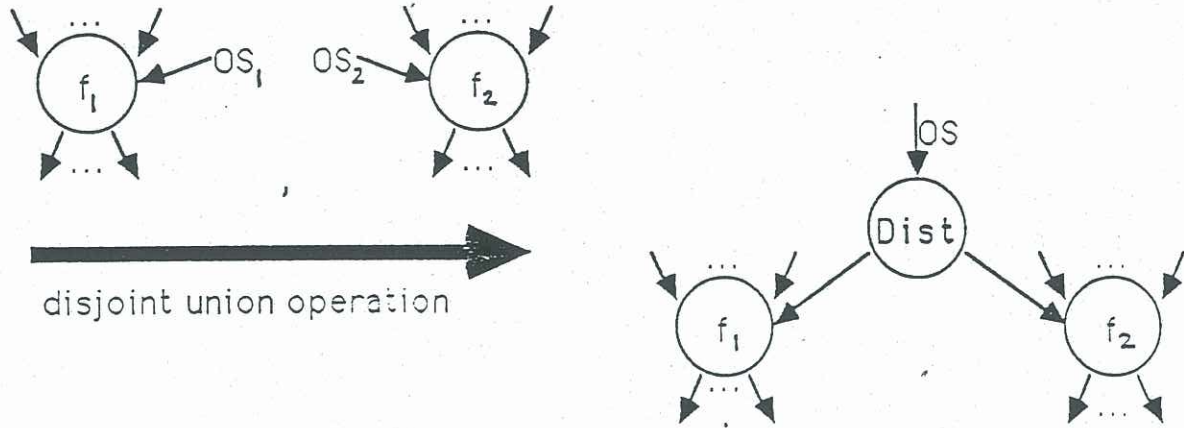


Figure 3. Sketch of the disjoint union operation.

**Composition Operator.** The composition operators form a process from two component processes by feeding the output of one into the input of the other. As in the case of disjoint union, the component processes receive independent oracle inputs as a result of splitting the single oracle sequence of the composition.

Let  $g$  be of type  $[p \rightarrow q]$  and  $f$  of type  $[q \rightarrow r]$ . Then the composition operator,  $Comp$ , from  $[q \rightarrow r] \times [p \rightarrow q]$  to  $[p \rightarrow r]$  is defined by

$$Comp(f, g)(h, \delta) = f(g(h, \lambda(\delta)), \rho(\delta)).$$

Figure 4 illustrates the composition operator.

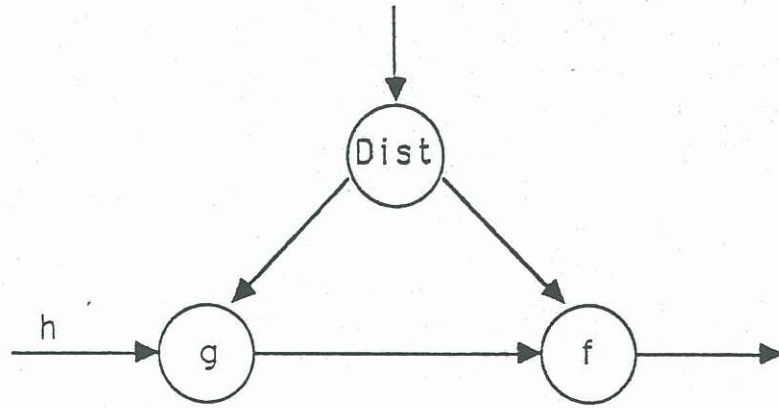


Figure 4. Sketch of the composition operation.

**Link Operator.** These operators are designed to form a new process from a given process by connecting an input port to an output port. Note that linking is more general than composition since it provides for the output of a process to be fed back to its own input. In this case we need to calculate the history on the looped data stream as a least fixed point, following the Kahn semantics.

Let  $f \in [p \rightarrow q]$  and let  $i \in \{1, \dots, p\}$ ,  $k \in \{1, \dots, q\}$ .

Intuitively  $\text{LINK}_{ki}(f) \in [(p-1) \rightarrow (q-1)]$  is as depicted in Figure 5. Formally, it is defined as follows.

For  $r = 1, \dots, k-1, k+1, \dots, q$

$$[\text{LINK}_{ki}(f)(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_p, \delta)]_r = [f(x_1, \dots, x_{i-1}, H, x_{i+1}, \dots, x_p, \delta)]_r$$

where  $H$  is the least fixed point of

$$H = [f(x_1, \dots, x_{i-1}, H, x_{i+1}, \dots, x_p, \delta)]_k.$$

Note that oracle lines cannot be linked.

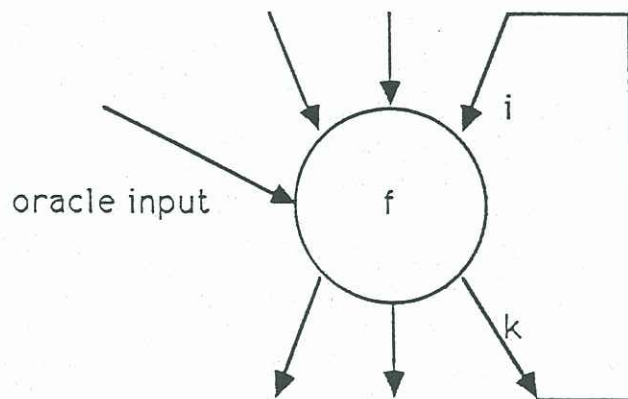


Figure 5. Sketch of the Link operation.



## Function Tupling Operators

For  $i = 1, \dots, m$ , let  $O_i$  be an operator from  $[p_1 \rightarrow q_1] \times \dots \times [p_n \rightarrow q_n]$  to  $[r_i \rightarrow s_i]$ . Define the operator  $[O_1, \dots, O_m]$  from  $[p_1 \rightarrow q_1] \times \dots \times [p_n \rightarrow q_n]$  to  $[r_1 \rightarrow s_1] \times \dots \times [r_m \rightarrow s_m]$  by :  
 $[O_1, \dots, O_m](f_1, \dots, f_n) = (O_1(f_1, \dots, f_n), \dots, O_m(f_1, \dots, f_n))$ .

This completes the definition of the basic classes of operators used for the definition of MO. Next we define a set DP of dominance preserving operators.

**Definition.** The set DP is the least set of operators closed under the following conditions.

1. The t-monotonic constant operators, the modelling constant operators and the projection operators belong to DP.

2.  $DU, Id, Comp \in DP$ .

3. For all  $G_1, \dots, G_m \in DP$ ,  $[G_1, \dots, G_m] \in DP$ .

4. For all  $G_1, G_2 \in DP$ ,  $G_1 G_2 \in DP$ .

It is straightforward to verify that all operators in DP are dominance preserving.

Now we define the class MO recursively as follows.

**Definition.** The set MO is the least set of operators closed under the following conditions. Recall that Comp is the composition operator on processes;  $Comp [G, D]$  denotes the (operator) composition of the operators Comp and  $[G, D]$ .

1. The modelling constant operators are in MO.

2. For all  $G \in MO$  and  $D \in DP$ ,  $Comp [G, D] \in MO$ .

3. For all  $G \in MO$ ,  $LINK_{ki} G \in MO$ .

4. For all  $G \in MO$  and every t-monotonic constant operator C,  $Comp [C, G] \in MO$ .

5. For all  $G_1, G_2 \in MO$ ,  $DU [G_1, G_2] \in MO$ .

6. For all  $G_1, \dots, G_m \in MO$  and all  $i$  such that  $1 \leq i \leq m$ ,  $P_i [G_1, \dots, G_m] \in MO$ , where  $P_i$  is the  $i$ -th projection operator.

7. For all  $G_1, \dots, G_m \in MO$ ,  $[G_1, \dots, G_m] \in MO$ .

8. For all  $G \in MO$  and  $D \in DP$ ,  $G D \in MO$ .

9. For all  $G_1, G_2 \in MO$ ,  $G_1 G_2 \in MO$ .

We can show that all operators in MO are modelling. It follows that for each operator  $G$  in MO the least fixed point of  $G$  is a modelling process. That least fixed point defines a modelling constant operator which is also in MO.

## 7. Equivalence classes of modelling processes

**Definition.** Let  $f, g$  be modelling processes of type  $[r \rightarrow s]$ . We say  $f$  is *behaviour sub-equivalent* to  $g$ , written  $f \ll g$ , if for all infinite  $X$  and  $\delta \in \Delta$  there exists a  $\delta' \in \Delta$  such that  $f(X, \delta) = g(X, \delta')$ . More generally if  $\underline{f} = (f_1, \dots, f_m)$ ,  $\underline{g} = (g_1, \dots, g_m)$  are vectors of modelling processes of the same type,  $\underline{f} \ll \underline{g}$  means that  $f_i \ll g_i, i = 1, \dots, m$ .

**Lemma.** Associativity of disjoint union with respect to behaviour equivalence.

$$DU(DU(f_1, f_2), f_3) \equiv DU(f_1, DU(f_2, f_3)).$$

**Lemma.** Associativity of Composition with respect to behaviour equivalence.

$$\text{Comp}(\text{Comp}(f_1, f_2), f_3) \equiv \text{Comp}(f_1, \text{Comp}(f_2, f_3)).$$

To show that behaviour equivalence is a congruence with respect to network construction we prove a theorem whose intuitive content is:

if two processes are behaviour equivalent then so are their substitutions in any modelling context.

To achieve this result we must first give a formal definition of the modelling contexts being considered. Recall that we construct network processes only as the least fixed points  $z$  of equations

$$z = G(z)$$

where  $G$  is an operator in  $MO$ . Thus, each result of substituting a process into a modelling context is such a  $z$ .

Our formal model for substituting a process  $f$  to achieve  $z = G(z)$  is that

$$G(w) = F(f, w)$$

where  $F \in MO$  also. Thus a formal statement of the theorem we prove is:

**Theorem.** For all  $F \in MO$  of type  $t \times u \rightarrow u$  and all modelling processes  $f$  and  $g$  of type  $t$ , if  $f \equiv g$  and if  $y, z$  are the least fixed points of  $y = F(f, y)$ ,  $z = F(g, z)$  respectively, then  $y \equiv z$ .

For convenience and generality we extend this result to vectors  $\underline{f}, \underline{g}$  of functions, and to operators with vector results. Thus the theorem we prove is the following. Here the notation  $F(\underline{f}, \underline{z})$ , for example, indicates some partition of the arguments of  $F$  into two vectors.

**Theorem.** For all  $F \in MO$  of type  $\underline{t} \times \underline{u} \rightarrow \underline{u}$  and all  $\underline{f}$  and  $\underline{g}$  of type  $\underline{t}$ , if  $\underline{f} \equiv \underline{g}$  and if  $\underline{y}, \underline{z}$  are the least fixed points of  $\underline{y} = F(\underline{f}, \underline{y})$ ,  $\underline{z} = F(\underline{g}, \underline{z})$  respectively, then  $\underline{y} \equiv \underline{z}$ .

## 8. Conclusion

We have presented a fully abstract model for nondeterministic real time processes. The introduction of nondeterminism allows abstraction from precise timing requirements.



The use of the functional level of analysis permits the use of powerful verification techniques and tools.

A limitation of our model is the restriction to the use of a discrete time scale. While this will often be appropriate for the analysis of real time systems, it seems that the use of more general time models could sometimes be more convenient. It is possible to extend the present approach by the use of timestamps: extra data values tagging data items which represent the times at which data values are produced. We intend to analyse this extension.

## References

A.A. Faustini, *The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow*, Ph.D. Thesis, University of Warwick, 1982.

A.A. Faustini and E.B. Lewis, Towards a Real-Time Dataflow Language, *IEEE Software*, 3, 1, (January 1986), 29-35.

G. Kahn, The Semantics of a Simple Language for Parallel Programming, in: *IFIP Proceedings 1974*, (North-Holland, Amsterdam, 1974) 471-475.

P. Kearney and J. Staples, An Extensional Fixed Point Semantics for Nondeterministic Data Flow, Computing and Information Technology Research Report 26, Griffith University, 1988. To appear in *Theoretical Computer Science*.

D. Park, The "Fairness" Problem and Nondeterministic Computing Networks, in: J.W. de Bakker and J. van Leeuwen, eds., *Foundations of Computer Science IV.2*, Mathematical Centre Tracts 159, (Amsterdam, 1983) 133-161.

L.C. Paulson, *Logic and Computation*, C.U.P., Cambridge 1987.

W. W. Wadge and E.A. Ashcroft, *Lucid, the Dataflow Programming Language*, New York, Academic, 1985.