Nonexpressibility of Fairness and Signaling

David McAllester
Prakash Panangaden*
Vasant Shanbhogue

TR 89-972
February 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Research supported in part by NSF grant DCR-8602072.

Nonexpressibility of Fairness and Signaling

David McAllester* Prakash Panangaden†
Vasant Shanbhogue‡
Computer Science Department, Cornell University

March 2, 1989

Abstract

In this paper we establish new expressiveness results for indeterminate dataflow primitives. We consider choice primitives with three differing fairness assumptions and show that they are strictly inequivalent in expressive power. We also show that the ability to announce choices enhances the expressive power of two of the primitives. These results are proved using a very crude semantics and thus apply in any reasonable theory of process equivalence.

1 Introduction

Fairness is regarded as an important property of real systems and there is considerable interest in semantic theories and proof systems for reasoning about fairness [9]. In the present paper we examine the relative expressive power of a variety of fair choice primitives. We prove new inexpressibility results in the context of asynchronous systems. We prove that three different choice primitives have different expressive power. We also consider the effect of adding signaling to each primitive. By "signaling" we mean that a choice primitive has a mechanism for announcing what choices it has made.

*Currently at MIT
†Supported in part by NSF grant DCR-8602072.
‡Supported in part by NSF grant DCR-8602072.

Our investigation is carried out in the context of static dataflow networks, i.e. networks whose structure remains fixed throughout execution.

Our interest in this work stemmed from earlier discoveries by Panangaden and Stark [23,26,24] that the so called fair merge primitive [14] is strictly "more powerful" than other primitives exhibiting unbounded indeterminacy. This showed that one could not classify indeterminate primitives on the degree of branching they embodied. All fair systems include primitives with countable indeterminacy [8]. In the programming model studied by Chandra [3,8], countable indeterminacy and fairness are equivalent. In the case of asynchronous dataflow networks [13], the analysis is complicated by the fact that a process may receive data from different autonomous processes in an asynchronous fashion. This means that fair merges need to avoid empty data channels as well as to make fair choices.

Work by Apt and Plotkin [4] shows that the presence of countable indeterminacy in a programming language leads to failures of continuity. The result about fair merge shows that there is a breakdown of a monotonicity property that occurs in that case.

Having identified monotonicity as a property that differentiates two kinds of countable indeterminacy, we are led to focus attention on monotone primitives. Since semantics of networks including fair merge are notoriously difficult it is possible that one might develop simpler semantic theories for systems that do exhibit countable indeterminacy but are monotonic. We discovered that there were provably inequivalent primitives here too. This paper discusses these primitives and establises the difference in their expressive power. There appears to be a richer taxonomy of indeterminate primitives than had been suspected earlier.

Recently there has been considerable interest in developing semantic theories to handle countable indeterminacy [2,4,5,7,15,27]. Our work shows that there are several flavors of countable indeterminacy. We feel that such a semantic theory must take this diversity into account.

In the rest of this introduction we describe the setting and state the results informally. The following section describes an automata-theoretic formalism essentially due to Lynch and Tuttle [18] and Stark [32]. We show how one can pass from these automata to traces of the networks. Recent work by Jonsson [12] (and independently by Panangaden and Shanbhogue [25]) shows that traces are fully abstract for such networks and hence constitute a good abstraction of the detailed operational aspects of network behaviour.

The rest of the paper works with traces exclusively and develops the machinery to reason about process equivalence and implementability. We use a very weak notion of process equivalence; one that is not even a congruence. The significance of this is that our nonimplementability proofs will survive any passage to a more accurate semantic theory. Clearly our positive implementability results are then of not great significance.

1.1 Kahn Networks and Indeterminate Primitives

We define an asynchronous dataflow network to be a finite set of autonomous computing agents, called *nodes*, connected by directed arcs, called *channels*. The directed arcs coming into a node are called input channels and those leaving a node are called output channels. The interconnection structure is fixed throughout execution. Nodes can only "listen" to a single channel at a time. One can think of each node as executing a sequential program. Communication between nodes is effected by the transmission of messages along the channels. The channels are unbounded queues where the sending of a message and the receipt of the message are distinct activities. There is no synchronization on message passing such as in CSP [10] or CCS [20].

We consider abstractions of different schedulers. This leads to three primitives that we call *choice* processes. Each can be regarded as a dataflow primitive with an input port and two output ports. Tokens are consumed from the input port and are placed on one or other of the output ports. One can now distinguish between different choice primitives on the fairness properties that they satisfy with respect to choosing between the output channels. The inexpressiveness results here may be considered to be in the same spirit Stark's investigation to the expressive power of semaphore primitives [31] extended to the dataflow case.

Another inexpressiveness phenomenon at work here arises from *sequentiality*. We consider augmenting the choice primitives with an additional output channel on which a bit is output every time a choice is made. This allows other processes in the network access the choices made. It turns out that this interaacts quite delicately with the fairness properties. With a strong fairness assumption one can show that adding signaling does not add to the expressive power, whereas with a weaker fairness property one can prove that the choices that cannot signal are strictly weaker. The proof

methods hinge on using the fact that individual processes are sequential in an appropriate sense. One may also view this as an analysis of how information gets dispersed in a network.

We describe the six distinct choice nodes below:

1. **Choice(C)** has one input channel and two output channels. It reads a, possibly infinite, stream of values and splits it into two streams, possibly being unfair in the sense that one output channel may receive no input values for an infinite input stream.

2. **Weakly fair choice(WFC)** is similar to **choice** except that, for an infinite input stream, each output stream will have at least one value. Nothing is guaranteed if the input stream is finite.

3. **Strongly fair choice(SFC)** is similar to the above, except that for an infinite input stream, each output stream is guaranteed to be infinite.

4. **Choice with signal(CS)** is identical to **choice** except that there is a third output channel called the *signal channel.* The ith value output on this channel is 0 if the ith input value was output on the first output channel and is 1 if the ith input value was output on the second output channel. If there is no ith input value then there is no ith output value on the third output channel.

5. **Weakly fair choice with signal(WCS)** is identical to **weakly fair choice** except that it has a signal output channel.

6. **Strongly fair choice with signal(SCS)** is identical to **strongly fair choice** except that it has a signal output channel.

1.2 Results

The expressiveness situation that we establish is depicted in Figure 1. An arrow between two primitives indicates that there exists a network built from instances of the first primitive and "ordinary" (essentially sequential and deterministic) nodes that implements the IO-relation of the second primitive. An arrow with a line through it indicates that we have proven that no such implementation is possible.

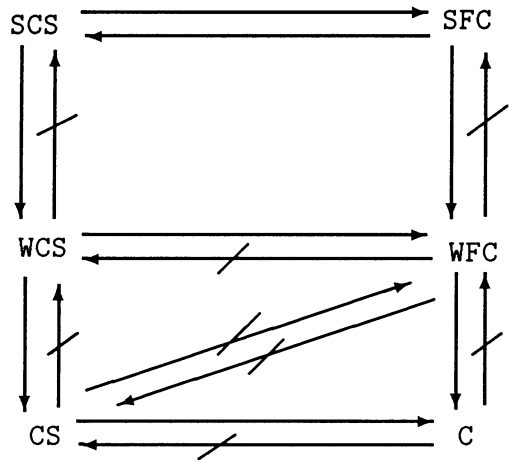

Figure 1: The Relative Expressiveness of Choices

Based on experience with semaphores [31], one might expect that choice cannot implement WFC. It is somewhat more surprising that WFC, or even WCS, cannot be somehow "iterated" to implement SFC. The second interesting result is that we cannot simulate signals *except* when we have strong fairness. This result seems to be related to sequentiality.

2 Processes and Networks

In this section, which is essential to justify that the formalism used in the proofs actually corresponds to some operational description, we establish the basic definitions and give an abstract operational formulation of the class of networks that we consider. The formalism is based on work of E. W. Stark [32,34]. We will define the notion of a "computation sequence", and identify which are the "valid" ones. We will need to analyze the local activity of each process in a network. So we will need "traces", abstractions of computation sequences of the automata, with rich enough information that the local activity of each process can be extracted. The corresponding automata from which these traces are constructed needs to distinguish

between these different kinds of events – arrival events, output events and internal events, of which input events are a special case. We will use traces in our discussion and arguments. We show that this use of traces is justified by showing, on operational grounds, that traces of networks can be described in terms of traces of subnetworks. So we can safely abstract traces from computation sequences of networks, instead of first obtaining traces of the individual processes in the network, and then composing them.

Concurrency is represented by interleaving of "causally independent" events. Concurrency can be analyzed by examining which pairs of events can be permuted, that is, can happen in either order.

In all our implementability and non-implementability proofs, whenever we talk of building a network of processes using a particular process P, we assume that we can use a base set of processes, that correspond to normal sequential programs. In other words, the network may contain processes from this base set, besides copies of process P. We will formally define which processes are assumed to be in this base set.

2.1 Automata

We describe individual processes in terms of a particular kind of automaton, that communicates by sending "data values" through "ports". The set of events of an automaton comes equipped with a concurrency relation, that describes which pairs of events are causally independant and can be permuted in execution sequences.

Definition 1. A **concurrent alphabet** is a set X, together with a symmetric, irreflexive binary relation $\|$ on X, called the **concurrency relation**.

This concept is used in trace theory [1,19] to obtain an algebraic structure for traces. Formally, let V be a set of **data values** called the *value alphabet.* Throughout this paper, we will assume a fixed countable value alphabet.

Definition 2. A **port automaton** is a tuple

$$M = (E, Q, A)$$

where

- E is an alphabet of **events** equipped with a concurrency relation. Let **Arr,Out** and **Inp** be disjoint subsets of E, called the sets of **arrival , output** and **input events**, respectively. $\mathbf{Arr} = P^{in} \times \{+\} \times V$, $\mathbf{Out} = P^{out} \times \{+\} \times V$, and $\mathbf{Inp} = P^{in} \times \{-\} \times V$ for some disjoint finite sets P^{in} and P^{out}. The elements of P^{in} are called **input ports**, and the elements of P^{out} are called **output ports**. The events $(p_0, +, v)$ and $(p_0, +, v')$ are not related by the concurrency relation of E, for any p_0 and any $v \neq v'$. Neither are any pair of events e and e' related by the concurrency relation, if both e, e' are in $E \backslash \mathbf{Arr}$. The elements of $E \backslash (\mathrm{Arr} \cup \mathrm{Out})$ are called **internal events**.

- Q is a set of **states**, and $q^\iota \in Q$ is a distinguished **initial state**.

- A is a transition function that maps each pair of states q, r in Q to a subset $A(q,r)$ of $E \cup \{\epsilon\}$. ϵ is called the **identity event**. If $a = (p, +, v) \in A$, or $a = (p, -, v) \in A$, then we write *port(a)* for the port component p, and *value(a)* for the value component v.

satisfying the following conditions :

(Disambiguation) $r \neq r'$ implies $A(q,r) \cap A(q,r') = \emptyset$.

(Identity) $\epsilon \in A(q,r)$ iff $q = r$.

(Receptivity) For all states q and arrival events a, there exists a state r such that $a \in A(q,r)$.

(Commutativity) For all states q and events a, b, if $a \| b$, $a \in A(q,r)$ and $b \in A(q,s)$, then there exists a state p such that $a \in A(s,p)$ and $b \in A(r,p)$.

This definition is similar to the definitions of a *port automaton* and an *input-output automaton* due to Stark in [17,26,33]. We explicitly introduce *input events* here. Note that, by the definition of the concurrency relation, if two events are concurrent, then either they are both arrival events on different input ports, or one of them is an arrival event, and the other is an internal event or an output event.

From a particular state, an event cannot take us to two different states. This is what *disambiguation* says.

We would also like to have arrival events always "enabled". The arrival of data on input channels should not be dependent on the state, and so, for any state and for any event corresponding to a value arriving on an input channel, there is a new state corresponding to the value having arrived. This is captured by *receptivity.*

Moreover, if two events are concurrent, according to the concurrency relation, and if both of them are "enabled" in a particular state, then doing any one of these two events does not "disable" the other, and moreover, doing both these events in either order, brings us to the same final state. This is captured by *commutativity.*

The *transitions* of an automaton are the triples (q, a, r) with $a \in A(q, r)$. We may denote the transition (q, a, r) by $q \xrightarrow{a} r$. The transition $q \xrightarrow{\epsilon} q$ is called an *identity transition*, and is denoted by id_q.

One should note that there is a difference between the notions of *event* and *transition.* A transition describes two states and an event such that when it is executed in the first state, one reaches the second state. An event may execute in different states. For example, a "$x := x + 1$" event may be executed in a state in which x is 3, as well as in a state in which x is 4. But they will correspond to different transitions.

Definition 3. A **computation sequence** γ is a finite or infinite sequence of transitions of the form

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$$

The *domain* $dom(\gamma)$ of γ is the state q_0. A computation sequence is said to be *initial* if $dom(\gamma)$ is the distinguished start state q^ι. Two computation sequences γ and δ are *coinitial* if $dom(\gamma) = dom(\delta)$.

We explicitly introduced a set of events called *arrival events*, and a set of events called *input events*, in the definition of a port automaton. The intent was that arrival events should represent arrival of data at input channels, and that input events should represent the consumption of data from the input channels. Arrival of data must always precede consumption of data.

Moreover, in some computation sequence

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$$

if a_i is an arrival event and a_{i+1} is an event that is neither the corresponding input event, nor an arrival event on the same input channel, then the arrival

event a_i does not "cause" the event a_{i+1}, and so could occur after a_{i+1}. Formally, there must exist a state q' such that $q_{i-1} \stackrel{a_{i+1}}{\rightarrow} q'$ and $q' \stackrel{a_i}{\rightarrow} q_{i+1}$. We shall refer to this as a **pushforward** of an arrival event.

The above requirements are seen to be natural, if one describes states of an automaton to include explicitly the contents of the input channels. We then see that a input event can happen only if the content of the input channel, as described by the corresponding sequence in the state, is non-empty, and the value being read in the input event is the same value as the one at the head of the sequence. Rather than describe states in such an unwieldy manner, we will impose these natural conditions on our automata.

So we are naturally led to the following definition to capture the notions of arrival and input.

Definition 4. A **causal port automaton** is a port automaton, such that for any finite initial computation sequence s, and for any input port p, $\Pi_{p,-}(s)$ is a prefix of $\Pi_{p,+}(s)$, and moreover, the set of computation sequences is closed under pushforwards.

The first part of the above definition ensures that arrival of data precedes their being read.

We will now give two examples of automata.

Example 1. *Buffer* : This process has one input channel and one output channel, and simply reads values and outputs them, guaranteeing to read and output all values that arrive on the input channel.

Let V be a set of data values, and V^* be the set of all finite sequences of values from V, including the empty sequence $<>$. Let the set of states Q be $V^* \times (V \cup \{\epsilon'\})$. A state represents the contents of the input channel, and a value, if any, that has been read but not yet output. ϵ' indicates that there is no value that has been read but not output. Let the set of arrival events Arr be $\{i\} \times \{+\} \times V$, the set of output events Out be $\{o\} \times \{+\} \times V$, and the set of input events Inp be $\{i\} \times \{-\} \times V$. Let the set of all events E be ArrUOutUInpU$\{\epsilon\}$, where ϵ is the identity event.

We will now define the transition relation. $A(q,r) = \{(i,+,v)\}$ iff either $q =< s, a >$ and $r =< s \cdot v, a >$, or, $q =< s, \epsilon' >$ and $r =< s \cdot v, \epsilon' >$. $A(q,r) = \{(i,-,v)\}$ iff $q =< v \cdot s, \epsilon' >$ and $r =< s, v >$. $A(q,r) = \{(o,+,v)\}$ iff $q =< s, v >$ and $r =< s, \epsilon' >$. $A(q,q) = \{\epsilon\}$.

Every event in Arr is concurrent with every event in Out∪Inp, and ϵ is concurrent with any other event.

Example 2. *Poll* : This process has one input channel and one output channel. It repeatedly polls its input channel for data. If a data value is present, then it is read and output. If not, a special value $*$ is output.

Let Q, Arr and Inp be the same as for the previous example. Let the set of output events Out be the set in the previous example, together with an extra event $(o, +, *)$.

Besides the transitions in the previous example, there is an extra transition $A(q, q) = \{\epsilon, (o, +, *)\}$ iff $q = <<>, \epsilon' >$. $A(q, q) = \{\epsilon\}$ otherwise. The only thing new added to the concurrency relation of the previous example is that ϵ is concurrent with $(o, +, *)$.

Notice that arrival events are not concurrent with $(o, +, *)$, so that arrival of input disables an output of a $*$. So the input has the power of interrupts.

Our automata will be simple, in the sense that arrival of input cannot disable an already enabled output. If we had represented states as containing the contents of the input channels as well, then what this means is that if a value arrives in an input channel, then this does not affect the process's "internal state". So if the process could output a value before, it can still do so, except that it might now have the option of reading a value.

We formalize this natural requirement below.

Definition 5. A port automaton is said to be **monotone**, if it satifies the following property :
(Monotonicity) For any arrival event a, and any event b that is not an arrival event at the same port, $a \| b$.

Similar to the notion of a pushforward of an arrival event, defined earlier, we can now define a **pushback** of an arrival event. In some computation sequence

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$$

if a_{i+1} is an arrival event and a_i is an event that is not an arrival event on the same input channel, then the arrival event a_{i+1} could happen earlier, before the event a_i. Formally, there must exist a state q' such that $q_{i-1} \xrightarrow{a_{i+1}} q'$ and $q' \xrightarrow{a_i} q_{i+1}$.

Note that closure of computation sequences under pushbacks follows from the monotonicity of the automaton.

All the port automata that we shall consider will be causal port automata with the monotonicity property, so we shall henceforth refer to these simply as automata.

We often decompose computation sequences into subsequences in order to focus on the local activity of a node, or the activity at a single port. We define the following notation for this purpose.

Definition 6. For any computation sequence σ, we define $\Pi_{p,+}(\sigma)$ to be the value sequence $v_1, v_2, \ldots$ such that $(p, +, v_1), (p, +, v_2), \ldots$ is the subsequence of σ consisting of the arrival events at port p.

Definition 7. For any computation sequence σ, we define $\Pi_{p,-}(\sigma)$ to be the value sequence $v_1, v_2, \ldots$ such that $(p, -, v_1), (p, -, v_2), \ldots$ is the subsequence of σ consisting of the $-$-events at port p.

Let P be the set of input ports and output ports of an automaton, and let V^∞ be the set of all finite and infinite sequences of elements of V. A *port history* is defined to be a function from P to V^∞. Then for any computation sequence σ, we can define a history H_σ by letting $H_\sigma(p)$ be the value sequence $\Pi_{p,+}(\sigma)$. We denote the restriction of H_σ to the input ports by H_σ^{in}, and call it the *input port history* corresponding to σ. We denote the restriction of H_σ to the output ports by H_σ^{out}, and call it the *output port history* corresponding to σ.

The main tool for working with computation sequences of automata is the notion of a "residual" $\gamma \uparrow \delta$ of one finite computation sequence γ "after" another coinitial sequence δ [17,26,33,34]. Intuitively, γ and δ may "overlap", and the residual is the part of γ that is left to do after doing δ.

Our discussion could have been carried out using port automata, as defined in [26], with internal events, by taking the "residual" as fundamental, and defining the concurrency relation in terms of the "residual". It turns out that in both cases, we obtain the same "residuals" and the same concurrency relation. Since a concurrency relation is more easily understood, we have developed our discussion by taking the concurrency relation as fundamental.

Definition 8. **A residual** is a partial operation $\uparrow$ on coinitial pairs of computation sequences. For single transitions $t : q \xrightarrow{a} r$ and $u : q \xrightarrow{b} s$, if $a = b$, then $t \uparrow u = id_s$ and $u \uparrow t = id_r$. If $a \neq b$, then $t \uparrow u$ is defined iff $a \| b$, in which case the commutativity property implies the existence of transitions $s \xrightarrow{a} p$ and $r \xrightarrow{b} p$, which we take to be $t \uparrow u$ and $u \uparrow t$ respectively.

We say that coinitial computation sequences γ and δ are *consistent* if $\gamma \uparrow \delta$ is defined; otherwise we say they *conflict.* We extend to finite computation sequences as in [26].

The residual operation is now used to define a preorder $\sqsubseteq\!\!\!\sim$ between coinitial computation sequences as follows: For γ, δ finite, define $\gamma \sqsubseteq\!\!\!\sim \delta$ iff γ and δ are consistent and $\gamma \uparrow \delta$ is a sequence of identity transitions. We extend this definition to infinite computation sequences by defining $\gamma \sqsubseteq\!\!\!\sim \delta$ iff for every finite prefix γ' of γ, there exists a finite prefix δ' of δ, such that $\gamma' \sqsubseteq\!\!\!\sim \delta'$.

It can then be shown, as in [26], that if M is an automaton and $\uparrow$ is the corresponding residual operation, then the relation $\sqsubseteq\!\!\!\sim$ is a preorder on the set of all computation sequences of M, and $\sqsubseteq\!\!\!\sim$ extends the prefix ordering. Moreover, the set of all $\sqsubseteq\!\!\!\sim$-equivalence classes of computation sequences, with the induced partial order, is a Scott domain, whose finite elements are exactly the equivalence classes of finite computation sequences. Moreover, as in [26], the map that takes each computation sequence γ to its port history H_γ is continuous, with respect to the $\sqsubseteq\!\!\!\sim$ preorder on computations, and the prefix ordering $\preceq$ on port histories. This last fact suggests that $\sqsubseteq\!\!\!\sim$ is the "right" preorder to work with rather than the prefix ordering on computation sequences.

As in [26], we choose the "valid" computation sequences to be exactly the $\sqsubseteq\!\!\!\sim$-maximal ones. We will also refer to these as completed computation sequences. The point here is that these are the sequences in which all the continuously enabled events have actually happened. As in [26], these are exactly the "fair computations". It is quite pleasant to be able to state this as a maximality property of computation sequences. A discussion of the kind of "fairness" that we consider here is in the next subsection, where we discuss what are the "valid" computation sequences of a network of automata.

Definition 9. A computation sequence is called **completed** if it is $\sqsubseteq\!\!\!\!\sim$-maximal among all the computations having the same input port histories.

We will obtain "traces" from these completed computation sequences, by abstracting them.

2.2 Networks of Automata

In the previous section, we described a single automaton. We now describe how to compose automata to obtain networks.

Definition 10. Suppose $\mathcal{M} = \{M_i : i \in I\}$ is a finite collection of automata, where $M_i = (E_i, Q_i, A_i)$ and each E_i is partitioned as before into $\mathrm{Arr}_i, \mathrm{Out}_i, \mathrm{Inp}_i$ and other events. We call the automata **compatible** if

- for all $i, j \in I$ such that $i \neq j$ we have $(E_i \backslash (\mathrm{Arr}_i \cup \mathrm{Out}_i)) \cap (E_j \backslash (\mathrm{Arr}_j \cup \mathrm{Out}_j)) = \emptyset$, that is, the internal events of any pair of automata are disjoint, and,
- a given port can be shared by at most two automata, in which case it must be an output port of one of them and an input port of the other.

This restricts us to linking processes by one-way channels across a pair of ports. The shared port names represent ports that are connected. So we will now refer to shared port names as channels, and when we refer to a port, we will be referring to one end of a channel, identified by the process at that end. We also restrict ourselves to building finite networks. This is the form of interconnection in Kahn networks.

The following definition looks slightly complicated but all that it says is that when we compose a collection of compatible automata, we obtain an automaton. The "internal ports" of this automaton include the channels connecting any pair of the M_i's, that is, the port names shared by any pair of the M_i's. The input ports are all the input ports of the M_i's, excluding those that are shared. The output ports are all the output ports of the M_i's, excluding those that are shared.

Definition 11. The **composition** of a set $\mathcal{M}$ of compatible automata is the automaton $\prod M_i = (E, Q, A)$, where

- $E = \cup E_i$, with $a \| b$ iff $a \|_i b$ for all $i \in I$ such that both a and b are in E_i. This implies that events of distinct automata, that do not share any ports, are concurrent, because they are not both in the event set of any single automaton.

- $\text{Out} = (\cup \text{Out}_i) \backslash (\cup \text{Arr}_i)$, and $\text{Arr} = (\cup \text{Arr}_i) \backslash (\cup \text{Out}_i)$, and $\text{Inp} = \{e \in (\cup \text{Inp}_i) \mid \exists e' \in \text{Arrsuch that} \mathit{port}(e) = \mathit{port}(e')\}$

- $Q = \prod_{i \in I} Q_i$

- $q^\iota = (q_i^\iota : i \in I)$

- $e \in A((q_i : i \in I), (r_i : i \in I))$ iff for all $i \in I$, either $e \notin E_i$ and $r_i = q_i$, or else $e \in E_i$ and $e \in A_i(q_i, r_i)$.

The set $\text{Internal} = ((\cup \text{Arr}_i) \cap (\cup \text{Out}_i)) \cup (\cup \text{Inp}_i \backslash \text{Inp})$ is called the set of **internal port events**.

Arr$\cup$Out$\cup$Inp$\cup$Internal are called **port events**. Intuitively, the component automata in a network communicate by transmitting data values on internal channels. The corresponding events are in Internal.

Note that two automata connected by a channel may execute a single event in the composed automaton, but this might correspond to an output event of one of them and an arrival event of the other. For example, suppose A and B are the two automata connected by a channel p, that is p is an output port for A, but an input port for B. Then $(p, +, v)$ is an output event for A, but an arrival event for B. The execution of this event by the composed automaton corresponds to the outputting of value v by A, and the arrival of v at an input channel of B. By defining composition in this way, we do not have to worry about liveness conditions, stating that if a value is output by A, it will eventually arrive at an input channel of B. This does not violate asynchrony, because the corresponding input event, if it happens, may happen any time after the arrival event.

We note that the difference between a network of automata and a single automaton is that we can recover the structure of the single automata from the network by appropriate projections. A network can be thought of as an automaton, coming with a predefined decomposition. One may, of course, specify a large automaton without giving such a decomposition. It is not

clear to us how an arbitrary automaton may be decomposed into simpler automata.

With each component automaton M_i, we associate restriction functions ρ_i from states of the network to states of M_i, and α_i from events of the network to events of M_i. ρ_i is defined by $\rho_i((q_i : i \in I)) = q_i$, and α_i is defined by $\alpha_i(a) = a$, if $a \in E_i$, and $\alpha_i(a) = \epsilon$ otherwise. Then we can define the restriction of a computation sequence $\gamma = q_0 \stackrel{a_1}{\rightarrow} q_1 \stackrel{a_2}{\rightarrow} \ldots$ of the network to a component automaton M_i by

$$\pi_{M_i}(\gamma) = \rho_i(q_0) \stackrel{\alpha_i(a_1)}{\rightarrow} \rho_i(q_1) \stackrel{\alpha_i(a_2)}{\rightarrow} \ldots$$

We can define *port history, input port history and output port history* corresponding to computation sequences of networks, just as we did for computation sequences of single automata.

We note that it is possible, as in [26], to define a residual operation for the network, induced "componentwise" by the residual operations of the component automata. Then we can define a preorder $\stackrel{\sqsubseteq}{\sim}$ corresponding to this residual operation, and we can talk about the $\stackrel{\sqsubseteq}{\sim}$-maximal computation sequences.

As in [26], these are exactly the "fair executions". Intuitively, a computation sequence is *fair*, if every component automaton can execute an event, that is not an arrival event, if it tries for a sufficiently long, uninterrupted interval. Formally, if M is the composition of a compatible collection $\{M_i : i \in I\}$ of automata, then a finite computation γ of M is *fair* if no event, that is not an arrival event of the network, is enabled in state $cod(\gamma)$. An infinite computation sequence γ is *fair* if for each $i \in I$, either there exist infinitely many transitions in γ whose actions are events of M_i, that are not arrival events for M_i, or else there exist infinitely many states in γ for which no event of M_i, that is not an input arrival event of M_i, is enabled.

Lemma 1. Suppose M is the composition of a compatible collection of automata $\{M_i \mid i \in I\}$. Then γ is a completed computation sequence of M iff $\pi_{M_i}(\gamma)$ is a completed computation sequence of M_i, for all $i \in I$.

Proof : The proof is straightforward, from the definitions of *completed* and the concurrency relation for the composition of a compatible collection of automata. For more discussion on a similar proof, see [18,23].

3 Traces and Implementability

A computation sequence describes the sequence of states and the sequence of events, including internal events, that cause the state transitions. We now abstract away from states and most of the internal events, and only consider sequences of events on the input, output and internal ports of a network.

We will refer to single automata as nodes, and to networks of compatible automata as networks of nodes. Henceforth we will also refer to input ports, output ports and internal ports as input channels, output channels and internal channels respectively, unless we specifically wish to talk about the ends of the channels and the nodes at those ends.

Definition 12. A **trace** of a node is the restriction of a completed computation sequence of the automaton to arrival events, input events and output events.

Definition 13. A **trace** of a network of nodes is the restriction of a completed computation sequence of the network to port events. That is, a trace of a network will contain arrival, input and output events for every component node in the network.

The following lemma says that reading of data must be preceded by the arrival of that data.

Lemma 2. Every trace of any node B has the property that the ith input event on an input channel is preceded by the ith arrival event on that channel, and, furthermore, the ith arrival event has the same value as the ith input event.

Proof : Immediate from the definition of a causal port automaton. ∎

Definition 14. An event is **local to** a node B if it is either a $+$ or $-$ event on an input channel of B or a $+$ event on an output channel of B.

Definition 15. Let t be a sequence of events in a network N. The **projection** of t on a particular node B in N is the sequence of events obtained by deleting all events in t other than those local to B.

The following lemma is very important, as it tells us that the set of traces is compositional in nature. That is, we obtain the same set of traces, whether we obtain them directly from the completed computation sequences of the network, or whether we first obtain traces of subnetworks and then compose them in an appropriate manner.

Lemma 3. A trace of a network N is a sequence t of events such that for each node B in N the projection of t onto B is a trace of B.

Proof : ($\Rightarrow$) Suppose t is a trace of network N, and it is the restriction of a completed computation sequence γ to port events. Then the projection of t onto B is exactly the projection of $\pi_B(\gamma)$, the restriction of γ to the automaton B, onto B. By lemma 1, $\pi_B(\gamma)$ is a completed computation sequence of B, and therefore its restriction to B is a trace of B, by definition.

($\Leftarrow$) Suppose that for each node B in N, the projection of t onto B is a trace of B. Corresponding to each such trace, there is a completed computation sequence for that node. We dovetail among them, preserving the order of events in t, to obtain a computation sequence γ of N, such that the restriction of γ to any node B is completed, and hence γ is completed by lemma 1. ∎

We now make precise what we mean by "implementing" a primitive. We first define the input-output relation or IO-relation of a node or a network.

Definition 16. The **input-output relation** of a node or a network of nodes is the set of all pairs $(H_\sigma^{in}, H_\sigma^{out})$ with σ being a completed computation sequence of the automaton, H_σ^{in} is the input port history corresponding to σ, and H_σ^{out} is the output port history corresponding to σ.

We note here that internal events of an automaton are not represented in an input-output relation.

Definition 17. Let R be a binary relation between n-tuples of streams and m-tuples of streams. A set S of nodes is said to **implement** R if, using any finite number of copies of nodes in S, we can build a network with n input channels and m output channels with R as its IO-relation.

It is important to note that we include the empty sequence of values as a possible stream when we talk about the IO-relation. Thus, for example, if N has a trace in which no output is produced on any of the output channels

then we will not consider N to have implemented M unless it is possible for M to exhibit this behavior as well.

It is possible to have nodes with different sets of traces, but the same IO-relation. Brock and Ackerman [6] have such an example, but their example uses a powerful primitive, *fair merge*. There are other examples using only finite indeterminacy [30]. Equality of the IO-relation is a very crude notion of equality but any semantic theory must refine this equality. Thus our inexpressiveness results apply in any "reasonable" semantic theory. The notion of network equivalence that we are using is very weak indeed. It is not even a congruence with respect to network formation [6].

3.1 Activity Sequences and Traces

We would now like the concept of a sequence of events of a node, all of which are "under the control of" the node. In particular, we would view arrival events not to be "locally controlled" [18]. The definitions below makes this precise.

Definition 18. An event of a node is said to be **locally controlled**, if it is either an input event or an output event of the node.

Definition 19. An **activity sequence** of a node is the restriction of a trace of the node to its locally controlled events.

A couple of useful lemmas follow. Intuitively, the first lemma says that if we take an activity sequence, and add an arrival event before every input event, and maybe add extra arrival events, then the resulting sequence of events is a trace. The second lemma says that if we take a trace, and rearrange some of the arrival events, but maintaining the fact that every input event has its corresponding arrival event preceding it, then the new sequence is a trace. Intuitively the arrival events could occur any time before their corresponding read events.

Lemma 4. If t is an infinite activity sequence of a node, and t' is a sequence of events, such that t is the restriction of t' to locally controlled events, and further, $\Pi_{p,-}(t'')$ is a prefix of $\Pi_{p,+}(t'')$ for every prefix t'' of t' and every port p, then t' is a trace of the node.

Proof : We need to construct a completed computation sequence of the node such that the corresponding trace is t'. Let s,

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$$

be the completed computation sequence, such that t is obtained from the trace corresponding to s. We note that the assumption, $\Pi_{p,-}(t'')$ is a prefix of $\Pi_{p,+}(t'')$ for every prefix t'' of t' and every port p, ensures that all arrival events precede the corresponding input events. We consider the computation sequence s obtained by "pushing forward" all the arrival events, not in t', to the end of s, and thus deleting them. Then this new sequence s',

$$q'_0 \xrightarrow{a'_1} q'_1 \xrightarrow{a'_2} \ldots$$

is a computation sequence. If this sequence is not completed, then there exists an integer $i > 0$ and an event e that is not an arrival event, is enabled in q'_{i-1} and commutes with every event in s' from the ith event a'_i onwards. Since the events in s' are all the events in s with some arrival events deleted, there is a $j \geq i$, such that the sequence of events in s from the jth event onward consists of the events in s from the ith event onward, together with some arrival events. By monotonicity, e commutes with every input event. And since e commutes with every event in s' from the ith event onward, e commutes with every event in s from the jth event onward. By the monotonicity property, since e is enabled in the state q'_{i-1}, it is also enabled in q_{j-1}. This, together with the previous statement, contradicts the assumption that s is completed. It is now easy to see that by pushing arrival events forward or backward, we can get a computation sequence, the trace corresponding to which is t'. ∎

Lemma 5. If t is a trace of a node, and t' is a sequence of events, containing exactly all the events in t, and such that the restrictions of t and t' to locally controlled events are the same, and further, $\Pi_{p,-}(t'')$ is a prefix of $\Pi_{p,+}(t'')$ for every prefix t'' of t' and every port p, then t' is a trace of the node.

Proof : Let s be the completed computation sequence to which the trace t corresponds. The conditions of the lemma imply that for every input event in t', the corresponding arrival event in t' precedes it. So we can push arrival events forward and back to get a completed computation sequence, such that the corresponding trace is t'. ∎

3.2 Sequential Automata

In all our discussions on implementability, we assume that we have a certain base set of automata or nodes that we can use in any implementation. These are simply the abstractions of normal sequential programs, and we call these sequential automata. It is known that these compute sequential functions, as defined by Berry and Curien [16]. We will prove and use some properties of sequential automata in a crucial way in our proofs.

Definition 20. An automaton is said to be **sequential**, if for every state of the automaton, there is at most one event enabled, that is not an arrival event.

We will henceforth refer to this property of automata as *sequentiality*. We note here that arrival of input is always enabled.

Example 3. *Parallel OR* : This process has two input channels and one output channel. It computes the following function POR : POR(true,<>) = POR(<>,true) = POR(true,true) = true, POR(false,false) = false, and POR(<>,<>) = POR(<>,false) = POR(false,<>) = <>, where <> represents the empty sequence, and true and false are two data values. This process outputs true if it gets a true on either of its input channels. It outputs false if it gets a false on both of its input channels. If we suppose that there is a sequential automaton for this function, then by the definition of sequentiality, the first locally controlled event can either be a read from the first input channel or a read from the second input channel, but not both. For either case, an arrival of a true on the other input channel is enough to show that this automaton does not compute the parallel OR function.

Lemma 6. A sequential node has the following properties :

(i) If t and t' are two distinct activity sequences of the node, then either one is a prefix of the other or at the point where they first differ they each have an input event on the same input channel.

(ii) for any sequence of arrival events on input channels, there is a *unique* activity sequence.

Proof : (i) Let u and u' be two sequences obtained from t and t' by putting an arrival event of the same value before every input event in t and t'. Then, by lemma 4, u and u' are traces. Let s and s' be the completed computation sequences corresponding to these traces, and let us assume that every arrival event immediately precedes the corresponding input event, because we can always ensure this by pushbacks and pushforwards of input events. By the sequentiality of the node, there is no state in which there are two non arrival events enabled. Therefore s and s' must first differ at a state q_i where the next events in s and s' are both arrival events. If these are both arrival events on the same input channel, then the corresponding input events are different, and so the two activity sequences first differ at input events on the same input channel. Otherwise,

$$q_i \stackrel{a_{i+1}}{\rightarrow} q_{i+1} \stackrel{a_{i+2}}{\rightarrow} q_{i+2} \text{ and } q_i \stackrel{a'_{i+1}}{\rightarrow} q'_{i+1} \stackrel{a'_{i+2}}{\rightarrow} q'_{i+2}$$

where a_{i+1} and a'_{i+1} are arrival events on different input channels, and a_{i+2} and a'_{i+2} are the corresponding input events. Since arrival events on different input channels commute, by monotonicity, there is a state q, such that

$$q_{i+1} \stackrel{a'_{i+1}}{\rightarrow} q \text{ and } q'_{i+1} \stackrel{a_{i+1}}{\rightarrow} q$$

and both a_{i+2} and a'_{i+2} are enabled in q by monotonicity. This violates sequentiality, because now there are two non arrival events enabled in a state.

(ii) If there are two distinct activity sequences, then, by (i), they first differ at a point where each has an input event on the same input channel. But this is not possible, because both activity sequences correspond to the same sequence of arrival events. ∎

We will need to reason about sequences of events, and we sometimes require a continuity property for this. Under certain conditions, this property is preserved by network composition. Here we define the property, and prove that all sequential nodes have this property.

Definition 21. A set of sequences, Σ, is said to be **prefix continuous** if for an arbitrary infinite sequence t, if every finite prefix of t is a prefix of some member of Σ, then t is a member of Σ.

Definition 22. A node is said to be **prefix continuous** if its set of activity sequences is prefix continuous.

Lemma 7. Sequential nodes are prefix continuous.

Proof : Let t be an infinite sequence of events, such that every finite prefix of t is a prefix of an activity sequence. Let t' be an infinite sequence of events obtained by putting an arrival event of the same value immediately before every input event in t. Then we note that every prefix of t' is a prefix of a trace, using lemma 4, coupled with the fact that every finite prefix of t is a prefix of an activity sequence that we can "fill in" in the above manner. Let $s_1, s_2 \ldots$ be the computation sequences corresponding to these traces, extending the prefixes of t'. We assume that in each of these computation sequences, every arrival event immediately precedes the corresponding input event, because we can always ensure this by pushforwards and pushbacks.

Let u_i be the smallest prefix of s_i containing all the first i events of t'. Then we claim that u_i is a prefix of u_{i+1} for every $i > 0$. If not, suppose u_i and u_{i+1} first differ at some state, from which the next events in u_i and u_{i+1} are different. They cannot first differ at some event in t'. And these next events cannot both be non arrival events, because this would violate sequentiality. So one of the events must be an arrival event, and the other must be an internal event. But then the event following the arrival event in s_i is an input event, and the internal event must be enabled in the same state as the input event, by monotonicity, thus violating sequentiality.

Then the limit s' of the sequences $u_1, u_2 \ldots$ is a computation sequence. By sequentiality, it follows that there is no event that is continuously enabled from some point on and does not happen – if there was some continuously enabled event, then since t is infinite, there would be some state in which two non arrival events are enabled, violating sequentiality. So the computation sequence is completed, and therefore t' is a trace and t is an activity sequence. ∎

3.3 Choice Automata

The choice nodes described in the introduction may be easily defined formally. Essentially, arrival events are always enabled, and internal events trigger the output behaviour. All we will say here is that once an input event is executed, the corresponding output event must be executed before the next input event. This is to ensure that unfair choice is prefix

continuous.

We note that weakly fair choice is not prefix continuous. Let $a(v)$ represent the arrival of value v on the input channel. Let $i(v)$ represent the corresponding input event – the value v getting read. Let $o_1(v)$ and $o_2(v)$ represent the output events for outputting value v on the first and second output channel respectively. Then $i(v_1), o_1(v_1), i(v_2), o_1(v_2), i(v_3), o_1(v_3)\ldots$ is a sequence of events such that every prefix is a prefix of an activity sequence, but the whole sequence is certainly not an activity sequence, because there are infinitely many output events on the first output channel and no output events on the second output channel – this is not possible by the definition of weakly fair choice.

4 Scheduling

In this section we examine certain general properties of sets of traces. One important point is that when one combines two networks, without making any interconnections, we must ensure that the traces of the composite system contain events from each of the subnetworks. To this end we define a certain notion of "scheduled" trace. We then show that every trace has an observably equivalent scheduled trace. Intuitively, a trace is scheduled if every node is guaranteed to make progress in a bounded number of steps. This means that in all such traces, if an event is "enabled" at some point, then the event occurs in the trace within a bounded amount of time. We show that the set of scheduled traces is prefix continuous.

We begin with a lemma that pinpoints the problem that arises if we do not worry about scheduling.

Lemma 8. The set of traces of a network composed of sequential processes is not necessarily prefix continuous.

Proof: Consider a network with two disjoint subnetworks. Consider a prefix of a trace of the network such that one of the subnetworks, say the first, has an enabled event. It is possible to have a sequence of prefixes of traces in which only the second subnetwork makes progress. Since there are no assumptions on the relative speeds of processes, these are possible prefixes of traces of the entire network. The limit of this sequence is an

infinite sequence in which only the second subnetwork makes any progress; this is not a trace of the entire network. ∎

We now develop the machinery needed to avoid this problem. We denote the ith element in sequence t by $t[i]$. We write $t[1..m]$ to stand for the prefix consisting of the first m events of t. We sometimes refer to $t[i]$ as the element at *time* i.

Definition 23. An **incomplete** prefix of a trace is a prefix of a trace that is not itself a trace.

This means that an incomplete prefix of a trace is finite, and can be extended to a trace by adding events at its end. An incomplete prefix of a trace is certainly not $\stackrel{\sqsubset}{\sim}$-maximal. A prefix of a trace will be incomplete because there is some event that is enabled but that has not occurred as yet. The definition of a scheduled trace ensures that such enabled events happen "as soon as possible."

We first describe how to construct a well-founded partial order, that will represent the "causal" order between events in the trace. We then prove that every total ordering of this partial order is a trace, and one of these total orders corresponds to "scheduling" the events of the trace. We first define a relation $\prec_1$, and obtain the desired partial order $\prec$ as its reflexive transitive closure.

Definition 24. For a trace t of a network N, $t[i] \prec_1 t[j]$ if

(i) $t[i]$ and $t[j]$ are both arrival events or both input events or both output events for the same channel, and $i < j$.

(ii) $t[i]$ is an arrival event and $t[j]$ is the corresponding input event on the same channel.

(iii) $t[i]$ and $t[j]$ are locally controlled events for some node P in the network, and $i < j$.

Definition 25. $\prec = (\prec_1)^*$

We are not claiming here that we get exactly the causality relation between the events of a trace, but we do get a refinement of it, and this is sufficient for our purposes.

Lemma 9. $\prec$ is a well-founded partial order.

Proof : First, we prove that $t[i] \prec_1 t[j]$ implies that $i < j$. If $t[i] \prec_1 t[j]$ by cases (i) or (iii), then $i < j$ by definition. If $t[i] \prec_1 t[j]$ by case (ii), then $t[i]$ and $t[j]$ are corresponding kth events on some channel, for some $k > 0$. In this case, $i < j$ because, in any trace, the kth arrival event on any channel precedes the corresponding input event on that channel. Now it is immediate that $\prec_1$ is well-founded, due to the well-foundedness of the natural numbers.

$\prec$ is certainly reflexive and transitive by definition. Now $t[i] \prec t[j]$ implies that there is a finite chain $t[i] = t[i_1] \prec_1 t[i_2] \prec_1 \ldots \prec_1 t[i_k] = t[j]$, with possibly $k = 1$. Now, if $t[i] \prec t[j]$ and $t[j] \prec t[i]$, then if $i \neq j$, then $i < j$, and so $j \not< i$, contradicting $t[j] \prec t[i]$. Therefore $i = j$, proving antisymmetry.

Well-foundedness follows from the fact that an infinite descending $\prec$-chain would imply an infinite descending chain of natural numbers, which is impossible. ∎

We shall sometimes denote the partial order $\prec$ associated with trace t by $\prec_t$.

Lemma 10. If t is a trace of a finite network of sequential and choice nodes, then any total ordering of the partial order of t is a trace of the network.

Proof : Let t' be the trace to which the partial order $\prec$ corresponds. Let t_P and t'_P be the projections of t and t' respectively onto node P. Then, t'_P is a node trace, because t' is a trace. We note that, for any node P in the network, the locally controlled events for that node are totally ordered. Therefore, the restriction of t_P and the restriction of t'_P to the locally controlled events of P are the same. Moreover, for every input event in t_P, the corresponding arrival event precedes it, because this precedence holds in t', and so in the partial order. Therefore, it immediately follows by lemma 5, that t_P is a node trace. Therefore the projection of t onto every node is a node trace, and therefore, t is a network trace. ∎

To schedule a trace, we dovetail among the sequences of arrival events, read events and output events at the various channels of the network, making sure at each step, that when an event is considered to be the next event in the new trace, then all its predecessors in the partial order have

already been considered. Corresponding to each input channel and each internal channel, there is a sequence of arrival events and a sequence of read events associated with it. Corresponding to each output channel, there is a sequence of output events associated with it. Each such sequence is associated with an end of a channel, or a **port**. Note that there are two ports associated with an input channel, one port is associated with the end of the input channel at the node. The other port can be thought of as being associated with the "other end" of the input channel, where an "external agent" outputs values onto the channel.

Definition 26. Let N be a network with m ports. A **scheduled trace**, t, of N is one that satisfies the following property. Suppose that $t[1..n]$ projected onto a particular node, say A, is an incomplete trace of A, then there will be an event local to A, and not an input + event to A, by time $n + m$.

Note that if every prefix of a trace is scheduled then the entire trace is scheduled.

Definition 27. A **port order** of a finite network with m ports is defined to be a total ordering $p_0, p_1, p_2, \ldots p_{m-1}$ of the m ports of the network.

Definition 28. For any finite network with m ports, and any port order $\mathbf{o}$, the **scheduling operation** $S_{\mathbf{o}}$ is defined as follows : Let $\prec$ be the partial order of a trace t. Then $S_{\mathbf{o}}(t)$ is a total ordering of the events of t, such that the following holds. $S_{\mathbf{o}}(t)[1]$ is an event on the first of the ports $p_0, p_1, p_2, \ldots p_{m-1}$ such that it has no predecessor in the partial order. This is guaranteed by well-foundedness. If $S_{\mathbf{o}}(t)[i]$ is an event corresponding to port k, then $S_{\mathbf{o}}(t)[i+1]$ is an event on the first of the ports $k+1 \mathbf{mod} m, \ldots k$ such that each of its $\prec$-predecessors is in $S_{\mathbf{o}}(t)[1..i]$.

Lemma 11. If t is a trace of a finite network of sequential nodes and choice nodes with m ports, then $S_{\mathbf{o}}(t)$, for any port order $\mathbf{o}$, is a scheduled trace of the network.

Proof : $S_{\mathbf{o}}(t)$ is a total ordering of the partial order of the trace t. Hence, by lemma 10, it is a trace of the network.

Suppose the projection s of $S_{\mathbf{o}}(t)[1..i]$ onto a node A in the network is an incomplete trace of A. Then there is a next input $-$ event or output $+$

event for A, in $S_{\mathbf{O}}(t)$. If this event is an input $-$ event then the input $-$ event must be in $S_{\mathbf{O}}(t)$ by time $i+m$ by the definition of $S_{\mathbf{O}}$. Otherwise the event is an output $+$ event with all its $\prec$-predecessors in $S_{\mathbf{O}}(t)[1..i]$, and, by the definition of $S_{\mathbf{O}}$, it must be in $S_{\mathbf{O}}(t)$ by time $i+m$. ∎

Definition 29. Let $\prec$ be a partial order. An infinite sequence $\beta_1 \prec \beta_2 \prec \ldots$ is said to be **eventually increasing** if it is non-decreasing and there is no i, such that $\beta_i = \beta_{i+1} = \ldots$.

Lemma 12. For any finite network of prefix continuous nodes, the set of all scheduled traces is prefix continuous.

Proof : Suppose N is any network of prefix continuous nodes. Suppose that τ is an infinite sequence with every prefix being a prefix of a scheduled trace. Suppose that τ is not a trace of the network. Then the projection of τ onto the channels of some node A of the network is not a node trace. Let α_i be the projection of the prefix $\tau[1 \cdot \cdot i]$ of τ onto the channels of the node A. Then each α_i is a prefix of a node trace, and the limit of the α_i's is the projection of τ onto the channels of node A. Further, let β_i be the activity sequence corresponding to α_i, i.e. all events of α_i except the arrival events.

Case 1 : The β_i's form an eventually increasing sequence. Then, by prefix continuity, the limit of the β_i's is an infinite activity sequence of the node. Then, using lemma 4, the limit of the α_i's is a node trace, contradicting the assumption.

Case 2 : for some i, for all $j \geq i$, $\beta_j = \beta_i$. Since, by assumption, the projection of τ onto the channels of the node A does not form a node trace, some α_j for $j \geq i$ is an incomplete prefix of a node trace. Then, by the definition of scheduled trace, since $\tau[1..j+m]$ is a prefix of a scheduled trace, β_{j+m} extends β_j. This contradicts the assumption that $\beta_j = \beta_i$ for all $j \geq i$.

Thus τ is a trace of the network, and hence is scheduled, as every prefix of τ is a prefix of a scheduled trace. Therefore the set of scheduled traces is prefix continuous. ∎

5 Nonexpressibility of Fair Choice

The proofs of our first two theorems are based on the branching structure of tree representations of traces. The first theorem states that choice with

signal cannot implement weakly fair choice. One might be tempted to think that this amounts to an easy proof that discontinuous primitives cannot be implemented by continuous components. Unfortunately the set of all traces of a network composed of prefix continuous nodes is not prefix continuous. We can, however, construct a tree such that every path in it represents a scheduled trace of the network. A continuity argument can be applied to the set of scheduled traces.

Our main theorem states that there is no network consisting of finitely branching prefix continuous nodes and WCS nodes that implements SFC. This of course does not follow directly from continuity arguments, because WCS does not satisfy our continuity property. We consider a network that supposedly implements SFC. We express the set of scheduled traces of the network as the union of a countable family of trace sets. We show that the traces in each member of the family is prefix continuous. We build a tree representation of the traces in each family. We quotient the tree by contracting all edges that do not correspond to an output event on an output channel. Each quotiented tree is finitely branching. Finally we diagonalize to exhibit a possible output sequence of strong choice that is not produced by any trace of the network.

First we establish the required definitions and lemmas.

Definition 30. If C is a set of traces of a network, then $T(C)$ is the tree whose nodes are finite prefixes of traces in C and such that prefix s' is a child of prefix s just in case s' can be derived from s by adjoining a single event onto the end of s. We assume that each edge is labeled with the last event of the prefix of the descendant node.

We note that the set of sequences corresponding to the paths in the tree from the root is not necessarily equal to the set of traces C. All that can be said is that, for every sequence corresponding to a path in the tree from the root, every prefix of this sequence is a prefix of a trace in C.

Definition 31. A node P is said to be **finitely branching** if for any sequence of events t that is an incomplete prefix of a node trace for P, there are only finitely many locally controlled events that can be the next event after the sequence of events t in any node trace of P.

Note that there are clearly infinitely many arrival events that can be the

next event after the sequence of events t. The definition restricts the number of locally controlled events that can be the next event.

We note that all sequential primitives are finitely branching. Moreover, all our choice primitives are also finitely branching.

Lemma 13. If N is a network of finitely branching nodes then the tree of traces of N for a fixed input is finitely branching.

Proof : Suppose s is a prefix of a trace. Then the next event of the trace could be an input event on any of the finitely many input channels, or, an event corresponding to some node in the network. There are finitely many of these too, because there are finitely many nodes, and each node is finitely branching. Therefore, s has only finitely many children in the tree of traces of N for a fixed input at the input channels. So every vertex in the tree has finitely many children, i.e. the tree is finitely branching. ∎

The following theorem follows easily from Kœnig's Lemma.

Theorem 1. No network of finitely branching, prefix continuous nodes, including CS, can implement weakly fair choice.

Proof : Suppose there is a finite network of finitely branching, prefix-continuous nodes, implementing WFC. Then the network has one input channel and two output channels, corresponding to those for a WFC node. We fix the input stream to be $1, 2, 3 \ldots$. Then the first output channel c, say, of the network is guaranteed to have at least one value output on it. This value can be any positive integer.

Let S be the set of scheduled traces with respect to a port order. Every trace has a scheduling with respect to any port order. Therefore, for every possible output sequence of the network onto channel c, there is a scheduled trace in S which outputs that sequence on c. We consider the tree $T(S)$, and we prune every path at the first output event on that path onto channel c. By lemma 12, every path in the tree is a network trace, and so has an output event on c. Moreover, by lemma 13, $T(S)$ is finitely branching. Therefore the pruned tree is a finitely branching tree with no infinite paths. By Kœnig's lemma, the tree is finite. So there are finitely many leaves, i.e. finitely many possibilities for the first output event on channel c. This means that the network does not implement weakly fair choice – contradiction. ∎

The next theorem shows that strongly fair choice cannot be produced from a weakly fair choice even with a signal. The proof requires a diagonalization argument; cardinality or Kœnig's lemma arguments by themselves do not seem sufficient.

Theorem 2. No network of finitely branching, prefix continuous nodes and WCS nodes can implement SFC.

In order to prove this theorem we need several definitions and lemmas.

We make explicit the fact that WFC embodies a countable choice. First we give a definition for events that lead upto this countable choice.

Definition 32. Let N be any network and let t be any sequence of events in that network. i is said to be a **choice initiation time** for t if there is a weak choice or weak signal choice node n in N and a non-signal output channel c of n such that $t[i]$ is either

1. the first output event on c in t, or,
2. there is no output event on channel c prior to time i in t, and $t[i]$ is a $+$ event on the input channel of n.

Such a $t[i]$ is called an **initiation event.**

Definition 33. Let s be a finite sequence of events on a network N. An **initiation-free extension** of s is a sequence t such that s is a prefix of t and such that all initiation events in t occur in the initial sequence s.

The next lemma follows from the definition of weak fair choice.

Lemma 14. For any trace t of a network, there exists some finite prefix s of t such that t is an initiation-free extension of s.

Proof : We show that each weak choice or weak signal choice node in the network has a last choice initiation time.

Let t' be the projection of t onto the channels of a weak choice or weak signal choice node.

(i) there are output events on both the output channels (both the non-signal output channels in the case of a weak signal choice node). In that case, the first output event on the first output channel is the mth event, say, in t, and the first output event on the second output channel is the m'th event, say, in t. Then the last choice initiation time for this node is $\max(m, m')$.

(ii) there are no output events on one of the output channels. Then there must be only finitely many input + events in t'. Let the last of these be the mth event in t. Also let the first output event on the other output channel be the m'th event in t. Then the last choice initiation time for this node is $\max(m, m')$.

Since there are finitely many weak choice and weak signal choice nodes, if i is the maximum of their last choice initiation times, then t is an initiation-free extension of $t[1], \ldots t[i]$. ∎

Definition 34. Let N be a network and s a finite sequence. We define C_s to be the set of all scheduled traces of the network, that are initiation-free extensions of s.

Lemma 15. For any finite network N, there are countably many sets of the form C_s, for s any finite sequence of events of the network.

Proof : Each event is a triple, and there are countably many of these, assuming that there are countably many values that may be transmitted on a channel. Therefore, there are countably many finite sequences of events, and so there are countably many sets of the form C_s. ∎

Note that even though every member of C_s is an initiation-free extension of s, every path in $T(C_s)$ is not necessarily a member of C_s, and so not necessarily an initiation-free extension of s. So the following lemma is required.

Lemma 16. For any finite network N, and any sequence of events s of the network, any path in $T(C_s)$ is an initiation-free extension of s.

Proof : Let t be a sequence such that every prefix of t is a prefix of some member of C_s. Since every trace in C_s starts with the prefix s, t must also

start with the prefix s. It follows from the definition of an initiation time that if i is an initiation time in t, and a trace t' is identical to t up to and including the ith event, then the ith event is also an initiation event in the trace t'. Hence this ith event is in s. Since every prefix of t is a prefix of some trace t' in C_s, t cannot contain any initiation-events other than events in the prefix s. ∎

Lemma 17. For any network N of prefix continuous nodes, weak choice nodes, and weak signal choice nodes, and for any finite sequence of events s, C_s is prefix continuous.

Proof : Let τ be a sequence such that every prefix of τ is a prefix of some member of C_s. By lemma 16, τ is an initiation-free extension of s. We must show that τ is a network trace.

Suppose that τ is not a network trace. Then the projection of τ onto the channels of some node A of the network is not a node trace. We will proceed as in lemma 12. Let α_i be the projection of the prefix $\tau[1..i]$ of τ onto the channels of the node A. Then each α_i is a prefix of a node trace, and the limit of the α_i's is the projection of τ onto the channels of node A. Further, let β_i be the activity sequence corresponding to α_i, i.e. all events of α_i except the arrival events.

Case 1 : The β_i's form an eventually increasing sequence. Then, if the node A is not a weak choice or a weak signal choice node, then, by prefix continuity, the limit of the β_i's is an infinite activity sequence of the node. Then, using lemma 4, the limit of the α_i's is a node trace, contradicting the assumption.

If the node A is a weak choice or a weak signal choice node, then the limit of the β_i's is an infinite sequence, containing infinitely many input events and infinitely many output events. Since this is not an activity sequence of the node, it must be the case that all the output events are on the same output channel, contradicting the requirement that there be output events on both the output channels if the input is infinite. This means that the projection of τ onto node A has infinitely many arrival events for A, and all of these are initiation events for τ. This contradicts the fact that s is finite, and τ is an initiation-free extension of s. Therefore this case cannot happen.

Case 2 : for some i, for all $j \geq i$, $\beta_j = \beta_i$. Since, by assumption, the projection of τ onto the channels of the node A does not form a node trace,

some α_j for $j \geq i$ is an incomplete prefix of a node trace. Suppose that the network has m ports. Then, by the definition of scheduled trace, since $\tau[1..j+m]$ is a prefix of a scheduled trace, β_{j+m} extends β_j. This contradicts the assumption that $\beta_j = \beta_i$ for all $j \geq i$.

Thus τ is a trace of the network, and hence is scheduled, as every prefix of τ is a prefix of a scheduled trace. As noted earlier, by lemma 16, τ is also an initiation free extension of s. Therefore τ is a member of C_s, and so C_s is prefix continuous. ∎

Definition 35. The **complement** of an increasing infinite sequence **S** of positive integers is defined to be the increasing sequence of all those positive integers that are not in the sequence **S**.

We now define a quotienting operation on trees that conceals events that are not output events on a fixed channel.

Definition 36. Let T be a tree in which the edges are labeled events from a network N. Let c be a channel of N. We define the **quotient** of T with respect to c, written T/c, to be the tree obtained by contracting every edge in T that is not an output event on c.

Proof of Theorem 2: Suppose there is a network of finitely branching, prefix continuous nodes and WCS nodes, implementing strong choice. Then the network has one input channel and two output channels. We fix the input stream to the network to be $1, 2, 3, ...$ Then the first output channel c, say, of the network has all increasing infinite sequences of positive integers, whose complements are also increasing infinite sequences of positive integers, as possibilities.

Since every trace has a scheduled trace that is locally equivalent to it, every possible output sequence of the network onto channel c is output by some scheduled trace.

Let S be the set of all scheduled traces of the network. We divide S into subclasses C_s, as defined earlier. We obtain a countable family of trees $T(C_s)/c$.

We claim that each tree $T(C_s)/c$ is finitely branching. Every path in $T(C_s)$ has infinitely many output events on channel c, since every path in the tree is a network trace by lemma 17. Consider any node n of the tree such that the prefix associated with that node ends in an output event on

c. These are exactly the nodes that remain after the quotienting. We prune every path from n at the first output event on channel c. Since the tree $T(C_s)$ is finitely branching the pruned tree below n is also finitely branching and has no infinite paths. By Kœnig's lemma, the tree is finite. So there are finitely many leaves. Thus in the quotiented tree, n has finitely many children corresponding to the finitely many leaves of the above pruned tree.

We name these quotiented trees $T_1, T_2, \ldots$. Each path in any of these trees must correspond to an infinite increasing sequence of positive integers. To obtain a contradiction, we construct, by diagonalization, an infinite increasing sequence of positive integers with infinite complement, that will be in none of these trees. Since every tree, T_i, is finitely branching, every level of each T_i has finitely many vertices. Hence, there is a maximum positive integer that occurs at that level. Let this maximum positive integer for the jth level in the ith tree be called $M_{i,j}$. We define $s[1]$, the first element of the sequence being constructed, to be any positive integer greater than $M_{1,1}$, say $M_{1,1}+1$. Having fixed the elements $s[1], s[2], \cdots, s[i-1]$, we define $s[i]$ to be any positive integer greater than $\mathbf{max}\{M_{i,i}, s[i-1]+1\}$. This is certainly an infinite increasing sequence. Moreover, between any two consecutive elements $s[i-1]$ and $s[i]$ of the sequence, there is at least one positive integer not in the sequence, namely $s[i-1]+1$. So the sequence has an infinite complement. But this sequence is not in any of the trees $T_1, T_2, \cdots$. This is because, for any i, the ith element of the sequence is greater than $M_{i,i}$, and this is the greatest integer at the ith level of T_i.

This means that there is an infinite increasing sequence of positive integers with infinite complement, that is not a possible output sequence at channel c for the network. Hence the network could not have implemented strong choice. ∎

6 Nonexpressibility of Signaling

In this section we explore the nonexpressibility arising from the sequentiality of the individual processes. Understanding sequentiality is a fundamental concern in the semantics of modern programming languages [16,28]. Our results in this section may be viewed as a first step towards understanding how sequentiality interacts with indeterminacy. The main theorem states that one cannot obtain a choice with a signal from an ordinary choice.

The point is that the signal channel is guaranteed to have as many tokens output on it as there are inputs. Unfair choice has no output channels on which a stipulated number of tokens are guaranteed to appear. The only nodes on which one can guarantee that a certain number of tokens will appear is a sequential node. In this case, however, the output values are determined by the input values. We show that this argument extends to networks composed of choice nodes and sequential nodes. It turns out that the theorem holds for weakly fair choice as well but not for strongly fair choice. Thus the result is quite delicate and somewhat counterintuitive.

Definition 37. Suppose that N is a network and c is a channel of N. Let t be a trace of N. A triple $\langle \pm, c, n \rangle$ is said to **occur** at time i in trace t if $t[i]$ is the nth arrival event or nth input event respectively on channel c.

We may also say $\langle \pm, c, n \rangle$ occurs in t if it occurs at some time in t. Note that a triple is not the same as an event. A triple represents an event in a trace.

Definition 38. Suppose that N is a network and c is a channel of N. Let R be a subset of the traces of N. We say that a triple $\langle \pm, c, n \rangle$ is **guaranteed in** R if it occurs in every trace in R.

Definition 39. Suppose that N is a network and c is a channel of N. Let R be a subset of the traces of N. We say that a triple $\langle \pm, c, n \rangle$ is **determined in** R if

$$\forall t_1, t_2 \in R.\ \langle \pm, c, n \rangle \text{ occurs at } i \text{ in } t_1 \text{ and at } j \text{ in } t_2$$
$$\Rightarrow t_1[i] = t_2[j].$$

We use the following notation. If t is a sequence of events, and c is a channel of a network, then we define $\Pi_c(t)$ to be the subsequence of t consisting of all the arrival events on channel c. If t is a sequence of events, and P is a node of a network, then we write $\Pi_P(t)$ to be the subsequence of t consisting of the locally controlled events of P.

The following is the central lemma of this section.

Lemma 18. For any network N of sequential nodes and unfair choice nodes, if R is the set of all network traces with a particular input I, then every triple $\langle \pm, c, n \rangle$ that is guaranteed in R is determined in R.

Proof: The proof proceeds by induction on the earliest occurrence of a guaranteed triple. Suppose that $\langle +, c, n\rangle$ occurs at time 1 in a trace t. Then clearly $n = 1$. Also c has to be either the output channel of a sequential node or an input channel of the network. In the first case it is clearly determined by sequentiality, in the second case it is determined since we are considering a fixed input.

Suppose that the lemma holds for all triples that have an earliest occurrence time less than k in R. Let $\langle -, c, n\rangle$ be a guaranteed triple with earliest occurrence equal to k. Then $\langle +, c, n\rangle$ has an earliest occurrence time $< k$, hence it is determined, by the induction hypothesis. Therefore, $\langle -, c, n\rangle$ is determined also.

The other case is that a triple $\langle +, c, n\rangle$ has an earliest occurrence time equal to k in R. Suppose that this triple is not determined in R. Then there are two traces g and h such that $\Pi_c(g)[n] \neq \Pi_c(h)[n]$. Since they differ, c cannot be an input channel of the network. Because the triple $\langle +, c, n\rangle$ is guaranteed, c cannot be the output channel of an unfair choice node. Thus c must be the output channel of a sequential node A. Without loss of generality, we can assume g to be the trace in which $\langle +, c, n\rangle$ occurs at time k. Let the sequence $\Pi_A(g)$ be s and the sequence $\Pi_A(h)$ be s'. They differ and, since A is sequential, by lemma 6, they must first differ at the same triple $\langle -, d, m\rangle$ where d must be an input channel of A. Clearly $\langle -, d, m\rangle$ has an earliest occurrence time less than k since it occurs before $\langle +, c, n\rangle$ in g. If $\langle -, d, m\rangle$ is guaranteed then, by the induction hypothesis, it must be determined, which is a contradiction. The remaining possibility is that $\langle -, d, m\rangle$ is not guaranteed. Consider the first non-guaranteed triple, $\langle \pm, p, l\rangle$ that occurs in s. We know that such a triple exists in s since $\langle -, d, m\rangle$ is a non-guaranteed triple that does occur in s. Since the triple $\langle \pm, p, l\rangle$ is not guaranteed, there is another trace, r, in which there are exactly $l-1$ events on channel p and $\Pi_A(r)$ agrees with s upto the occurrence of $\langle \pm, p, l\rangle$ in s. Since A is sequential, there are no more locally controlled events of A in r. Thus the triple $\langle +, c, n\rangle$ does not have an occurrence in r, which is also a contradiction. Thus $\langle +, c, n\rangle$ must be determined. ∎

Theorem 3. No finite network of sequential nodes and unfair choice nodes can implement choice with signal.

Proof : Suppose there is a finite network N showing this implementation. Let c be the signal output channel in this implementation. Let the input

stream to the network be a single element, and suppose R is the set of network traces with this particular input. Then at least one event is guaranteed on channel c in every trace in R. Moreover, it is the case that this first event on channel c could be $\langle c, 0, + \rangle$ or $\langle c, 1, + \rangle$. This contradicts the earlier lemma. ∎

The following theorem is the extension to the case where we allow weakly fair choice instead of unfair choice.

Theorem 4. No finite network of sequential nodes and weakly fair choice nodes can implement choice with signal.

In order to prove this theorem, we need several definitions and lemmas.

Definition 40. Let N be a finite network and s a finite sequence of events. We define $C'_{s,I}$ to be the set of all traces of the network for a particular input I, that are initiation-free extensions of s.

Lemma 19. For any network N of sequential nodes and weakly fair choice nodes, then every triple $\langle \pm, c, n \rangle$ that is guaranteed in $C'_{s,I}$ is determined in $C'_{s,I}$.

Proof : The proof proceeds exactly as in lemma 18, except for the following case. $\langle +, c, n \rangle$ has an earliest occurrence time equal to k in $C'_{s,I}$, and all triples with earliest occurrence times less than k are guaranteed by the induction hypothesis. Suppose that this triple is not determined in $C'_{s,I}$. Then there are two traces g and h such that $\Pi_c(g)[n] \neq \Pi_c(h)[n]$. We consider the case where c is an output channel of a weakly fair choice node. In that case, $n = 1$, because only one event is guaranteed on an output channel of a weakly fair choice node. Therefore this is an initiation event, and so must be in s. Hence g and h cannot disagree on $\langle +, c, n \rangle$ because both g and h have the same prefix s, and this contradicts the supposition that the triple is not determined.

The rest of the cases are exactly as in lemma 18. ∎

Proof of theorem 4 : Suppose there is a finite network that is supposed to implement WCS. Let c be the signal output channel in N. Let the input stream to N be some infinite stream I. This guarantees that the output stream on c is infinite for every network trace.

Every network trace is in some class $C'_{s,I}$, as in lemma 14. Moreover, every trace in $C'_{s,I}$ has the same output on channel c. This is because, since the input I is infinite, there are infinitely many events guaranteed on channel c, and by lemma 19, they are all determined.

As in lemma 15, there are countably many such classes $C'_{s,I}$, and so for the input I, there are at most countably many different outputs on channel c. But, by the definition of a signal-choice node, for an infinite input, there are uncountably many output stream possibilities for the signal output channel. This means that the network N does not implement a signal-choice, which contradicts the assumption that it does. ∎

7 Implementability Results

The main expressiveness results in this paper refer to non-implementability, using a weak notion of implementation. In this section we establish the positive implementation results for the primitives that we have introduced. We do not intend that a great deal of importance be attached to these results, precisely because we use a relatively weak notion of process equivalence. We do expect, however, that these implementation results would be true with a sharper notion of process equivalence. In the present context, the main point of these results is to establish there are no more negative results possible; in other words we have completely settled the expressibility situation for choices and signals.

It is trivial to observe that CS can implement choice, WCS can implement WFC, and SCS can implement SFC – one just hides the signal output channel. We will sketch the ideas showing that strong choice can implement weak choice, and that weak choice can implement choice.

Fact 1. The infinite stream $\mathcal{Z}^+$ of positive integers in increasing order can be generated by a simple deterministic sequential process.

Lemma 20. SFC can implement WFC.

Proof : We send $\mathcal{Z}^+$ into the input channel of a strong choice process. Then each output stream will be an infinite increasing sequence of positive integers, by the definition of strong choice. We now use one of these output streams as an "oracle". A deterministic Kahn process then uses this oracle

stream to decide whether either of the output streams of the weak choice should be finite, and if a stream should be finite, to decide which elements of the input stream should comprise the finite output stream. ∎

Lemma 21. WFC can implement choice.

Proof : As in lemma 20, we send $\mathcal{Z}^+$ into the input channel of a weak choice process. Then the first element of the first output stream is an unbounded positive integer. We use a finite number of weak choice processes to get unbounded positive integers. Using these, a deterministic Kahn process can then decide if either output stream of the choice should be empty, in which case the entire input stream should comprise the other output stream. Otherwise the process just uses a weak choice process to decide the distribution of input stream elements. ∎

There are analogous lemmas to prove that SCS can implement WCS, and that WCS can implement CS.

A surprising result is :

Theorem 5. SFC and SCS are implementation equivalent.

Proof : We trivially observed earlier that SCS can implement strong choice. To prove that strong choice can implement SCS, we first use a SFC process, as before, to get an "oracle" i.e. an infinite increasing sequence of positive integers. A determinate Kahn process with three output channels and two input channels can read this sequence and the input stream. The process puts the ith input stream element on the first output channel if $i \in$ oracle stream, and puts the ith input stream element on the second output channel if i does not belong to the oracle stream. This is decidable because if i does not belong to the oracle stream, an integer $> i$ will be read from the oracle stream. The process also sends the appropriate signal 0 or 1 onto the third output channel. ∎

We now state a few corollaries, that, together with the theorems proved, completely describe the situation between the six choice nodes.

Corollary 1. Choice cannot implement WFC.

Proof : If not, then since CS can implement choice, CS can then implement WFC. This contradicts theorem 1. ∎

Corollary 2. Choice cannot implement WCS and strong signal-choice.

Proof : If not, then since both WCS and strong signal-choice can implement CS, choice can then implement CS. This contradicts above corollary. ∎

Corollary 3. WFC cannot implement WCS.

Proof : If not, then since WCS can implement CS, weak choice can then implement CS. This contradicts theorem 2. ∎

Corollary 4. WFC cannot implement strong choice.

Proof : If not, then since strong choice can implement WCS, weak choice can then implement WCS. This contradicts above corollary. ∎

In this paper, we have concentrated on choice primitives, but these came out of a study of merge primitives, in particular, the study of the power of "fair merge" [23,26,27]. One of the other merge primitives, proposed by Park, is called **infinity-fair merge**. Our interest in this primitive in the context of this paper is that this merge primitive is implementation equivalent to strong choice. So a primitive that merges two streams is seen to be equivalent to a primitive that splits a stream into two streams.

An infinity-fair merge has two input channels and one output channel. If both the input streams are infinite, then the process outputs every element of each of the two input streams, that is, the merge is fair. If either of the two input streams is finite, then the output stream is finite, and either every element of the first input stream is output, or every element of the second input stream is output.

Lemma 22. Strong choice is implementation equivalent to infinity-fair merge.

Proof : To prove that strong choice can implement infinity fair merge, we first use a SFC process, as before, to get an "oracle" i.e. an infinite increasing sequence of positive integers. A determinate Kahn process with three input channels and two output channels can read this sequence and the input streams. The process makes the ith output stream element, one from the first input channel if $i \in$ oracle stream, and makes the ith output stream element, one from the second input channel if i does not belong to

the oracle stream. This is decidable because if i does not belong to the oracle stream, an integer $> i$ will be read from the oracle stream.

To prove that infinity fair merge can implement strong choice, we let the two input streams of an infinity fair merge be an infinite stream of 0's and an infinite stream of 1's. The output of this infinity fair merge is read by a deterministic Kahn process that counts the sizes of blocks of 1's that appear between blocks of 0's. If these sizes are $a_1, a_2, a_3 \ldots$, then the output of this process is $a_1, a_1 + a_2, a_1 + a_2 + a_3 \ldots$. ∎

Similar to weak choice, one could think of **weak-j choice** processes, which all behave like a weak choice process, except that for an infinite input stream, it guarantees j data values on each output channel. We then have infinitely many processes, weak-j choice, for j = 1,2, ... But the following lemma says that all these processes are implementation-equivalent.

Lemma 23. Weak choice (or weak-1 choice) is implementation equivalent to weak-j choice, for all positive integers j.

Proof : We sketch the implementation both ways here. Given a weak-j choice process, send $\mathcal{Z}^+$ onto the input channel. A couple of deterministic Kahn processes can then each read one element from the two output channels of the weak-j choice process. At least one data value is ensured on each output channel for $j \geq 1$. The processes then ignore the next $j-1$ elements on each output, and use the rest of the output streams of the weak-j choice process to simulate a choice process.

For the other way, use enough weak choice processes (at most $2j$ needed) with input streams $\mathcal{Z}^+$ to get the positions in the input sequence of $j-1$ elements on either output. This is possible because from each weak choice process, we can get an unbounded positive integer, as described in an earlier lemma. Then use a weak choice process to get the rest of the output. ∎

8 Conclusion

In this paper we establish new expressiveness results of indeterminate dataflow primitives. Our work, and other recent results [23,26], shows that there is a much richer hierarchy of expressiveness properties than had been suspected. Our next two goals are to characterize the class of relations that

are computed by networks containing various indeterminate primitives and to develop appropriate semantic theories for such networks.

Recent work by E. W. Stark [33] shows that one can view all the primitives discussed in this paper as exhibiting "internal" nondeterminism. In other words all the indeterminacy can be taken as arising from internal choices made by the individual processes. He uses this to characterize the class of relations computed by such processes. We are looking for a similar characterization of the relations computed by processes that can signal. We hope to shed light on the elusive concept of sequentiality [16] in the course of this study.

Gilles Kahn has developed a very pleasant semantic theory of determinate networks where one can use a simple fixed-point principle to compute the behaviour of systems. Having a fixed-point principle amounts to being able to use induction when reasoning about such systems. Given how complex the purely operational formalisms can be when one deals with indeterminacy and concurrency, it is important to have semantic principles that can be used reason about system behaviour. Recently, Rabinovich and Trakhtenbrot [29] have characterized those networks for which a fixed-point principle applies. We are seeking to extend the domain of applicability of such principles. We are using category-theoretic techniques [21,22] and techniques based on new powerdomains [11] in order to formulate such principles.

Acknowledgements

We would like to thank Keshav Pingali, Radhakrishnan Jagadeesan and Jim Russell for helpful discussions. This research was suported in part by NSF grant DCR-86-02072.

References

[1] I. J. Aalbersberg and G. Rozenberg. Theory of traces. *Theoret. Comput. Sci*, 60(1):1–82, 1988.

[2] S. Abramsky. On semantic foundations for applicative multiprogramming. In J. Diaz, editor, *Proceedings of the Tenth International Con-*

ference On Automata, Languages And Programming, pages 1–14, New York, 1983. Springer-Verlag.

[3] K. R. Apt and E.-R. Olderog. Proof rules and transformations dealing with fairness. *Sci. Comput. Prog.*, 3:65–100, 1983.

[4] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal Of The ACM*, 33(4):724–767, 1986.

[5] R. J. Back. A continuous semantics for unbounded non-determinism. *Theoretical Computer Science*, 23(2):187–210, 1983.

[6] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

[7] M. Broy. Fixed point theory for communication and concurrency. In *Formal Description of Programming Concepts II*, pages 125–148. North-Holland, 1983.

[8] A. Chandra. Computable non-deterministic functions. In *Proceedings of the 19th Annual Symposium of Foundations of Computer Science*, pages 127–131, New York, 1978. IEEE.

[9] N. Francez. *Fairness.* Springer-Verlag, 1986.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985.

[11] R. Jagadeesan. L-domains and lossless powerdomains. In *Proceedings of the Fifth Conference on Mathematical Foundations of Programming Semantics*, 1989.

[12] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Uppsala University, 1987.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[14] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In *Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[15] R. M. Keller and P. Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[16] P l Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. John Wiley and Sons, 1986.

[17] N. A. Lynch and E. W. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 1989.

[18] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, M. I. T. Laboratory for Computer Science, April 1987.

[19] A. Mazurkiewicz. *Advanced Course in Petri Nets*, volume 255 of *LNCS*, chapter Trace Theory, pages 279–324. Springer-Verlag, 1986.

[20] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[21] P. Panangaden. Abstract interpretation and indeterminacy. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 497–511, 1985. LNCS 197.

[22] P. Panangaden and J. R. Russell. A category-theoretic semantics for unbounded indeterminacy. In *Proceedings of the Fifth Conference on Mathematical Foundations of Programming Semantics*, 1989.

[23] P. Panangaden and V. Shanbhogue. On the expressive power of indeterminate primitives. Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[24] P. Panangaden and V. Shanbhogue. Mccarthy's amb cannot implement fair merge. In *Proceedings of the Eighth FSTTSC Conference*, pages 348–363, 1988. LNCS 338.

[25] P. Panangaden and V. Shanbhogue. Traces are fully abstract for networks with fair merge. unpublished manuscript, 1989.

[26] P. Panangaden and E. W. Stark. Computations, residuals and the power of indeterminacy. In Timo Lepisto and Arto Salomaa, editors, *Proceedings of the Fifteenth ICALP*, pages 439–454. Springer-Verlag, 1988. Lecture Notes in Computer Science 317.

[27] D. Park. The "fairness problem" and non-deterministic computing networks. In *Proceedings of the Fourth Advanced Course on Theoretical Computer Science, Mathematisch Centrum*, pages 133–161, 1982.

[28] Gordon Plotkin. Lcf considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[29] A. Rabinovich and B. A. Trakhtenbrot. Nets of processes and dataflow. Manuscript, 1988.

[30] J. Russell. Private communication, 1988.

[31] E. W. Stark. Semaphore primitives and starvation-free mutual exclusion. *Journal of the ACM*, 29(4):1049–1072, 1982.

[32] E. W. Stark. Concurrent transition system semantics of process networks. In *Proceedings Of The Fourteenth Annual ACM Symposium On Principles Of Programming Languages*, pages 199–210, 1987.

[33] E. W. Stark. On the relations computable by a class of concurrent automata. Manuscript in preparation, 1988.

[34] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 1989.