A Proof System for Dataflow Networks with Indeterminate Modules

Abha Moitra
Prakash Panangaden

TR 86-782
September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

A Proof System for Dataflow Networks with Indeterminate Modules

Abha Moitra
Prakash Panangaden
Computer Science Department, Cornell University

Abstract

In this paper we discuss a model for dataflow networks containing indeterminate operators and the associated proof system. The model is denotational and associates with each network the set of possible behaviors. The possible behaviors are represented by traces. The novel feature of our proof system is that we give an inductive proof rule for recursively defined networks based on a fixed point construction given by Keller and Panangaden. We show soundness and relative completeness of our proof system.

1 Introduction

In this paper we discuss a model for dataflow networks containing indeterminate operators and the associated proof system. The model that we present is due to Keller and Panangaden [8,7,11]. The model is denotational and associates with each network the set of possible behaviors. A possible behavior of the network is represented by the sequence of possible events that occur in a particular execution. Precedence of events in the sequence is interpreted as temporal precedence. This kind of model was suggested by Keller in [6] and was developed further by Brock and Ackerman [2] and by Pratt [12]. The novel feature of the model developed by Keller and Panangaden [7] is that a fixed–point construction is given. The model of Brock and Ackerman does not describe how one can combine infinite behaviors. Subsequent proof systems and models, for example that of Nguyen, Gries and Owicki [10] present an infinitary proof rule for networks with loops in them. We present an inductive proof rule based on the fixed–point construction.

It has long been felt that a fixed–point construction was impossible for fair networks containing indeterminate operators because fairness is not "continuous". Indeed the fixed–point construction that we give does not use standard domain theoretic techniques but instead uses a category theoretic limit construction. The use of category theoretic methods was first proposed by Lehmann [9]. Subsequently Abramsky gave a denotational semantics for unbounded indeterminacy based on category theory [1]. The inductive proof rule that we obtain is somewhat complicated since it has to express the fairly complicated limit construction rather than the somewhat simpler least upper bound construction of domain theory. We present a couple of examples of how inductive proof methods can be used to reason about networks. We also prove the soundness and relative completeness of our proof system.

2 The Model

In this section we introduce the semantics of networks containing indeterminate modules. The basic scenario is as follows. The networks we consider are dataflow networks. They consist of autonomous computing agents connected by unidirectional data arcs along which the data tokens flow. The data arcs, or *channels* as we shall refer to them hereafter, are named and can be referred to explicitly by name. These channels provide the only means for the computing agents to communicate with each other. This situation was discussed originally by Kahn [4] for the case that all the computing agents are executing programs that are functional in their input histories. The extension to indeterminacy was carried out by several authors including Brock and Ackerman [2], Pratt [12,13] and Keller and Panangaden [8,7]. The material in this section is a summary of [7]. For a detailed discussion see [11].

The notion of behavior that we use is that of a *trace*. A trace represents a possible computation of a system and consists of a sequence of observable events. The basic building blocks of traces are *events*. An event is a triple consisting of a tag (either + or −), a channel (or arc) name and a data token. We write an event in the form $< +, a, 3 >$ to indicate that the token "3" was *produced* on channel a. The + tag is used to indicate that the event corresponds to the production of the indicated token whereas the − tag is used to indicate that the corresponding token has been consumed from the indicated channel. This distinction between production events and consumption events allows the model to have roughly the discriminating power of the failures model []. Since consumption events and production events are not required to be tightly synchronized in traces, the semantic model describes asynchronous communication between networks. In the model the denotation of a system is precisely the set of traces corresponding to all possible computations that could have been performed by the system. Specific operations to compose trace sets need to be defined so as to obtain a compositional semantic theory. This we now proceed to do.

The following notation is used to capture basic aspects of traces:

- $\Pi_b(x)$ represents the sequence obtained by projecting the trace x on the channel b. In other words, all but the b-events are discarded.

- $\Pi_B(x)$ represents the sequence obtained by projecting the trace x onto the set of channels B.
- $\Pi_b^+(x)$ represents the sequence obtained by projecting all the production events on channel b. Similarly we also use Π_b^- as well as analogous notation for sets of channels.
- $\Pi_{b=v}(x)$ represents the subsequence of x obtained by discarding all but the b-events with value v.
- $||x||$ is the length (number of events) of the sequence x. If the sequence has infinite length then its length is undefined.
- $\backslash_a(x)$ is dual to $\Pi_a(x)$, i.e. it represents the sequence of events from x with the a-events removed.
- $V_b(x)$ is the sequence of *values* corresponding to b-events, in other words it consists of $\Pi_b(x)$ with the first two components of each event triple removed. As with the Π notation, this can be modified by $+$ or $-$ superscripts.

When we wish to display an event as part of a sequence of events we use the notation

```
+ -
b b
y y
```

with the sequential order of the events representing the order in which they actually occur. In the above example, the token y appears on the channel b first and then the token is consumed from the same channel. When we wish to display an event as part of a sequence of events and we do not wish to explicitly distinguish the appearance and consumption events, we will often use the following notation:

```
b
y.
```

A *trace* of a node is a *sequence* of events that can occur in *one* possible execution. For example, one trace for a module that produces, on channel d, sums of pairs of values consumed on channels b and c, might be

```
+ + + - + - + + - - + - + - +
b c b c c b d b b c d b c c d
1 2 3 2 4 1 3 5 3 4 7 5 4 4 9
```

For another example, this time one that exhibits indeterminate behavior, consider a *merge* module that passes its input values unchanged, rather than adding them, but passes them in some arbitrarily-interleaved order. The following are two possible traces for a merge:

```
+ - + + - + + + - + + - + - + + - +
b b d c c d c b c d c c d b d c c d
1 1 1 3 3 3 2 6 2 2 7 7 7 6 6 4 4 4

+ + - + - + + - + + + - + - + + - +
b c b d c d c c d b c b d c d c c d
1 3 1 1 3 3 2 2 2 6 7 6 6 7 7 4 4 4
```

The description of a module can always be given by placing it in the setting of a node (which provides labeled channels with which to describe it), and presenting traces using the labels. The set of all possible traces is called the *archive* for the module, and characterizes the module, i.e. forms a *denotation* for it.

We use $\sqsubseteq$ to designate the *prefix ordering* on sequences. Some relations recur sufficiently often that we give them special names: $\boldsymbol{equal}(b, c, x)$ means that $V_b(x) = V_c(x)$, i.e. that the sub-sequence of b and c values in x are the same. $\boldsymbol{precedes}(b, c, x)$ means that each c-event in x is preceded by a corresponding b-event, i.e. $\forall y \sqsubseteq x$, we have $\|\Pi_b(y)\| \geq \|\Pi_c(x)\|$. $\boldsymbol{leads}(b, c, x)$ means that each c-event in x is *immediately* preceded by a corresponding b-event. Thus, $\boldsymbol{leads}(b, c, x)$ implies $\boldsymbol{precedes}(b, c, x)$, but not conversely.

The *leads* and *precedes* relations are used to capture essential *causal* relationships within traces. For example, if each output event on channel c of a module is produced in response to an input event on channel b, we will have $\boldsymbol{precedes}(b, c, x)$. If we wish to be explicit about the fact that we are referring to production events on c and assimilation events on b we write this as $\boldsymbol{precedes}(b^-, c^+, x)$. Note that if we begin with a sequence having the property $\boldsymbol{leads}(b, c, x)$, and insert events other than b- and c-

events in arbitrary positions, we shall always have a sequence y such that $precedes(b, c, y)$.

We now use the predicates defined above to give the archive for an unbounded buffer. The buffer is assumed to have an input arc labeled b and an output arc labeled c. The archive is:

$$\mathcal{A}(Buffer) = \{h | precedes(b^+, b^-, h) \wedge equal(b^+, b^-, h) \wedge precedes(b^-, c^+, h) \wedge equal(b^-, c^+, h)\}.$$

A common operator used to combine traces is the shuffle operator. To describe the shuffle (designated Δ) more precisely, assume initially that x and y are event sequences over *disjoint* sets of channel labels, B and C. Then

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

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

$$S \Delta T = \cup_{x \in S, y \in T}(x \Delta y)$$

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

In constructing a network from a set of modules (or sub-networks) we will only permit the connection of an input from one module to the output of another, and for ease of notation an input channel c of one module will be connected to the ouput channel c of another module. So the required interconnections between modules can be achieved by appropriately renaming the channels and we implicitly assume that no two or more distinct input (or output) channels have the same name. We designate the composition of modules $A_1, .., A_n$ by $||_{i=1}^n A_i$, we will also use the infix notation $A||B$ to denote the network composition of networks A and B. If A is a network, with labels for its unconnected channels, then $\mathcal{A}(A)$ stands for the corresponding archive, in other words the denotation of the network A.

To describe the effect of network composition on an archive, we need to introduce a new operator M. First consider the simple network $C = A||B$. $M(t_A, t_B)$ with channel u in A connected to channel u in B is a set of traces and is obtained as follows. Each event e in t_A on channel u is matched with an identical event e' in t_B on channel u. All such matched pairs are identified as single events. Suppose e_1 and e_2 are consecutive u-events in t_A and e'_1 and e'_2 are the consecutive matching u-events in t_B then the set of traces is obtained by identifying the pair (e_1, e'_1) and (e_2, e'_2) and producing all possible shuffles of the event sequence between e_1 and e_2 with the event sequence between e'_1 and e'_2. Analogously when considering the network $C = ||_{i=1}^{n} A_i$, $M(t_{A_1}, .., t_{A_n})$ is obtained by identifying pairwise matched pairs and taking shuffles of the sequences between matched pairs.

3 The Proof System

In this section we shall describe the assertion language which we use in the rest of the paper. The language is a first–order language which abstracts the essential elements of the semantic model given in [7] and summarized in the previous section. Rather than reuse the notation already developed we shall redefine our notation from scratch so as avoid confusion between the semantics and the proof theory.

We will use the full power of temporal formulae in our assertion language. The reason for this being twofold. First of all this allows for more compact specifications of networks. Secondly, and more importantly, we believe that temporal assertions are necessary to ensure the completeness of the proof system which we develop. We should point out that Nguyen, Gries and Owicki [10] also use temporal assertions for a proof system for networks of processes; Chen and Hoare [3] do not use temporal assertions but their proof system is not complete.

We shall specify the assertion language in the usual inductive fashion. The *terms* of the assertion language are:

$$c|e|t|i|A|T_A|\ ||t||\ |val(t[i])|chan(A)|inchan(A)|outchan(A)|\Pi_{chan(A)}(t)$$

There are five sorts. The symbol c stands for a generic channel name. The symbol e stands for an event. The symbols t, i and A stand for traces, integers and networks respectively. $chan(A)$ stands for the set of channels connected to the network A. Similarly, $inchan(A)$ and $outchan(A)$ represent the set of channels which are input or output channels to the network A. The logic supports reasoning about lengths of traces, accordingly, we have the symbol $||t||$ to stand for the length of the trace t. We can look at subsequences of a trace. The symbol $\Pi_{chan(A)}(t)$ stands for the subsequence of the trace t formed by the events on the channels of the network A. T_A is the set of all possible traces for network A. In the semantics this set is written $\mathcal{A}(A)$. The events on a trace can be indexed directly, thus $t[i]$ means the ith event on trace t. We also allow arithmetic expressions. It is the ability to index the individual events on a trace which gives us the expressive power of temporal logic; the modal operators can clearly be expressed in terms of quantifiers.

The formulae of the logic are:

$$true|false|e_1 = e_2|P_1 \wedge P_2|\neg P|$$

$$\exists t \in T_A.P(t)|\forall t \in T_A.P(t)|\Diamond P|\Box P$$

where P_1 and P_2 are themselves formulae of the logic. We allow quantification over traces and over integers but not over channels. In fact some care needs to be taken to forbid implicit quantification over channels. This is a standard restriction on proof systems; it is hard to have any sound rules if quantification over channel names is allowed.

The notation given so far does not describe situations where we connect networks together. We need additional notation to express this. Networks $A_1, .., A_n$ can be combined to obtain a new network C; and this is represented as $C = ||_{i=1}^n A_i$. In such a parallel composition networks $A_1, .., A_n$ are connected together by identifying input channel c of one module with output channel c of another module. This corresponds to the network composition defined in the previous section.

The notation for parallel composition given in the preceding paragraph needs to be defined for sets of traces. In the semantics we used the notation $\mathcal{M}(C)$ to stand for the set of traces for the composite network $C = ||_{i=1}^n A_i$. In the proof system we define the predicate $M_{A_1,..,A_n}(t_1, .., t_n, t)$ to indicate that if t_i is a trace of A_i for $i = 1, .., n$ then t is a trace of the composite network C obtained by matching the traces $t_1, .., t_n$. We shall frequently write simply $M(t_1, .., t_n, t)$ as an abbreviation if there is no danger of confusion.

The *meaning* of the matching predicate can be formally expressed in terms of our semantics as

$$M_{A_1,..,A_n}(t_1, .., t_n, t) \Longleftrightarrow \wedge_{i=1}^n t_i \in \mathcal{A}(A_i) \wedge t \in \mathcal{M}(t_1, .., t_n)$$

Now that we have the language fixed we can define the proof rules. The proof system given here is as powerful as the proof systems of Nguyen, Gries and Owicki [10] and Chen and Hoare [3]. As in [10] we shall present an infinitary proof rule. In a later section we shall introduce an inductive proof rule which is equivalent. The completeness theorem is, however, less messy in terms of the infinitary rule.

We first give an axiom about the lengths of the traces under network composition. For simplicity our axiom will correspond to the case where

in $C = A||B$ only output channel u of A is connected to input channel u of B. The case where multiple channels are connected and the network is composed of a number of modules can be handled in a straightforward but notationally complicated manner and so will not present the general case here. (All the other axioms will however deal with the most general case.)

Axiom 1 (Length)

$$\frac{M(t_A, t_B, t_C)}{\|t_C\| = \|t_A\| + \|t_B\| - \|\Pi^+_u(t_B)\|}$$

where $C = A||B$ *and* $chan(A) \cap chan(B) = \{u\}$ *and* $u \in inchan(B)$

Axiom 2 (Matching)

$$\frac{M(t_{A_1}, .., t_{A_n}, t_C)}{t_C = \Pi_{\cup_{i=1}^n chan(A_i)}(t_C) \wedge (\wedge_{i=1}^n t_{A_i} = \Pi_{chan(A_i)}(t_C))}$$

Axiom 3 (Network Composition)

$$\frac{\wedge_{i=1}^n P_i(t_{A_i}), M(t_{A_1}, .., t_{A_n}, t_C)}{\wedge_{i=1}^n P_i(\Pi_{chan(A_i)}(t_C))}$$

These rules axiomatize the properties of matching. If t_{A_i} is an infinite trace then establishing $M(t_{A_1}, .., t_{A_n}, t_C)$ is clearly equivalent to having infinitely many proof obligations.

Axiom 4 (Renaming)

$$\frac{P(t_A)}{P'(t'_A)}$$

where t'_A is obtained from t_A by changing some channel names with the proviso that all channels in A have distinct names; and P' is obtained from P by substituting all free occurrences of the old channel names by the new channel names. This rule allows renaming of channels.

Axiom 5 (Consequence)

$$\frac{P(t), P \Longrightarrow Q}{Q(t)}$$

In addition to these rules we assume the standard logical axioms. Note that these rules are compositional like the system of Zwiers et. al. [15].

4 Soundness and Completeness

In this section we prove that the proof system presented earlier is both sound and relatively complete. For simplicity we will assume that networks are obtained by combining pairs of modules as opposed to combining them all at once; this allows for simpler notation but does not limit the scope of the theorems and lemmas that we prove.

Theorem 1 *The proof system is sound with respect to the archive model of network behavior.*

Proof: We consider each axiom and show that it is sound. The Length and Matching axioms are sound. This follows immediately from the definition of the matching predicate M. It is easy to see that the Renaming axiom and the Consequence axiom are also sound. The remaining axiom is the Network Composition axiom. This is sound because the matching operator defined in the semantics preserves the traces being matched. More precisely, suppose that $C = A||B$. The matching operator is $\mathcal{M}$. Now suppose that $t \in \mathcal{M}(t_1, t_2)$ where $t_1 \in \mathcal{A}(A)$ and $t_2 \in \mathcal{A}(B)$. Clearly $\Pi_{chan(A)}(t) = t_1$, $\Pi_{chan(B)}(t) = t_2$ and $M(t_1, t_2, t)$ by the definition of matching and the meaning of the M predicate. The soundness of the rule is now obvious.

Before we prove the relative completeness of the proof system we first introduce the following definition and lemmas.

Consider a network A and an assertion P over the channels of A. The assertion P is said to be *precise* for A if $t \in \mathcal{A}(A)$ if and only if $P(t)$. So, an assertion is precise for a network if it is the strongest statement (in our assertion language) that can be made about the behaviour of that network. This idea will be crucial in proving that our proof system is relatively complete.

Lemma 1 *Let $C = A||B$ and let trace t have events only from channels of A and B. If $\Pi_{chan(A)}(t) \in \mathcal{A}(A)$ and $\Pi_{chan(B)}(t) \in \mathcal{A}(B)$ then $t \in \mathcal{A}(C)$.*

Proof: Suppose t is a trace satisfying the conditions of the lemma. Now we need to establish that $t \in \mathcal{A}(C)$. Let e_1 and e_2 be two events in t. Suppose they are required to satisfy a precedence relation in every trace of the network C. We need to show that they must satisfy this relation in t.

How might such a requirement arise? The two events may be both from one of the networks and may be required to satisfy the condition there. In that case they certainly satisfy this relation in t because they must satisfy the relation in the appropriate projection of t. The other possibility is that the events are from different networks but the matching process forces them to have a definite precedence, say $e_1 \rightarrow e_2$ for definiteness. If matching is forcing them to satisfy a particular precedence then there must be an event e which results from the identification of two events e' and e'' in the two networks. Furthermore, it must be the case that $e_1 \rightarrow e'$ and $e'' \rightarrow e$. But since $\Pi_{chan(A)}(t) \in \mathcal{A}(A)$ and $\Pi_{chan(B)}(t) \in \mathcal{A}(B)$ it must be the case that e occurs in t and again because the projections of t are viable behaviors of the original networks we have $e_1 \rightarrow e$ and $e \rightarrow e_2$ in t. Thus $e_1 \rightarrow e_2$ in t. Because the events picked are arbitrary it must be the case that *all* the causal and other constraints of C are satisfied by t. In other words, the trace t is a possible behavior of C.

Note that this lemma is essentially a non-interference condition which we have proven semantically. The key ingredient which makes this work is the matching process. This is similar to the situation with Soundarajan's [14] proof system for CSP. He uses a predicate called "compat" which performs a function similar to our matching. It is interesting to note that similar ideas work for both synchronous and asynchronous systems.

Lemma 2 *The Network Composition axiom preserves preciseness.*

Proof: Consider $C = A||B$. Let P and Q be precise specification for networks A and B respectively. Now if t has events only from channels of A and B and $(P \wedge Q)(t)$ then we must show that $t \in \mathcal{A}(C)$. From the definition of Π it follows that $P(\Pi_{chan(A)}(t))$ and $Q(\Pi_{chan(B)}(t))$. Since P and Q are precise specifications for networks A and B respectively it follows that $\Pi_{chan(A)}(t) \in \mathcal{A}(A)$ and $\Pi_{chan(B)}(t) \in \mathcal{A}(B)$. From Lemma 1 it then follows that $t \in \mathcal{A}(C)$.

To complete the proof we must show that if $t \in \mathcal{A}(C)$ then $(P \wedge Q)(t)$ - but this follows from the soundness of the Network Composition axiom.

Theorem 2 *The proof system is relatively complete.*

Proof: We must show that if $R(t)$ is a true specification where $t \in \mathcal{A}(C)$ then $R(t)$ is provable. Let S be the precise assertion for C then it follows that $S \Rightarrow R$ and hence $R(t)$ follows from one application of the Consequence axiom.

5 Fixed Point Semantics and Induction

The discussion so far has followed the standard practice of handling recursively defined networks through their (infinite) unfolding. Here we discuss how the fixed point semantics given by [7] can be used as the basis for an inductive proof rule.

First we recapitulate the fixed point semantics of Keller and Panangaden [7]. We shall view the denotations of modules as functions that act on archives. The actions of these functions on archives is the pointwise extension of their actions on individual histories. The action on a single trace inserts new events that correspond to the participation of that module in the trace. Thus these functions are called *extension rules.*

The archive of a network can be constructed by composing the extension rules of the component modules. Networks that contain loops have extension rules obtained by taking fixed points of the (recursive) composition of the extension rules of the component modules. The sense in which a fixed point is taken will be clarified later in this section, we do not use traditional complete partial order theory but give a construction that is similar to the category theoretic notion of a limit.

Let A be an module and let its archive be $\mathcal{A}(A)$. Let $\mathcal{A}(i)$ be the archive of all possible input histories to A i.e. $\mathcal{A}(i)$ consists of all traces that are serializations of possible events on the input arcs to A. Such an archive is called an *input* archive. Then the *extension rule* for A, written $\mathcal{E}[A]$, is the function from archives to archives which satisfies

$$\mathcal{E}[A](\mathcal{A}(i)) = \mathcal{A}(A)$$

It follows from this that if $\mathcal{A}(i)$ is any input archive then

$$\Pi_{inchan(A)}(\mathcal{E}[A](\mathcal{A}(i)) = \mathcal{A}(i)$$

in other words an extension rule will leave the original events unaltered.

An extension rule is required to satisfy several conditions. These conditions are essential for defining extension rules for networks containing *loops.* To express these conditions we use the following notation:

e stands for a generic extension rule

t stands for a generic trace

$\prec$ stands for the trace embedding relation. In other words, if $t \prec t'$ then every event in t appears in t' with relative event order preserved.

We shall also make use of the following terminology. An event v, in a trace t, is said to be *consumable* by an extension rule e, if it is a production event that appears on the input channel of an module in the network described by e and there is a trace in the archive that contains all the events of t together with a consumption event corresponding to v. An event v, is said to be *producible* in a trace t, by an extension rule e, if there is a trace in the archive of the network described by e that contains all the events in t and also contains v. We shall use the word *enabled* to mean either consumable or producible.

The conditions can now be stated as:

ER1 The action of an extension rule cannot remove an event from a trace; in symbols, if $t' \in e(t)$ then $t \prec t'$. This condition shall be referred to as the *monotonicity* condition.

ER2 An extension rule can only insert an assimilation event after the corresponding creation event and a creation event can be inserted only after the assimilation events that triggered that creation event. This rule will be referred to as the *causality* condition.

ER3 An extension rule must insert assimilation events of a given channel into the trace being extended in the same order as the corresponding creation events of that channel.

ER4 Within a chain of traces $t_1, t_2 \ldots t_i \ldots$ where for each i, $t_{i+i} \in e(t_i)$, no event remains assimilable, but not assimilated, forever and no producible event for the trace remains uninserted forever. This condition is called the *finite delay* condition.

The condition ER2 essentially ensures that an extension rule behaves in a causal fashion. The third condition, ER3, asserts that the data channels behave like queues. The fourth condition is a finite delay condition that is used to show that a suitable notion of taking limits of sequences can

be defined. It is actually a condition used in constructing the limits of sequences of traces for networks with loops rather than a constraint on the presentation of an extension rule as a function on traces. This condition is similar to that of [5].

In networks with loops the extension rule is defined by iterating the extension rule for a network without a loop. However, the limit of such a sequence of iterations needs to be defined. As is discussed at length in [11] it is not possible to use domain theory. Here we shall sketch the construction described in [8,7].

The notion of limit that we shall define is an inverse limit construction. Suppose we have an module A that has an input channel a and an output channel b. Let the extension rule for this module be denoted by $\mathcal{E}[A]$. Now suppose that the channel b is connected to the channel a. The extension rule must be modified by replacing every reference to b in $\mathcal{E}[A]$ by a. Let this modified extension rule be denoted by $\mathcal{E}_{ab}[A]$. Now whenever $\mathcal{E}_{ab}[A]$ is used to extend a given trace t there will be new input events to A in the extended trace $\mathcal{E}_{ab}[A](t)$. The extension rule $\mathcal{E}_{ab}[A]$ must therefore be applied again to the result. Given a particular initial trace, t, applying the extension rule $\mathcal{E}_{ab}[A]$ may result in several traces. Applying $\mathcal{E}_{ab}[A]$ to each trace in the resulting set of traces will also yield several new traces. Let us denote the set containing the original trace, t, by S_0, and the sets containing the subsequent extensions by $S_1, S_2, \ldots S_i, S_{i+1} \ldots$ respectively. As we construct each subsequent extension, we define relations, written R_i, from S_{i+1} to S_i that express which traces in S_i were extended by $\mathcal{E}_{ab}[A]$ to yield particular traces in S_{i+1}. Formally, if $t_1 R_i t_2$, then $t_1 \in S_{i+1}, t_2 \in S_i$ and $t_1 \in \mathcal{E}_{ab}[A](t_2)$. We refer to such a sequence of sets and relations as the *tower* of $\mathcal{E}_{ab}[A]$ over t.

Roughly speaking, the idea is to follow the chains indicated by the relations R_i and take the limit of each such chain. The difficulty with this is that if $t_i R_k t_j$ it does not mean that t_j is a prefix of t_i. However, as more events are added (which means that we are going higher up the tower) a longer and longer prefix of the traces are fixed, in the sense that no new events can be inserted into them. We shall say that such a prefix is *stable*. Now as we travel up a chain and pull out the maximal stable prefix of each trace we get a sequence $s_1, s_2, \ldots$ which does form a chain in the conventional domain theoretic sense with the prefix ordering. We define

the limit of a chain as the limit of the sequence of stable prefixes. It is, of course, not obvious that a growing sequence of such stable prefixes can be constructed but in fact this follows from the conditions that an extension rule must satisfy as is proved in [7]. The limit of the entire tower is obtained by constructing the limit in the above fashion for every sequence of traces in the tower. When this construction is used on an extension rule $\mathcal{E}$ we call the resulting extension rule $\mathcal{E}^*$.

The formulation of the inductive proof rule mimics this semantic account very closely. Essentially the idea is to use the extension rule $\mathcal{E}$ for a network rather than a logical specification. We can then do induction on the number of iterations of $\mathcal{E}$. We also assume that we can write a function S which given a trace returns the maximal stable prefix. For a given network these functions are fairly easy to write. Given these we can formulate the induction principle as follows:

Axiom 6 (Induction)

$$\frac{\forall n \in N \forall t \in \mathcal{E}^n(t_0) \forall t' \in \mathcal{E}(t).P(S(t)) \Longrightarrow P(S(t'))}{\forall t \in \mathcal{E}^*(t_0).P(t)}$$

Note that this is essentially fixed point induction being used on each chain in the tower. Stated in this way the induction rule is clumsy to use but we are now free to use induction. In our examples we will not actually use the induction rule literally as stated but will show how the properties of interest can be proved inductively.

6 Examples

We now illustrate the module of reasoning by induction by considering the merge anomaly discovered by Keller [6]. In this network we have a module called "merge". It has two input channel and an output channel. The module produces an arbitrary interleaving of the tokens on the input channels. In the example the output channel is connected to the input channel. Let that channel be called b and the other channel a. Consider the situation where there are two tokens, 1 followed by 2, on channel a. It is easy to see operationally that we have on channel b some sequence of 1's followed by an alternating sequence of 1's and 2's.

We would like to prove that for any trace t of the Merge network, two consecutive b^+ events with value 2 are not produced. We will prove this inductively on the lenght of $\Pi_b^+(t)$.

Base case. Trivial since there are no events in $\Pi_b^+(t)$.

Induction step. Assume that there are not two consecutive events with value 2 on $\Pi_b^+(t)$ when $||\Pi_b^+(t)|| = n$. We now prove by contradiction that when t is extended so that $||\Pi_b^+(t)|| = n + 1$ there are still no two such consecutive events on $\Pi_b^+(t)$.

Suppose that by extending t we obtain two consecutive events with value 2 on $\Pi_b^+(t)$. Now it must be the case that a $< +, b, 2 >$ event has been inserted into $\Pi_b^+(t)$ and this event is next to an existing $< +, b, 2 >$ event otherwise we would not get two consecutive events on $\Pi_b^+(t)$ with value 2 in this iteration. What could have triggered this new event? It must have been an earlier $< -, a, 2 >$ or $< -, b, 2 >$ event. In the former case we find that we cannot have a causal predecessor for the already existing $< +, b, 2 >$ event. Thus it must be the latter case. Thus there must already have been two b-events with value 2. What event separated them? It can only have been a b-event with value 1. But then we are violating the causality condition since the event with value 1 should produce a result first in the stable part of the trace. Thus no iteration of the extension rule can produce two consecutive events on $\Pi_b^+(t)$ with value 2.

We now present another example network which we call the ***Every−Other*** network or $E-O$ network for short. This network is illustrated in Fig. 1.

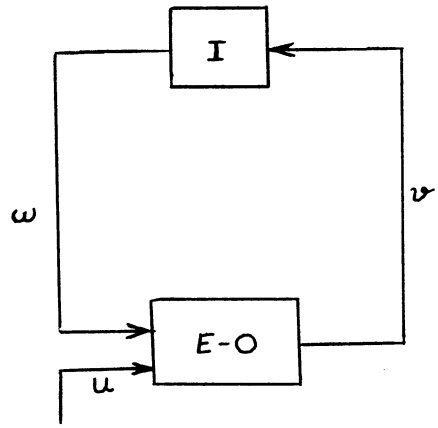

Fig. 1. The ***Every−Other*** Network

In Fig. 1, the module I is the identity module that continually reads its input token on input channel v and outputs it on the output channel w. The module $E-O$ reads every alternate token from the input channels, u and w in this case, and outputs the token on channel v. Now consider the scenario where we are only interested in talking about the lengths of token streams on the various channels. In that case we can prove inductively for any trace t of the $E-O$ network that $||t_v||$ is odd whenever $||t_u|| \geq 2$. We do this as follows. We can think of $E-O$ as having two unit size buffers, each associated with the two input channels. Initially both these buffers are empty and the way $E-O$ module works is that it repeatedly read a value on either input channel and if the corresponding buffer is empty then that value is placed in the buffer otherwise if the buffer is full then the buffer is emptied and the value read is output on channel v. We define an assertion P as follows.

$$P = (||t_w|| \text{ is odd} \wedge ||t_v|| \text{ is odd} \wedge \text{buffer in } E-O \text{ for } w \text{ is full})$$

We will prove inductively that this assertion P is true of the $E-O$ network when $||t_u|| \geq 2$. Notice that for this network it is sufficient to consider the case where $||t_u||$ is even.

Base case. When $||t_u|| = 2$ then the consumption of the two values on channel u produces one value on channel v. This value produced on channel v is consumed by module I to produce one value on channel w. This value on channel w is in turn consumed by the module E–O and no further event takes place; so $||t_v|| = 1 \wedge ||t_w|| = 1$ and the buffer corresponding to w is full.

Induction step. Suppose P is true when $||t_u|| = 2n$. Now consider the case where $||t_u|| = 2n + 2$. Then the two extra values on channel u produce an extra event on channel v. This produces an extra event on channel w. Now since the buffer corresponding to w is full, consumption of this new event on channel w empties the buffer and produces another event on channel v. This event again leads to an event being produced on channel w and now since the buffer for this channel is empty this event is consumed by $E-O$ module and no action is taken. No further action can be undertaken by the $E-O$ network and since a total of two event are produced on both v and w because of the extra two events on u, and the buffer correponding to w is full it follows that P is satisfied when $||t_u|| = 2n + 2$.

7 Conclusions

We have given a denotational semantics and proof system for networks containing indeterminate operators. The new feature of our presentation is that we have given an inductive proof rule. As the examples illustrate, the inductive proof rule is useful but awkard to use. This complexity is a reflection of the discontinuous nature of fairness. It is possible that a transfinite induction principle would lead to a cleaner approach to inductive proof rules for fair networks, such a possibility is being investigated. Though we have based our proof system on a semantic theory which was designed to give a fixed point account of fairness we have not focussed on proof rules for reasoning about fairness. Such a proof system is under development currently.

Though the inductive proof rule is tedious to use it is a proof rule which can be mechanized. Thus we do not expect that a user would have to grapple with all the tedious details by herself but could use automated tools to take care of most of the details. Such efforts are in fact being carried out by members of Robert Constable's PRL project at Cornell under the direction of the second author.

References

[1] Samson Abramsky. On semantic foundations for applicative multiprogramming. In *Proceedings of the International Conference on Automata Languages and Programming*, 1983.

[2] J. D. Brock and W. B. Ackermann. Scenarios:a model of nondeterminate computation. In I.Ramos J.Diaz, editor, *Formalization of Programming Concepts*, pages 252–259, Springer-Verlag, New York, 1981.

[3] Zhou Chao Chen and C. A. R. Hoare. Partial correctness of communicating sequential processes. In *IEEE International Conference on Distributed Computing Systems*, 1981.

[4] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP 1974*, pages 471–475, 1974.

[5] Richard M. Karp and R. Miller. Parallel program schemata. *JCSS*, 3(2):147–195, May 1969.

[6] R. M. Keller. *Denotational Models for Parallel Programs With Indeterminate Operators*, pages 337–365. North-Holland, Amsterdam, 1978.

[7] Robert M. Keller and Prakash Panangaden. Semantics of digital networks containing indeterminate modules. *Distributed Computing*, 1986.

[8] Robert M. Keller and Prakash Panangaden. Semantics of networks containing indeterminate operators. In S. D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 479–496, 1985.

[9] Daniel J. Lehmann. *Categories for Fixpoint Semantics.* PhD thesis, University of Warick, 1976.

[10] Van Nguyen, David Gries, and Susan Owicki. A model and proof system for networks of processes. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985.

[11] Prakash Panangaden. *A Denotational Semantics for Networks Containing Indeterminate Operators.* Master's thesis, University of Utah, 1985.

[12] V. Pratt. On the composition of processes. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 213–223, ACM, January 1982.

[13] Vaughn Pratt. The pomset model of parallel processes. In S. D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 180–196, 1985.

[14] Neelam Soundararajan. Axiomatic semantics of communicating sequential processes. *Transactions on Programming Languages and Systems*, 6(4):647–662, 1984.

[15] Job Zwiers, Willem Paul de Roever, and Peter van Emde Boas. Compositionality and concurrent networks. In *International Conference on Automata Languages and Programming*, 1985.