Semantics of Digital Networks Containing Indeterminate Modules

Robert M. Keller
University of Utah

Prakash Panangaden
Cornell University
TR 85-717
December 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

We discuss a formal model based upon dataflow, usable for high-level digital hardware design, among other things. One of our goals is to give a denotational semantics for this model, which includes indeterminate modules. While it is well-known that denotational semantics for networks containing only determinate modules can be simply expressed as a composition of stream functions, this approach has previously been shown unacceptable for networks with indeterminate modules. Our approach is to devise composition rules based on modelling a network by the set of its possible behaviors, *i.e.* sequences of computational events, where each event is the appearance or consumption of a token on a data path. A sequence of such events is called a *history* and a set of such histories is called an *archive*. We give composition rules that allow us to derive an archive for a network from the archive of its constituents. We show how causal and operational constraints on network behavior can be inferred from the specification of archives. We also present a construction which allows us to obtain the denotation of networks containing loops by a process of successive approximations. This construction is carried out using a construction resembling the category-theoretic notion of *limit*, which differs from that of more traditional domain theory.

1. Introduction

There has been recent interest in the use of dataflow concepts for high-level hardware design languages. The principal attraction is the ability to compose units of well-defined behavior into more complex units, while at the same time paying minimal attention to details such as system timing. Our previous work [Keller 80] demonstrated the concept, with an example carried through to a realizable logic level. More recently, there has been at least one "silicon compiler" designed using this approach [Jhon 85], including giving the necessary attention to layout schemes. Figure 1 demonstrates the approach using a simple standard example. It shows the first-level decomposition of a *routing switch*, a module which receives two streams of input packets, and routes them to one of two possible outputs depending on some information in the packets. Shown therein are two module types, *disc*, which discriminates packets based on routing information, and *merge,* which interleaves streams from two inputs in whatever order they happen to arrive.

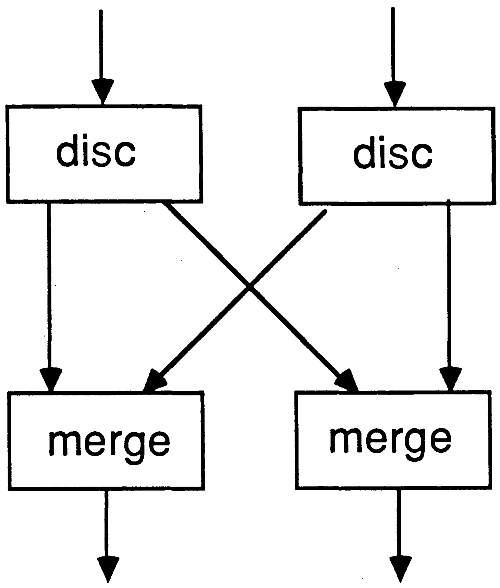

Figure 1: Routing switch composed of dataflow modules.

The present paper concerns itself with analysis and understanding, rather than synthesis tools. Of primary interest is a semantic definition method for a fairly general class of digital networks. We suggest that the model presented herein provides a straightforward model which is easily shown correct.

Component modules of the type discussed above breakdown into two major categories, both of which are illustrated in the above example: *determinate* (or *functional*) in which each output stream is a mathematical function of the input streams, and indeterminate, in which one or more outputs cannot be so-characterized. In the example above, *disc* is determinate, since each of its outputs is determined uniquely from the input stream, while *merge* is indeterminate, since the output stream depends on relative arrival orders in the input streams.

Determinate modules have the advantage that, when composed, they yield a network which behaves as a determinate module. This fact has been pointed out, for example, in the pioneering work of Patil [Patil 70], and of Kahn [Kahn 74]. Indeterminate modules, of course, compose as well, but whereas determinate modules can be nicely characterized by recursive stream functions, an equally succinct notation for composing indeterminate modules has been lacking. In earlier work [Keller 78], we showed that the merge module is not representable by a function with a domain which is a Cartesian product of *any* pair of domains, one for each input, regardless of structure. We inferred that some cross-domain information must be taken into account in characterizing behavior. Brock and Ackermann [Brock 81] showed that (even if cross-domain information is taken into account) the merge module is not representable by a function from input histories into output histories. They indicate that this shows that "history relations" are insufficient to represent the complete behavior of such modules. Since our method here does indeed entail a form of history relation, it might seem that this result is worded too sweepingly. However, reconciliation is to be found in that their informally-presented method can be demonstrated using our techniques.

2. Modules and Networks

Programs or systems in our model are networks of *modules*. Operationally, a module may carry an internal state from a possibly-infinite state set and communicates with its environment by sending and receiving value-carrying *tokens*. The behavior of some modules could, if desired, be described by a sequential program. At any step of its execution, the program may specify polling of its inputs, consuming zero or more input tokens, changing its internal state (i.e. assigning to some internal variables or moving to the next instruction in sequence) and production of tokens on zero or more of its outputs. Our modules are more general than those of [Kahn 74] in that ours can act on the absence of a token [Keller 77] , or equivalently, generally don't need to wait for the presence of tokens.

In addtion to the *disc* and *merge* modules described above, other examples of modules which might be found in digital networks are:

Add module: This module awaits a pair of tokens, one on each of its input arcs. When both tokens are present, it consumes the input tokens and produces an output token on its output arc, which has the value of the sum of the two input token values.

Fork module: This module has one input and two output arcs. It copies each input token onto both of its output arcs.

Choice module: This module has one input and two output arcs. Upon receiving a token at its input, it consumes the latter and produces a token with the same value on *either* (but not both) of its output arcs. Although the choice module might seem to be too random to be useful, it is of interest in modeling situations where the decision of *which* output arc is actually made deterministically, but with information which is not explicitly represented.

k-bounded Queue module: This module produces output tokens with the same values as its input tokens, in the same order received. However, at any one time, the difference between the number of tokens consumed and produced may be up to k in number

Unbounded Queue (UBQ) module: This module behaves as above, except there is no bound on the number of tokens contained at a given time. This may seem unrealistic for hardware networks, but it models the case where the bound is so large that we don't care what it is.

By a *prototype* of a module, we mean one which exemplifies its behavior in terms of designated arc labels (b, c, d, etc.). To emphasize that behavior will be described in terms of these arcs, we subscript the module name for the prototype with its arc labels. It is understood that when a module is used in a network, these labels will generally be different, since we shall require the set of arc labels in a network to be unique. In this case, the behavior of a module in question is simply given by *renaming* arcs mentioned in its behavioral description with the appropriate labels. Figure 2 shows a prototype for the $fork_{bcd}$ module.

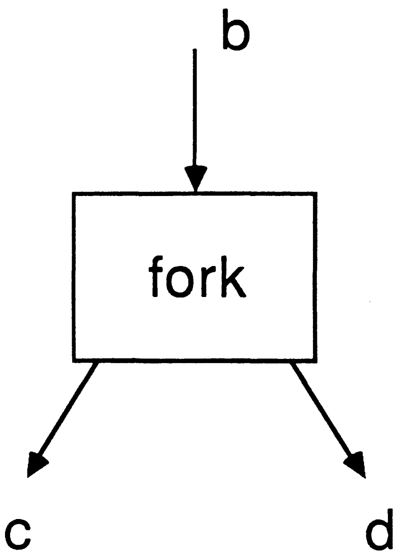

Figure 2: Fork_{bcd} module

A *network* is a directed graph, each node of which is labelled with the name of a module. Each arc connects up to two nodes, one as input and one as output. Arcs which are only input or output to the entire network (*i.e.* do not connect two modules within it) are labelled only once. Interconnecting arcs have *two* labels, an *output* label associated with the node from which the arc is output, and an *input* label associated with the node to which the arc is input. This dual labelling will be helpful when we describe behavior of interconnected modules. Figure 3 shows a fork module in the context of a network. We notice that each arc inherits a label as the fork and has a second label from the other node to which the arc is connected.

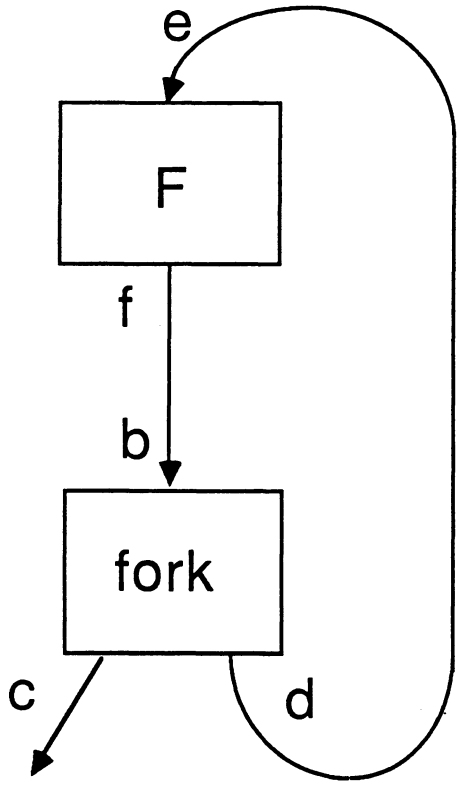

Figure 3: Example of a loop network

3. Behavior Modeling

Our model is both operational and denotational. It is operational in that the behavior can be clearly discerned from examining the denotation, and denotational because compositions of networks correspond to compositions of behaviors.

The model is base on the notion of *events*, *histories*, and *archives*. There are two types of *events,* both represented as pairs of the form

(b v)

where *b* is an arc label and *v* is the value of a token. A *production event* has *b* being the output label of an arc emanating from a module, and a*consumption event* has *b* as the label of an input arc to a module. If (b v) is an event, we refer to the value part v as a b-value.

A *history* of a module is a sequence of events which can occur in one possible use of it. For example, one history for the Add_{bcd} module, as described above, might be,

(b 1) (c 3) (d 4) (c 5) (b 2) (d 7)

An *input history* is the projection of a history on the input arcs, i.e. any event labeled with an output arc is discarded. An *output history* is defined similarly.

Determinate modules, such as the Add module above, have the property that for a given input history, the output history is determined. That is, even though there may be several histories corresponding to the given input history, each of those histories corresponds to the same output history. It is worth noting that all of the different queue modules described in the previous section have the same functional behavior, namely the identity function. Thus, functional behavior alone does not act as the finest discriminator of actual behavior, and this is a clue to its limitations.

As another example, a Merge module passes its input values unchanged, rather than adding them, but passes them in consupmtion order. The following is one history for a $Merge_{bcd}$ with the same inputs b, c as above:

(b 1) (d 1) (c 3) (d 3) (c 5) (d 5) (b 2) (d 2)

However, assuming there could be some delay between the time the module consumes an input token and acts upon its value, such as if it contained a buffer at each input, another history might be

(b 1) (c 3) (d 3) (d 1) (c 5) (b 2) (d 5) (d 2)

We notice that both the histories have the same corresponding input history, but different output histories, thus merge is seen to be *indeterminate*.

The set of all possible histories is called the *archive* for the module, and characterizes the module, i.e. forms a denotation for it. Certain features of potential interest, in particular the "internal state" of the node, are suppressed in this description of the behavior of a module. For the purposes of the present discussion *we assume that the history characterizes all observable properties of interest*. This notion of observability is admittedly weaker than some others that have been used in the CCS framework [Hennessey 82].

Notation: If S is a set of sequences, then S^* designates the set of all concatenations of a finite number of sequences in S, S^∞ designates the set of all concatenations an infinite number of sequences in S, and $S^\sim$ designates the union of these two. For example, the archive of the fork_{bcd} module can be expressed as

$$\{ (b\ v)\ (c\ v)\ (d\ v) \mid v \varepsilon V\}^\sim$$

which indicates that each consumption of a value v (on arc b) is followed by a production of the same value v on arc c and on arc d, in that order. Although we normally think of the output production of the fork as being simultaneous, it simplifies the expression of the archive to specify an order, as we have done. Since modules are normally combined with others which represent delay on interconnection lines, this ordering will usually become irrelevant in a network.

To describe archives of other modules, some additional notation will be useful. For history x and arc b, $\Pi_b(x)$ is the sub-sequence of b-events in x, and $\Pi_{b=v}(x)$ is the sub-sequence of b-events with value v. Similarly, for a *set* B of arcs, $\Pi_B(x)$ is the sub-sequence of b-events where b is any element of B. Furthermore, $V_b(x)$ is the sequence of b-values in x, in the order they appear, while $\#_b(x)$ is length of that sequence, i.e. the number of b-events in x. Finally, $K_{bc}(x)$ is the sequence of events in x which are neither b-events nor c-events. We use $\leq$ to designate the *prefix* ordering on sequences, as well as the usual integer ordering.

Some relations recur sufficiently often that we give them special names:

equal(b, c, x)

means that $V_b(x) = V_c(x)$, i.e. that the sub-sequence of b and c values in x are the same;

precedes(b, c, x)

means that each c-event in x is preceded by a corresponding b-event, i.e. for every prefix $y \leq x$, we have $\#_b(y) \geq \#_c(y)$. This is used to reflect causal ordering between b and c events, as is required in interconnections to be described. We also define

leads(b, c, x)

to mean precedes(b, c, x) together with the condition that for every prefix $y \leq x$, $\#_b(y) \leq 1 + \#_c(y)$. In other words, each b-token must be consumed as a c-token before another b-token can be produced.

The *leads* and *precedes* relations can be used to capture essential causal relationships within histories. For example, if each output event on arc c of a module is produced in response to an input event on arc b, we will have precedes(b, c, x). This is also true if each input event on an arc c is produced on output arc b of a node. Note that if we begin with a sequence having the property precedes(b, c, x), and insert events other than b- and c- events in arbitrary positions, we shall always have a sequence y such that precedes(b, c, y).

Examples

The Unbounded-Queue module, UBQ_{bc}, with input arc b, output arc c, and value set V, has the archive

$$\{x \in \{ (b\ v), (c\ v) \mid v \in V\}^{\sim} \mid \text{equal}(b, c, x) \text{ and precedes}(b, c, x) \}$$

The comma between (b v) and (c v) means that we are collecting as *separate sequences* the values (b v) and (c v), then concatenating the result.

A 1-bounded queue module, with input arc b and output c, has the archive

$$\{ (b\ v)\ (c\ v) \mid v \in V\}^{\sim}$$

Note that equal(b, c, x) and precedes(b, c, x) automatically hold for each history in the latter.

3. Network Composition

We now discuss the syntax and semantics of network composition. Our discussion centers around rules for composing the archives for modules to obtain the archives of the resulting networks. Similar in spirt to [Patil 70], we wish to have the following:

Composition property: when several modules are interconnected, the resulting network itself can be characterized as module.

We consider three types of interconnections between sub-networks, which are evidently sufficient to construct any network: *aggregation*, *serial composition,* and *loop composition*. Of these, the second is a convenience which can be composed from the other two [Keller 83].

Aggregation: Here we simply "lump" modules M and N together without connecting them, as shown in Figure 4, designating the result by A(M,N).

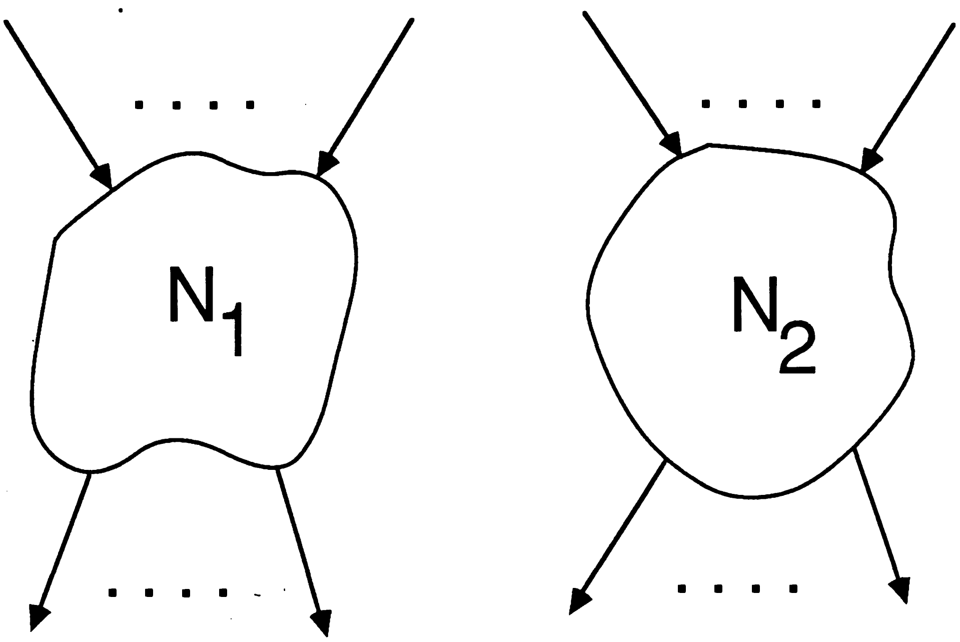

Figure 4: Aggregation of networks

The archive-composition rule corresponding to aggregation is the "shuffle". To describe the shuffle (designated Δ) more precisely, assume initially that x and y are event sequences over 2 *disjoint* sets of arc labels, B and C. Then

$$x \Delta y = \{z \mid \Pi_B(z) = x \text{ and } \Pi_C(z) = y\}$$

That is, the shuffle of two sequences on disjoint sets of arcs is a set of sequences, such that projecting on the respective sets of arcs gives back the original sequences. We extend this in the natural pointwise fashion to *sets*,

$$S \Delta T = \cup \{x \Delta y \mid x \varepsilon S, y \varepsilon T\}$$

In case that the label sets B,C are *not disjoint*, the definition of shuffle is slightly more complicated. The simplest definition seems to be to force them to be disjoint by *renaming* one set of labels, shuffling as before, then performing the inverse renaming to get the result.

Example: To shuffle two events sequences:

(b 1) (d 1) (b 2) (d 2)

with

(c 3) (d 3) (c 4) (d 4)

we have to rename one of the d's, say that in the second sequence, to d'. Shuffling then gives us, among others, the sequence

(b 1) (d 1) (c 3) (d' 3) (c 4) (b 2) (d' 4) (d 2)

and inverse renaming changes each d' to d to get

(b 1) (d 1) (c 3) (d 3) (c 4) (b 2) (d 4) (d 2)

We can summarize the preceding discussion as a rule for constructing the archive of an aggregation from the archives of the component networks. If N is a network, with labels for its unconnected arcs, then let Archive(N) denotes the corresponding archive.

Aggregation Rule: Archive(A(M,N)) = Archive(M) Δ Archive(N), where Δ is the shuffle operator.

The next type of interconnection rule is *serial composition*, shown in Figure 5, which we designate by $S_{bc}(N_1,N_2)$, b being an output arc of N_1 and c being an input arc of N_2. We assume here that arcs of N_1 and N_2 have *disjoint* sets of labels. Otherwise we rename the labels first to get this property. The archive semantics of serial composition will be easier to present after we present the loop interconnection next.

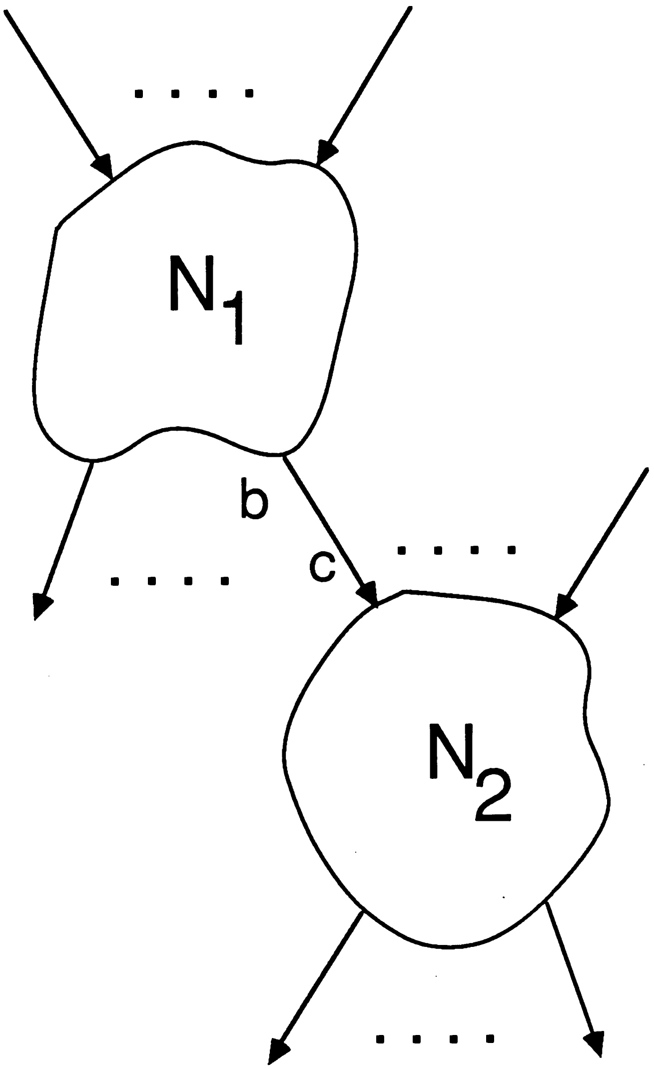

Figure 5: Serial composition

We also need the ability to construct a loop, as shown in Figure 6, designating the resulting network as $L_{bc}(M)$. It is now clear that connection S can be realized as an aggregation followed by a loop, so if we are treating archive operators for these, we need not treat S separately.

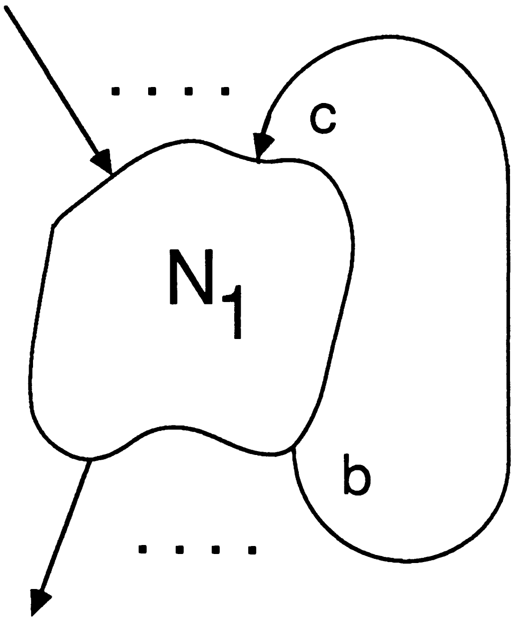

Figure 6: Network composition with a loop

To describe the effect of the *loop* construct L on an archive, we need to introduce a new *matching operator* M_{bc} where, as usual, b and c are arc names.

$$M_{bc}(S) = \{ K_{bc}(x) \mid x \varepsilon S \text{ and } leads(b,c,x)\}.$$

Here $K_{bc}(x)$ discards events on arcs b and c, and preserves others. The purposeof the *leads* condition is to express causality; If b values are fed to c, then in each history, each c-event is preceded somewhere by a b-event with the same value, and the number of b-events cannot be more than one plus the number of c-events for any prefix in the history. This observation is summarized as:

Loop Rule: $\text{Archive}(L_{bc}(N)) = M_{bc}(\text{Archive}(N))$

It is possible to formulate other loop composition rules. For example, if we replaced *leads* with *precedes* we would have a rule for the case where each the loop-closing arc has an implied unbounded queue.

According to earlier discussion, we also have the

Serial Composition Rule:

$$\text{Archive}(S_{bc}(N_1, N_2)) = M_{bc}(\text{Archive}(N_1)\ \Delta\ \text{Archive}(N_2)).$$

The combination $M_{bc}(S\Delta T)$ occurs sufficiently often that it deserves a special notation,

$$S\Delta_{bc}T$$

This matching shuffle operator Δ_{bc} is similar to the operators used by Riddle [Riddle 79] to manipulate event sequences. In his work, special synchronizing symbols are introduced into the event sequences, and shuffling is performed on substrings occurring between matched synchronizing symbols.

Before further illustrating reasoning about archives using the above composition rules, it will be useful to express a few observations about causality, expressed using the *precedes* primitive:

Causality Rules:

Prefix (precedes(b,c,x) and $y \leq x$) implies precedes(b,c,y).

Transitivity (precedes(b,c,x) and precedes (c,d,x)) implies precedes(b,d,x).

Antisymmetry If precedes(b,c,x) and precedes(c,b,x), then x contains neither b nor c events.

Here are examples of reasoning about archives using these properties.

Example Consider the network of Figure 3, assuming that F is 1-bounded queue module. We have

$$\text{Archive}(F_{ef}) = \{\ (e\ v)\ (f\ v)\ |\ v \in V\}^{\sim}$$

$$\text{Archive(fork}_{bcd}) = \{ (b\ v)\ (c\ v)\ (d\ v) \mid v\ \varepsilon\ V\}^{\sim}$$

The archive for the entire network is, according to our rules,

$$L_{de}(\Delta_{fb}(\text{Archive}(F_{ef}),\ \text{Archive(fork}_{bcd}))).$$

We first examine the inner expression, $\Delta_{fb}(\text{Archive}(F_{ef}),\ \text{Archive(fork}_{bcd}))$. It is just $M_{fb}(\Delta(\ \text{Archive}(F_{ef}),\ \text{Archive(fork}_{bcd}))$, which can be computed to be

$$\{ (e\ v)\ (c\ v)\ (d\ v) \mid v\ \varepsilon\ V\}^{\sim}$$

In other words, the 1-bounded queue module adds nothing in this case. Now we apply L_{de} to this archive to represent the loop. This adds to the above set the additional constraint that leads(d, e, x) for each history x. But leads(d, e, x) implies precedes(d, e, x), and by the property of antisymmetry, we see that no history in the overall network can contain a c-event or d-event. But the empty sequence Λ is the only remaining member of the above set after this condition is imposed, so we see that the overall archive is $\{\Lambda\}$. Note that, since the modules in this case are determinate, the fixed-point theory of Kahn could have arrived at this result by starting with the null sequence and applying the fork function, which would give the null sequence back, thereby determining the least fixed-point.

Example Instead of choosing F above to be a 1-bounded queue, choose it to be 1-fby_{ef}, which produces an output value of 1, followed by a verbatim copy of its input. The archive of 1-fby_{ef}, with input b and output c, is given by

$$\{(f\ 1)\}\ \{ (e\ v)\ (f\ v) \mid v\ \varepsilon\ V\}^{\sim}$$

where juxtaposition of sets indicates the set of all sequences formed by concatenating a member of the first with the member of the second.

When we compose 1-fby_{ef} with fork_{bcd}, we get the archive

$$\{(c\ 1)\ (d\ 1)\}\ \{ (e\ v)\ (c\ v)\ (d\ v) \mid v\ \varepsilon\ V\}^{\sim}$$

When we "close the loop", as shown, we are requiring, in each history x, equal(d, e, x). The only history in the above set having this property is

$$(c\ 1)\ (d\ 1)\ (\ (e\ 1)\ (c\ 1)\ (d\ 1)\ )^{\infty}$$

and this history does not violate the additional property leads(d,e,x). The overall archive, after masking d- and e- events, is just

$$\{(c\ 1)^{\infty}\}$$

Example The archive for a Merge_{bcd} module, with 1-queuing on inputs, as discussed aboves simply the shuffle of archives for two 1-bounded queue modules, with their output lines both identified as d, i.e.

$$\{\ (b\ v)\ (d\ v)\ |\ v\ \varepsilon\ V\}^{\sim}\ \Delta\ \{\ (c\ v)\ (d\ v)\ |\ v\ \varepsilon\ V\}^{\sim}$$

We illustrate on the "merge anomaly" network [Keller 78], as shown in Figure 7. and show that there are no causal anomalies in any of the histories of the archive. Here the module G functions so as to map successive input values to output, in such a way that if the token value is 1, it outputs 3, otherwise outputs the value itself.

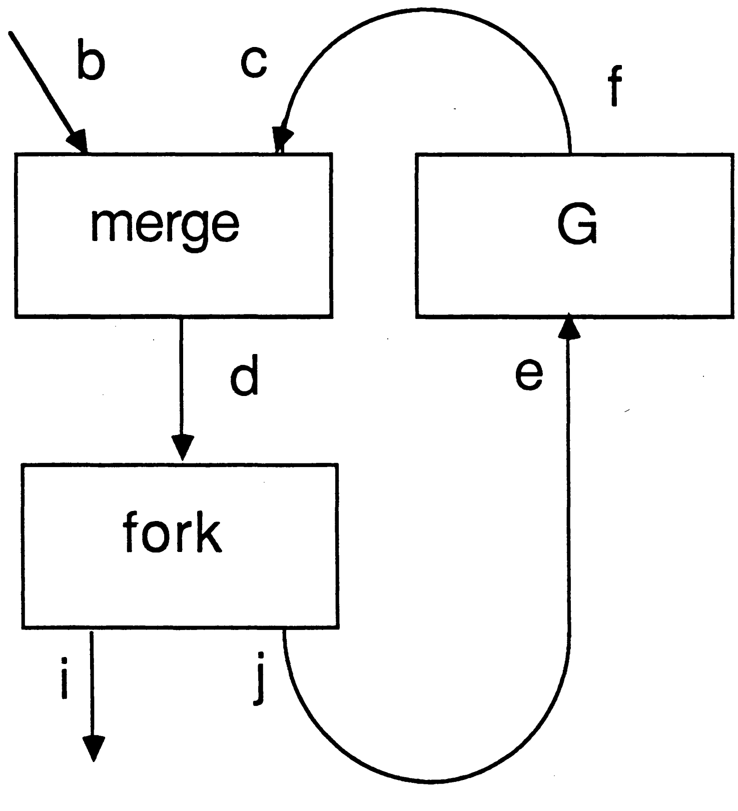

Figure 7: The Merge Anomaly Network

Thus the archive of G_{ef} is

$$\{(e\ 1)\ (f\ 3),\ (e\ v)\ (f\ v) \mid v \in V - \{3\}\}^{\sim}$$

When composed with $Merge_{bcd}$ and $Fork_{dij}$, but before closing the f-c loop, we get the archive

$$\{(b\ 1)\ (i\ 1)\ (f\ 3),\ (b\ v)\ (i\ v)\ (f\ v)\}^{\sim}\ \Delta\ \{\ (c\ 1)\ (i\ 1)\ (f\ 3),\ (c\ v)\ (i\ v)\ (f\ v)\}^{\sim}$$

where, as above, v represents a generic element other than 3. When we close the loop, we effectively apply the L_{fc} operator to the expression above. It is difficult to produce a closed form for the result, but we can resort to reasoning about specific input histories. For example, if V = {1, 2, 3} and the input history is

$$(b\ 1)\ (b\ 2)$$

then the overall histories are derivable from archive reasoning as being of the form

$$(i\ 1)\ \ (i\ 3)^n\ (i\ 2\ )\ ((i\ 3)\ (i\ 2))^{\infty},\ n \text{ arbitrary}$$

The matching requirement for the f-c loop forces the first f-event to precede the first c-event. This in turn means that the value of the first d-event must appear in response to a b-event. Since we are assuming that the first b-value is 1, it follows that the first d-value is 1. Secondly we observe that, by the definition of G, there are no c-values equal to 1. From the input and the archive for merge, we conclude for any history h that

$$V_d(h) = 1\ x\ 2\ y, \text{ where } V_c(h) = x\ y.$$

From the definition of G, we must then have y = 3 x 2 y. Closing the loop implies $V_f(h) = V_c(h)$, in other words x y = 3 x 2 y. From this, we conclude that x is of the form 3^n for some n. If we cancel 3^n from both sides, we get y = 3 2 y which has only the solution y = $(3\ 2)^{\infty}$. Putting these observations together gives us the indicated form of the history for this input.

Example Our final example of archive composition demonstrates how archives differentiate the networks of Brock and Ackerman, as shown in Figure 8. Here the modules are defined to have the archives

$D_{bc} = \{\Lambda,\ (b\ v)(c\ v)(c\ v) \mid v \in V\}$, $D_{de} = \{\Lambda,\ (d\ v)(e\ v)(e\ v) \mid v \in V\}$

$P_1 = \{\Lambda,\ (f\ u)(f\ v)(g\ u)(g\ v)\ x \mid u, v \in V,\ x \text{ an arbitrary sequence of } (f\ v) \text{ pairs}\}$

$P_2 = P_1 \cup \{(f\ u)(g\ u),\ (f\ u)(g\ u)(f\ v)(g\ v)\ x \mid u, v \in V,\ x$ an arbitrary sequence of $(f\ v)$ pairs$\}$

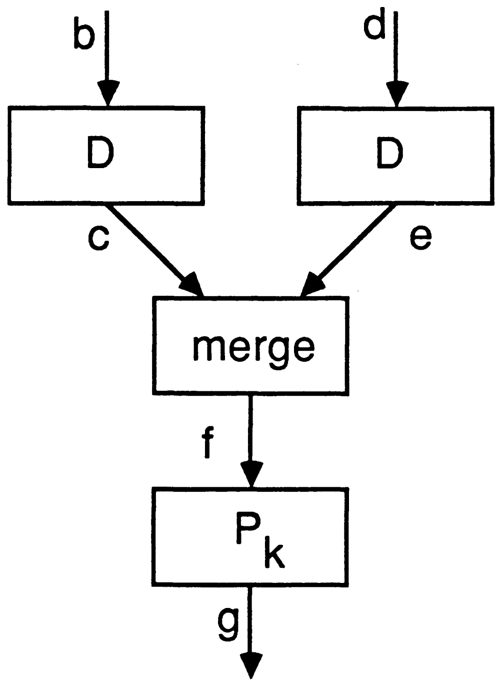

Figure 8: The networks (k = 1,2) of Brock and Ackerman

In other words, a D module replicates its first input. Whereas P_2 can produce its first output token as soon as it receives its first input, P_1 must wait for two input tokens before it can produce any output. The composition of the two D modules with the merge is easily seen to give the archive

$$\{\Lambda, (b\ v)(f\ v)(f\ v) \mid v \in V\}\ \Delta\ \{\Lambda, (d\ v)(f\ v)(f\ v) \mid v \in V\}$$

The possible outputs (on arc f) of the merge are thus characterized by the set

$$\cup \{ \{\Lambda, u, v\} \cup (\{u\ u\}\ \Delta\ \{v\ v\}) \mid u, v \in V\}$$

Let N_k designate the network containing P_k. The difference of combining with P_1 vs. P_2 shows up as $(f\ v)(g\ v)$ being in the archive for N_2 not in that for N_1. As pointed out by Brock and Ackerman, the two networks can be differentiated for a *fixed* input history (b u) (d v). In this case, the outputs of the merge are just those in the set $\{u\ u\}\ \Delta\ \{v\ v\}$. To see the difference, note that N_2 has in its archive the history

$$(b\ u)\ (g\ u)\ (d\ v)\ (g\ v)$$

whereas N_1 does not. The intuitive explanation here is that N_1, should it produce u before consuming v, cannot then ever produce v; since the component P_1 had to have consumed two inputs before producing any output, it will never consume v, and thus never produce v.

The fact that archives are represented as sets of sequences makes it simple to program analysis tools for them, at least for the case of finite histories. We have programmed some simple archive manipulating functions, which were able to mechanically differentiate between the two networks above. We believe these tools would be much more complicated to build if we were to base them on partial orders of events rather than sequences [Keller 78, Brock and Ackerman 81, Pratt 82], but on the other hand, the computational complexity of the tools can probably be improved by switching to set of partial orders, or at least sets of trees. This issue of tool building seems to be a relatively unexplored topic.

4. Extension Rules

In this section, we present a complementary approach to for deriving archives, which corresponds more closely to the traditional [Scott 70] approach to the denotational semantics of sequential programming languages. The key feature of this approach is that the denotations of programming constructs are viewed as functions and the denotations of complex constructs are obtained by ordinary functional composition.

We shall view the denotations of modules as functions which act on archives. The actions of these functions are the pointwise extension of their actions on individual histories. These functions act to *extend* the history by inserting new events which correspond to the participation of that module in the history. Thus these functions are called *extension rules*.

Examples The extension rule for a 1-bounded queue, with input b and output c, maps an input sequence of the form

(b 1) (b 2) (b 3) ...

into a set of one history

{ (b 1) (c 1) (b 2) (c 2) (b 3) (c 3) ... }

whereas for an unbounded queue, it would map into an infinite set of histories

$$\{ x \in (\{ (b\ 1)\ (b\ 2)\ (b\ 3) \ldots \} \Delta \{ (c\ 1)\ (c\ 2)\ (c\ 3) \ldots \}) \mid \text{precedes}(b, c, x) \}$$

It is then fairly clear that the extension rule for an acyclic network can be constructed by composing the extension rules of the component modules. Networks which contain loops have extension rules obtained by taking fixed points of an appropriate composition of the extension rules of the component modules (*cf.* [Keller 78]). The sense in which a fixed point can be constructed will now be clarified. We do not use traditional complete partial order theory but give a construction that is similar to the category theoretic notion of a limit. The use of extension rules will mimic the archive set-theoretic denotational semantics of the preceding sections quite closely.

Definition Let F be an module and let its archive be A . Let I_F be the input archive, i.e. the set of all possible input histories to F. I_F consists of all sequences of all possible events on the input arcs to F. The *extension rule* for F, written E_F, is a function from archives into archives which satisfies:

$$E_F(I_F) = A_F.$$

Thus, if I_F is the input archive, then $\Pi_I(E_F(I_F)) = I_F$, where Π_I represents the projection onto the input arcs of F.

An extension rule is required to satisfy several "reasonability" conditions. These conditions will turn out to be essential in constructing extension rules for networks containing loops. To express these conditions we use the following notation:

E will stand for a generic extension rule,

h will stand for a generic history, and

$h_1 \leq h_2$ means that h_2 extends h_1 in the sense described above (but not necessarily in the prefix sense).

We will also need the following terms:

Definition: An event v, in a history h, is said to be *consumable* by an extension rule E, if it is a production event that appears on the input arc of an module in the network described by E and there is a history in the archive which contains extends h, as well as containing a consumption event corresponding to v.

Definition: An event v, is said to be *producible* in a history h, by an extension rule E, if there is a history in the archive of the network which extends h and contains v.

We may use the word "enabled" to mean either consumable or producible, and "fired" to represent the occurrence of the corresponding production or consumption. The conditions on extension rules e can now be stated:

Monotonicity: The action of E cannot remove an event from a history; in symbols, if $h' \varepsilon E(h)$ then $h \leq h'$.

Causality: An extension rule can only insert an consumption event after the corresponding production event, and a production event can be inserted only after the consumption events which triggered that production event.

Stability: If a history h has no enabled events, then $E(h) = \{h\}$.

Finite Delay: Within a chain of histories $h_1, h_2, \ldots, h_i, \ldots$ where for each i, $h_{i+1} \varepsilon E(h_i)$, no event remains enabled but not fired forever.

Note that monotonicity is not the same as that for the *prefix* partial order [Kahn 74], since embedding is not a partial order. A simple example is provided by the two sequences $(ab)^{\infty}$ and $(aab)^{\infty}$. The anti-symmetric property does not hold, since these sequences are unequal, yet each may be embedded in the other, by adding a's or b's.

The finite-delay condition is similar to that of [Karp 69], which is used to show that a suitable notion of taking "limits" of sequences can be defined. It is actually a condition on such limits, rather than a constraint on the presentation of the extension rule.

The extension rule for a network with a loop is easily expressed via a system of recursive equations involving the extension rules of the individual modules in the network. To show that recursively-defined expressions have a well-defined meaning involves introducing a suitable mathematical structure on the set of archives and showing that limits of sequences of approximations can be defined. Normally, the mathematical structure used is that of a complete partial order. However, for our purposes, this structure is not appropriate. We would like our structure to reflect the intuitive idea that histories improve by the insertion of new events, which are inserted, but not necessarily added at the end of the histories. We have already mentioned that the set of sequences ordered in this way are not partial orders (at least, if infinite sequences are allowed).

With sets of histories the situation is even worse. Even if one does have a complete partial order, subsets of the complete partial order can only be viewed as a complete partial order if one restricts the subsets to be closed in some appropriate sense [Plotkin 76, Smythe 78]. We would like to be able to work with arbitrary sets of histories, since otherwise modules with different operational behaviors would have the same denotation.

The notion of limit that we shall define is similar to the category theory notion of limit. Suppose we have an module F which has an input arc c and an output arc b. Let the extension rule for this module be denoted by E. Now suppose that the arc b is connected to the arc c. The extension rule must be modified to associate every reference to a b-event with the corresponding c-event. Let this modified extension rule be denoted by E_{bc}. Whenever E_{bc} is used to extend a given history h, there will be new input events to F in the extended history $E_{bc}(h)$. The extension rule E_{bc} must therefore be applied again to the result. Given a particular initial history, h, applying the extension rule E_{bc} may result in several histories. Applying E_{bc} to each history in the resulting set of histories will also yield several new histories.

Theorem Under the stated reasonablity assumptions, there exists a limit for the sequence of histories described above.

To establish the exist,ence let us denote the set containing the original input history, h, by S_0, and the sets containing the subsequent extensions by S_1 S_2 ... respectively. As we construct each subsequent extension, we define relations, written R_i, which express which histories in S_i were extended by E to yield particular histories in S_{i+1}. Formally, if h_i R_i h_{i+1}, then $h_i \varepsilon S_i$, $h_{i+1} \varepsilon S_{i+1}$, and $h_{i+1} \varepsilon E_{bc}(h_i)$. We refer to such a sequence of sets and relations as the *tower* of E_{bc} over h. The limit of successive iterations of the extension rule E_{bc} applied to h can be defined in terms the tower of E_{bc} over h. The tower defines a set of sequences of histories through the relations R_i. Consider a typical such sequence, h_1 h_2 ..., where h_i R_i $h_{i+1}$for each i in N. Such sequences of histories satisfy the following property:

Lemma There exists a function f: N --> N (N the natural numbers), such that, for every k in N, prefix(k,$h_{f(k)}$) contains no consumable events or enabled events. Furthermore this f can be chosen to be monotonic. Here prefix(k, x) is the length k prefix of x.
Proof: Suppose h contains no consumable events or enabled events. Then by the stability condition, E_{bc} will leave h unchanged and no history in the tower will have either an consumable event or an enabled event. Suppose to the contrary that h does contain an consumable event or enabled event. Then there must be a unique first such event. By the finite-delay property, at some finite point in the tower, this event must be consumed or produced. Suppose this consumption or production occurs in step t of the tower and the event that was consumed or produced is the n*th* event in this history. Using causality, we can define f(n) = t. Because an extension rule can only insert events after their cause,we see that f is indeed monotonic.

Using this lemma, it is possible to construct the following sequence of embeddings:

$$\text{prefix}(1,h_{f(1)}) \leq \text{prefix}(2, h_{f(2)}) \leq \dots \leq \text{prefix}(i, h_{f(i)}) \leq \dots$$

However, these embeddings are additionally prefixes of one another, so the limit of this sequence does exist and is unique. We denote this limit sequence by h^*. By construction, none of the members of the sequence has any consumable or enabled events, so neither does h^*. Thus $e(h^*) = \{h^*\}$ by the stability condition. The limit of the entire tower is obtained by constructing the limit in the above fashion for every sequence of histories in the tower. Let us denote by $E^*_{bc}(h)$ the limit of the tower of E_{bc} over h. Then E^*_{bc} defines the extension rule for the network with b connected back into c.

The asymmetry between the roles of b and c noted in Section 3 is implicitly present here. The new input events present in the N*th* iteration of the extension rule were generated in the (N-1)*th* iteration. Thus the b-events have two roles: as output events of the (N-1)*th* iteration and as input events in the N*th*. The effect would have been the same as if we had used the original extension rule E with the change that whenever a c-event is inserted, an b-event is also inserted immediately following the c-event. This view of the extension rule shows that it must necessarily construct histories of the form that appear in $L_{bc}(F)$.

Extension rules potentially provide a more tractable computation scheme than do archives, since they can be applied to a given input archive, rather than having to deal with all possible input archives. Some examples of computable extension rules are discussed in an earlier version of this paper [Keller 84].

5. Conclusions

We have described an approach for modeling digital networks containing possibly-indeterminate modules, and given rules for determining denotations for interconnected networks based upon denotations for the constituents. The denotational approach we have outlined in this paper shares some features commonly associated with operational semantics. In particular, causality constraints are explicitly stated in the definitions of most archives. More importantly, the basic concept on which the theory is built is the computational event, which is a very operational concept. This closeness to operational ideas makes this formalism a convenient and flexible tool for modelling systems of practical interest, while we retain the advantages of a denotational approach.

Our approach is similar to a variety of other semantic theories for networks containing indeterminate modules. These go under the general name of "trace" domains: a domain is built from sequences of computational actions. As far as we are aware, our theory is the only one using the category theoretic style of limit. Abstract semantics have been given by [Plotkin 76], [Broy 81], and [Abramsky 83]. The Plotkin construction does not handle unbounded indeterminacy or non-flat domains. Our construction does allow us to discuss unbounded indeterminacy, manifest by the merge module, for example. The Broy construction does not allow one to describe arbitrary sets of possible results, as the sets in his domain constructions are required to satisfy a suitable closure condition. Important recent work on semantics of indeterminate constructs is that of Abramsky 83. His formalism is explicitly category theoretic; using multi-sets of possible results, he can handle arbitrary sets, unbounded indeterminacy, and power constructions over domains which are not flat.

6. References

[Abramsky 83] Abramsky, S. Semantic Foundations of Applicative Multiprogramming. In Diaz, J. (editor), *Automata, Languages and Programming*, pages 1-14. Springer-Verlag, July, 1983.

[Brock 81] Brock J.D.,W.B.Ackermann. Scenarios:A Model of Non-Determinate Computation. In J.Diaz, I.Ramos (editor), *Formalization of Programming Concepts*, Lecture notes in computer science 107, pages 252-259. Springer-Verlag, New York, 1981.

[Broy 81] Broy M. A Fixed Point Approach to Applicative Multiprogramming. In Lectures at the International Summer School on Theoretical Foundations of Programming Methodology. July, 1981.

[Hennessey 82] Hennessey, M. Synchronous and Asynchronous Experiments on Processes. Technical Report, University of Edinburgh, September, 1982.

[Jhon 85] Jhon, C.S., G.E. Sobelman, and D.E. Krekelberg. Silicon compilation based on a data-flow paradigm. IEEE circuits and devices magazine, May 1985, 21-28.

[Kahn 74] G. Kahn. The semantics of a simple language for parallel programming. in Information Processing 74, pages 471-475. IFIPS, North Holland, 1974.

[Karp 69] Karp, R. M. and R. Miller. Parallel Program Schemata. JCSS 3(2):147-195, May, 1969.

[Keller 78] Keller R.M. Denotational Models for Parallel Programs with Indeterminate Operators. In E.J.Neuhold (editor), *Formal Descriptions of Programming Concepts*, pages 337-365. North-Holland, Amsterdam, 1978.

[Keller 80] Keller, R.M. G. Lindstrom, and S. Patil. Data-flow concepts for hardware design. Digest of papers, IEEE Compcon '80, pages 105-111, Feb., 1980.

[Keller 83] Keller, R.M. Unpublished presentation on archives. Applicative Languages Workshop, Massachusetts Institute Technology, Endicott House, June, 1983.

[Keller 84] Keller, R.M. and P. Panangaden. Semantics of networks containing indeterminate operators. In S.D. Brookes, A.W. Roscoe, and G. Winskel (eds)., Seminar on Concurrency, Lecture notes in computer science 197, 479-496, July 1984.

[Patil 70] Patil, S.S. Closure properties of interconnections of determinate systems. Record of the Project MAC conference on concurrent systems and parallel computation, 107-116, June 1970.

[Plotkin 76] Plotkin G. A Powerdomain Construction. SIAM J. of Computing 5(3), September, 1976.

[Pratt 82] Pratt V. On the Composition of Processes. In Ninth Annual ACM Symposium on Principles of Programming languages, pages 213-223. ACM, January, 1982.

[Riddle 79] Riddle W.E. An Approach to Software System Behavior Description. Computer Languages 4:29-47, 1979.

[Scott 70] Scott D.S. Outline of a Mathematical Theory of Computation. In Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, pages 169-176. 1970.

[Scott 71] Scott D.S., Strachey C. Towards a Mathematical Semantics for Computer Languages. Technical Report PRG-6, University of Oxford, 1971.

[Smythe 78] Smythe, M.B. Power Domains. J. CSS 16:23-36, 1978.