# The Expressive Power of Indeterminate Dataflow Primitives

Prakash Panangaden[*]

Vasant Shanbhogue[†]

Computer Science Department, Cornell University

November 16, 2012

**Abstract**

We analyze the relative expressive power of variants of the indeterminate fair merge operator in the context of static dataflow. We establish that there are three different, provably inequivalent, forms of unbounded indeterminacy. In particular, we show that the well-known fair merge primitive cannot be expressed with just unbounded indeterminacy. Our proofs are based on a simple trace semantics and on identifying properties of the behaviors of networks that are invariant under network composition. The properties we consider in this paper are all generalizations of monotonicity.

## 1 Introduction

The study of indeterminate computing systems is motivated by seeking to understand the theory underlying what is commonly called "distributed systems" programming. In such systems indeterminacy is introduced when one abstracts away from low level hardware or timing details. The subject receives further impetus from recent interest in parallel and distributed computing. From the purely theoretical point of view the subject is interesting because of the presence of new fundamental concerns, for example, deadlock and fairness, that have no analogue in sequential, determinate programming.

[*]Present Address: School of Computer Science, McGill University, Montreal, PQ, CANADA

[†]Present Address: Department of Computer Science, Wichita State University, Wichita, Kansas.

The foundational work in the theory of determinate distributed systems is due to Gilles Kahn [11]. His model of distributed systems is as follows. A system is viewed as a collection of autonomous computing agents communicating by one-way, unbounded data channels. Each computing agent executes its own program and, from time to time, may communicate with other agents by sending data tokens along the channels or reading tokens from incoming channels. If an agent attempts to read a token from an incoming channel that has no data it suspends until data becomes available. Under these conditions an agent computes a continuous function from input token streams to output token streams. One can thus model agents as functions. If one has a network of such agents then one can write down a system of equations to model the behavior of the network. If the network has feedback loops the system of equations will be recursive. Such systems of equations can be solved using elementary fixed-point theory.

Much of the research in indeterminate dataflow is aimed at developing a theory of comparable elegance and utility as Kahn's was for determinate dataflow. The contribution of this paper can be seen as a step towards such a goal as it provides a classification of the types of indeterminate primitives available and shows that monotonicity properties may be violated by common indeterminate primitives.

In this paper we analyze fairness properties of indeterminate dataflow networks. Fairness has been the paradigmatic liveness property and has been studied extensively in a variety of formalisms with several different definitions; see, for example, the recent book by Francez [9]. It is known that fairness introduces unbounded indeterminacy (or countable nondeterminism), the proof is by a simple Koenig's lemma argument. Plotkin's pioneering study of powerdomains for indeterminacy included the observation that the powerdomain that he introduced had been specifically designed for bounded indeterminacy and therefore excluded the study of fair systems [24]. Several people have worked on generalizations of powerdomain techniques that would apply to unbounded indeterminacy [2, 5, 7, 22, 23, 25].

Till recently fairness had been identified with unbounded indeterminacy [4]. Considerable effort had been expended in formalizing semantics of dataflow networks that would include a satisfactory treatment of the fair merge primitive [7, 8, 13, 20, 22]. All these efforts seemed to take the view that this was the next natural step after Plotkin's work. In the present paper we demonstrate that there are in fact three "levels" of expressive power all embodying unbounded indeterminacy. (Recent work has shown that there are, in fact, even more [19].) More importantly, from a semanticist's point of view, we show that there are fundamentally different order-theoretic properties corresponding to these levels of expressive power. Thus different semantic theories may be appropriate at these levels. Fair merge, in particular, violates basic monotonicity properties. An important consequence of our proof is that fair merge *cannot* be viewed as being a determinate process with some inputs concealed – a so-called "oracle driven" primitive. Some of the expressiveness results reported here were announced earlier by Panangaden and Stark using an operational approach [21]. Many of the results, particularly those relating to monotonicity problems, were observed by Broy [7], though he did not provide proofs. Smyth had earlier observed that McCarthy's **amb** is not monotone when viewed as a function on the Plotkin powerdomain [26].

The three levels that we consider in this paper are expressed as fairness properties of dataflow merges. A merge consists of two input ports and a single output port. Tokens arriving at the input ports are transmitted unaltered to the output port, the relative ordering of tokens on the input port is preserved on the output port but tokens arriving at different input ports may be output in either order at the output port. A *fair* merge will transmit all the tokens that appear at its input ports. An *angelic* merge will transmit all the tokens at a given port if the sequence of tokens appearing at the *other* port is finite. Thus an angelic merge will behave like a fair merge if both input sequences are finite. An *infinity-fair* merge has the dual property. If the sequence of tokens at one of the input ports is infinite all the tokens at the other port will be transmitted. Thus an infinity-fair merge will behave like a fair merge if both input sequences are infinite. The main expressiveness result is that fair merge cannot be implemented by angelic merge and that angelic merge cannot be implemented by infinity-fair merge. It is easy to implement angelic merge with fair merge and a construction by Eugene Stark [33] shows how to implement infinity-fair merge with angelic merge. Precise definitions of all these primitives and of "implements" will be given in later sections. The terminology for the merge primitives is due to Park [22].

Our entire analysis is based on the description of a process as a set of possible behaviors. We use *traces* as abstractions of process behavior. By traces we simply mean sequences of events on the ports of the processes forming a dataflow network. Traces have been studied at various levels of mathematical rigor and have a rich mathematical theory [1]. The order-theoretic properties that we alluded to above are formulated in terms of traces rather than in terms of relations. It would have been preferable if we could have used purely extensional properties like the input-output relation, but the well-known Brock-Ackerman example shows that the input-output relation computed by a network is not compositional [6].

It turns out that traces are not only compositional but are *fully abstract*. In the first half of the paper we describe an operational semantics for networks and use it to show that traces do indeed give rise to a compositional semantics of dataflow networks. The operational semantics that we describe is closely modeled on work of Stark [21, 30, 31, 33] and also on the IO automata formalism of Lynch and Tuttle [17]. Our expressiveness results do not require that traces be fully abstract, so we do not present a proof here, but a proof of this result for a slightly different operational semantics may be found in Jonsson's paper [10] that used the suggestions of Joost Kok [15]. It is important to be aware that we (and Kok and Jonsson) view the complete output in response to a given (possibly infinite) input stream to be observable. This is, of course, a rather liberal view of observable. It is only with this rather strong notion of observable that we can say interesting things about fairness. There is a fascinating study, by Rabinovich and Trakhtenbrot, of the semantics of dataflow networks with finite observations only permitted [27].

We would like to emphasize that there are three abstractions of processes that are used in this paper – (a) the automata, which describe the operational semantics, (b) trace sets, which are compositional and on which the monotonicity properties are defined, and (c) input-output relations, that fail to be compositional. In fact, our proofs based on trace sets do not rely on the specific operational semantics, once the connection between automata and trace sets has been established. We have included the details of the operational semantics here in order to make the paper more self-contained.

## 2 Operational Semantics of Dataflow Processes

The operational semantics of dataflow networks is given in an automata theoretic formalism in which the notion of causal independence between concurrent events is taken to be primitive. This is based on the work of Lynch and Tuttle [17, 16] and Stark [30, 31]. Each process has a set of events associated with it. We represent concurrency by a binary relation on events, that tells us when two events are independent. This is the abstraction of describing when two commands in a program can be executed in parallel. This binary relation, which is called the *concurrency relation*, is axiomatized via equations that express the fact that the order of execution of concurrent events can be permuted, thus capturing the causal independence of concurrent events.

## 2.1 Port Automata

We first describe computing agents as automata, that can receive values at "input ports" and output values at "output ports." We use the term "port" instead of "channel" to emphasize that this is where an automaton interfaces with its environment. The set of events of an automaton comes equipped with a concurrency relation as mentioned above.

*Definition* 1. A **concurrent alphabet** is a set $E$, equipped with a symmetric, irreflexive binary relation $\|_E$, called the **concurrency relation**.

This concept is used in trace theory [1, 18] to obtain an algebraic theory of traces. We call events related by the concurrency relation *concurrent.* Let $V$ be a set of **data values** called the *value alphabet.* Throughout this paper, we will assume a fixed countable value alphabet. We refer to $V^\infty$ as the domain of streams. We use the term "stream" interchangeably with the term "value sequence."

We now describe the notion of an automaton that can execute events. The input and output events are described as (port,value) pairs. The rest of the events need not be of this form.

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

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

where

- $E$ is a concurrent alphabet of **events**, and $\mathit{Inp}$ and $\mathit{Out}$ are disjoint subsets of $E$, called the sets of **input events** and **output events**, respectively. $\mathit{Inp} = P^{in} \times V$, and $\mathit{Out} = P^{out} \times V$, 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 elements of $E \backslash (\mathit{Inp} \cup \mathit{Out})$ 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 $E \cup \{\epsilon\}$. $\epsilon$, a special event not in $E$, is called the **identity event**.

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 any state $q$ and any input event $a$, there exists a state $r$ such that $a \in A(q,r)$.

**(Commutativity)** For any state $q$ and any 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 [16, 21], and is closely related to the *input-output automata* of Lynch and Tuttle [17].

The intuitive significance of these conditions is as follows. Disambiguation states that from a particular state, an event cannot take the automaton to two different states. A basic property of systems is that they cannot control what their inputs are. They may, of course, ignore their inputs but they cannot determine their inputs, which are supplied by the external environment. The receptivity property makes this precise. If two events are concurrent, and if both of them are enabled in a particular state, then the execution of any one of these two events does not disable the other, and moreover, the execution of these events in either order results in 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 will denote the transition $(q, a, r)$ by $q \overset{a}{\to} r$. The transition $q \overset{\epsilon}{\to} q$ is called an *identity transition*, and is denoted by $id_q$.

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

$$q_1 \overset{a_1}{\to} q_2 \overset{a_2}{\to} \ldots$$

The *domain dom($\gamma$)* of $\gamma$ is the state $q_1$. A computation sequence is said to be *initial* if *dom($\gamma$)* is the distinguished start state $q^\iota$. Two computation sequences $\gamma$ and $\delta$ are *coinitial* if *dom($\gamma$) = dom($\delta$).*

In earlier work [21], automata in which inputs cannot disable other events were considered. These are called *monotone* automata, and these are port automata satisfying the following additional property.

*Definition* 4. *(Non-Disabling Inputs)* If $e$ is an input event at an input port, then $e \| e'$ for any event $e'$ that is not an input event at the same port.

As Example 2 below shows, not all port automata satisfy this property, and since we would like to be able to represent automata as in Example 2, we will not restrict our discussion to monotone automata in this paper. The fact that networks of monotone automata cannot implement non-monotone automata in general is the essential content of one of the expressiveness results stated operationally.

We will now give two examples of automata. We use $^\wedge$ as an infix operator for representing concatenation of sequences.

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

Let the set of states $Q$ be $V^*$. A state represents the contents of the input port. The initial state is $\Lambda$. Let the set of input events *Inp* be $\{i\} \times V$ and the set of output events *Out* be $\{o\} \times V$. Then the set of events $E$ is *Inp*$\cup$*Out*.

We now define the transition relation, using $v$ to represent a member of $V$. $A(q, r) = \{(i, v)\}$ iff $r = q^\wedge v$. $A(q, r) = \{(o, v)\}$ iff $q = v^\wedge r$. $A(q, q) = \{\epsilon\}$.

Every event in *Inp* is concurrent with every event in *Out*, and $\epsilon$ is concurrent with any other event.

*Example* 2. *Poll* : This automaton has one input port and one output port. It repeatedly checks its input port for data. If a data value is present, then it is read and output. If not, a special value $\star$ is output.

Let $Q$ 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 $(o, \star)$.

Besides the transitions in the previous example, there is an extra transition, and we redefine $A(q, q)$. $A(q, q) = \{\epsilon, (o, \star)\}$ if $q = \Lambda$, and $A(q, q) = \{\epsilon\}$ otherwise. The only new thing added to the concurrency relation of the previous example is that $\epsilon$ is concurrent with the new event $(o, \star)$.

Notice that input events are not concurrent with $(o, \star)$, so that arrival of input disables the output of a $\star$. So the input here has the power to interrupt already enabled output. This automaton is therefore not monotone. It will turn out that poll has the expressive power of fair merge.

We end this subsection with some notation. We refer to events of the form $(p, v)$ as $p$-events. We denote the value component $v$ of an event $e = (p, v)$ by *value*$(e)$. We also extend the definition of *value* to sequences – *value*$((p, v_1)(p, v_2) \ldots)$ is $v_1 v_2 \ldots$. For any computation sequence

$$\sigma = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots$$

we define $ev(\sigma)$, the *sequence of events of* $\sigma$ to be $a_1 a_2 \ldots$. We will use the symbol $\Pi$ as a projection operator on sequences of events. Given any sequence $t$ of events and a set $S$ of ports, we use $\Pi_S(t)$ to represent the subsequence of $t$ consisting of the $p$-events in $t$ for all ports $p$ in $S$. If $S$ is a singleton set $\{p\}$, then we use the notation $\Pi_p(t)$ instead of $\Pi_{\{p\}}(t)$. If $t$ is the sequence of events of a computation sequence $\sigma$, then we also write $\Pi_S(\sigma)$ to mean the same thing as $\Pi_S(t)$, and $\Pi_p(\sigma)$ to mean the same thing as $\Pi_p(t)$. To project out the $p$-events of a sequence $t$, we define $\Pi_{\sim p}(t)$ to be the subsequence of $t$ consisting of all the events of $t$ except the $p$-events. When we compare the projections of a sequence of events onto different ports, we follow the convention of implicitly applying *value* to the projections. We use the notation $t[i]$ to represent the $i$th event in a sequence $t$ of events, and the notation $\gamma[i]$ to represent the $i$th transition in a computation sequence $\gamma$.

## 2.2 Completed computations and the input-output relation

In this subsection, we describe which computation sequences of automata we view as "completed," i.e. cannot be extended further. Once we establish this, we then show how we can abstract the input-output behaviour of an automaton from its completed computation sequences.

We first describe the notion of a *history*. Let $P$ be the set of input ports and output ports of an automaton. A *history over $P$* is defined to be a function from $P$ to $V^\infty$. Then for any computation sequence $\sigma$, we can define a history $H_\sigma$ by letting $H_\sigma(p)$ be *value*$(\Pi_p(\sigma))$. Similarly, for any sequence $t \in (P \times V)^\infty$, we can define a history $H_t$ by letting $H_t(p)$ be *value*$(\Pi_p(t))$. We denote the restriction of $H_\sigma$ to the input ports by $H_\sigma^{in}$, and call it the *input port history* corresponding to $\sigma$. We denote the restriction of $H_\sigma$ to the output ports by $H_\sigma^{out}$, and call it the *output port history* corresponding to $\sigma$.

We now describe the computation sequences that we consider as "completed." To do this, we extend the prefix ordering on computation sequences to include the concurrency information in the concurrency relation. A finite computation sequence $\gamma$ is a *prefix* of a computation sequence $\delta$, and we write $\gamma \preceq \delta$, iff there exists a computation sequence $\xi$ with $\gamma\xi = \delta$. We define *permutation equivalence* to be the least congruence $\sim$, respecting concatenation, on the set of finite computation sequences of an automaton such that whenever $a \| b$, the computation sequences $q \xrightarrow{a} r \xrightarrow{b} p$ and $q \xrightarrow{b} s \xrightarrow{a} p$ are $\sim$-related. We define the *permutation preorder* relation $\precsim$ on finite computation sequences of $A$ as the transitive closure of $\preceq \cup \sim$. Define $\simeq = \precsim \cap \succsim$. We can now extend the permutation preorder relation to infinite computation sequences by defining $\gamma \precsim \delta$ iff for every finite $\gamma' \preceq \gamma$, there exists a finite $\delta' \preceq \delta$, such that $\gamma' \precsim \delta'$. We define $\simeq = \precsim \cap \succsim$ for infinite computation sequences also.

We would like a notion of "completed" computation sequence, in which all events that could happen at any state have either happened or been disabled. This is a *weak fairness* condition, and we formalize it as follows.

*Definition* 5. A computation sequence $\gamma$ is called **completed** if it is either finite and no non-input event is enabled at its end, or it is infinite and there is no event $e$ enabled at every state in $\gamma[i..]$ and concurrent with every event in $\gamma[i..]$ for some $i$.

This turns out to be identical to $\underset{\sim}{\sqsubset}$-maximality for a particular input [29].

For the port automata considered by Panangaden and Stark [21], no two non-input events were concurrent. In this paper, the notion of a *fair* initial computation sequence was introduced. For single automata, this notion is equivalent to our notion of completion. We will consider networks of automata presently, and for the networks of monotone automata considered by Panangaden and Stark [21], the fair computation sequences coincide with the completed computation sequences, which coincide with the $\underset{\sim}{\sqsubset}$-maximal computation sequences. Moreover, the projections of maximal computation sequences onto individual automata in the network are themselves maximal. When we consider networks of general port automata as in this paper, this is no longer the case. So, as we will discuss in the next subsection, we will give a different notion of completion for networks of automata. For networks of monotone automata however [21, 33], the projections of a maximal computation sequence onto individual automata are again maximal. This means that even though our above notion of completion is equivalent to maximality for general automata, this notion can be extended to networks of monotone automata but not to networks of general automata. This is unfortunate, but unavoidable.

We could think of the preorder $\underset{\sim}{\sqsubset}$ as the prefix ordering in which concurrency information has been encoded. It is quite pleasant to be able to state completedness as a maximality property of computation sequences.

*Example* 3. Consider an automaton with one output port $o$ and only two internal events $e_1$ and $e_2$. There are only three states $q_1, q_2, q_3$. $q_1$ is the initial state. The transitions are completely described by $A(q_1, q_1) = \{\epsilon, e_1\}$, $A(q_1, q_2) = \{e_2\}$, $A(q_2, q_3) = \{(o, v)\}$ where $v$ is some fixed value, $A(q_2, q_2) = \{\epsilon\}$ and $A(q_3, q_3) = \{\epsilon\}$. There is an infinite sequence $\gamma$ of $e_1$-transitions from the initial state, and the event $e_2$ is continuously enabled, following which an output would be enabled. But the event $e_2$ is not concurrent with $e_1$, and the automaton is really making an indeterminate choice at each step without any fairness constraints. So we would accept $\gamma$ as a completed computation sequence.

This example makes clear the distinction between completion and continuous enabling.

Now we can describe the *input-output relation* of an automaton. This describes the input-output behaviour and corresponds to the *observable* aspect of network behavior.

*Definition* 6. The **input-output relation** of an automaton is the set of all pairs $(H_\sigma^{in}, H_\sigma^{out})$ with $\sigma$ being a completed computation sequence of the automaton.

We can also equivalently consider the input-output relation to be a set of pairs of tuples of streams, the first tuple of each pair consisting of streams at the input ports and the second tuple consisting of streams at the output ports. We refer to the input-output relation as the *IO-relation.*

If the IO-relation turns out to be the graph of a function then the automaton is said to *compute* that function. We now describe a subclass of the class of automata that compute functions. The following definition is due to Stark [30].

*Definition* 7. An automaton is **determinate** if it satisfies the following condition : $b \| b'$ whenever $b, b'$ are distinct non-input events both enabled at some state.

Intuitively, a determinate automaton does not exhibit "internal indeterminacy". These are the automata that correspond to the computing agents that Kahn considered in his treatment of networks, in view of the following theorem of Stark [30].

*Theorem* 1. A function $f$ is computed by a determinate automaton iff $f$ is a continuous function.

## 2.3 Networks of Automata

We now describe how we can build networks of automata. We consider three operations – aggregation, feedback and output hiding. Ordinary sequential composition of programs can be described in terms of these operations. Thus these operations cover the normal intuitions of forming networks from individual components.

Aggregation involves taking two networks with disjoint sets of ports and "keeping them side by side" to obtain a new network. Feedback (see Figure 1) involves identifying an output port $p_1$ of the network with an input port $p_2$ of that network, so that the port $p_2$ can no longer be observed by the environment and can no longer serve as an input port. The values appearing at $p_2$ will be those that are output at $p_1$. One stipulation that we make is

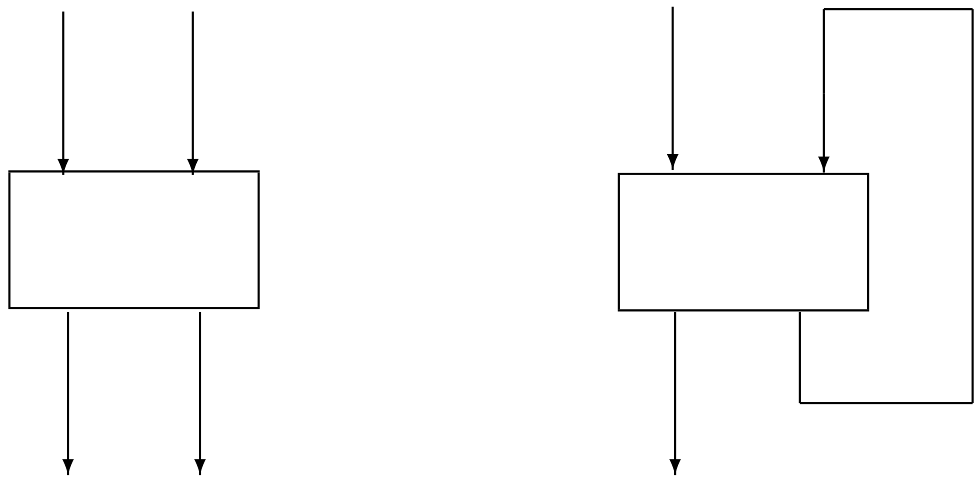

Figure 1: Feedback

that the output port $p_1$ and the input port $p_2$ not be ports of the same automaton in the network. Our last operation, output hiding, removes an output port, so that the environment can no longer observe values at this hidden port.

We can define the first two operations as special cases of the *composition* of a finite *compatible* set of automata.

*Definition* 8. If $I$ is a finite index set, then a set $\mathcal{S} = \{M_i : i \in I\}$ of automata is said to be **compatible** if

- for all $i, j \in I$ such that $i \neq j$ we have $(E_i \backslash (\mathrm{Inp}_i \cup \mathrm{Out}_i)) \cap (E_j \backslash (\mathrm{Inp}_j \cup \mathrm{Out}_j)) = \emptyset$, that is, the sets of internal events of any pair of automata are disjoint, and,
- for any port name, at most two automata may have that name in common, and in that case, it must be the name of an output port of one automaton and an input port of the other automaton,

where $M_i = (E_i, Q_i, A_i)$, and $\mathrm{Inp}_i$ is the set of input events of $M_i$, and $\mathrm{Out}_i$ is the set of output events of $M_i$.

The shared port names represent ports that will get connected when the set of automata are composed together. We will then obtain a *network* automaton. The input ports of the network will be all the input ports of the $M_i$'s, excluding those that are shared. The output ports of the network will be all the output ports of the $M_i$'s.

*Definition* 9. The **composition** of a compatible set $\mathcal{S}$ of 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 $E_i$.
- $\mathit{Out} = (\cup \mathit{Out}_i)$, and $\mathit{Inp} = (\cup \mathit{Inp}_i) \backslash (\cup \mathit{Out}_i)$,
- $Q = \prod_{i \in I} Q_i$
- $q^\iota = (q_i^\iota : i \in I)$
- $e \in A((q_i : i \in I), (r_i : i \in I))$ 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 definition of $\|$ above 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. We now note that the above definition includes the definition of *aggregation*, which is what happens when two automata do not share any ports, and the definition of *feedback*, which is what happens when two automata share a port name and it is the output port of one automaton and an input port of the other automaton. We now explicitly define *output hiding.*

*Definition* 10. If $\mathcal{A}$ is an automaton with input ports $P^{in}$ and output ports $P^{out}$, and $S$ is a subset of $P^{out}$, then the **output hiding** of $S$ in $\mathcal{A}$ results in the automaton with input ports $P^{in}$ and output ports $P^{out} \backslash S$ with exactly the same sets of events and states and the same transition relation as $\mathcal{A}$.

When we compose two automata with a shared port name $p$, $p$ must be an output port of one automaton and an input port of the other automaton. The two automata connected in this manner may execute a single event in the composed automaton. It is important to remember that this corresponds to two different events in the original collection of automata. By defining composition in this way, we do not have to worry about liveness conditions, to ensure that values output onto $p$ will eventually arrive at the input port of the other automaton.

We can define *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 recall that, for a single automaton, the completed computation sequences were the $\precsim$-maximal ones. We could

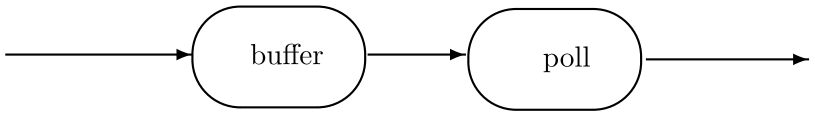

Figure 2: A Buffer and a Poll

now define *permutation equivalence* and the *permutation preorder* $\precsim$ as we did for single automata, and then talk about the $\precsim$-maximal computation sequences. But we argue that this maximality condition is not a reasonable definition of completion for networks in general. We illustrate this point of view by an example.

*Example* 4. Consider the network in Figure 2, consisting of a buffer automaton, as in example 1, and a poll automaton, as in example 2. Suppose we took maximality of computation sequences as defining completion. Then the computation sequence starting with an input event at the input port of the buffer, followed by an infinite sequence of $\star$ outputs by the poll, is a maximal computation sequence, because even though the output event for buffer (input event for poll) is enabled at every state following the read event of the buffer, it does not commute with a $\star$ output. But this computation sequence does not really make sense as a completed sequence, because we want the buffer to output all its input values. In other words, the projection of the network computation sequence onto the ports of the buffer is not a completed computation sequence for the buffer.

The following definition, however, captures the proper notion of completion.

*Definition* 11. A computation sequence $\gamma$ for a network of automata is **completed** if, for every automaton $M$ in the network, the restriction of $\gamma$ to $M$ is a completed computation sequence for $M$.

Intuitively, this definition makes sense, because we would like to call those computation sequences of the network "completed," in which every component automaton gets chances to execute and exhibits a completed computation sequence in every computation of the network.

Just as we defined the input-output relation for a single automaton earlier, we can now define the following :

*Definition* 12. The **input-output relation** of a network of automata is the set of all pairs $(H_\sigma^{in}, H_\sigma^{out})$ with $\sigma$ being a completed computation sequence of the network, $H_\sigma^{in}$ being the input port history corresponding to $\sigma$, and $H_\sigma^{out}$ being the output port history corresponding to $\sigma$.

## 2.4 Observability, Buffering and Processes

It is clear that, for an "external" observer of an automaton, all that can be observed is a sequence of input events at each input port of the automaton, and a sequence of output events at each output port of the automaton. Due to the asynchronous nature of communication between automata, there may be an arbitrary delay between the emission of a value at an output port and the observation of that value. So it is not possible for an observer to determine the order of emission of values on *different* ports,

To reason about processes, we would like to abstract away from computation sequences by considering only the input events and output events, as these are the events that interface the process with the external world. But the definition of automata do not allow any two events on different ports to be commuted. That means that if there is a computation sequence $\gamma$ with event $e$ on output port $p$ occurring before an event $e'$ on some other output port $p'$, then there may be no computation sequence with $e'$ preceding $e$ and the rest of $\gamma$ unchanged. But, as far as the external observer is concerned, he should not be able to distinguish the automaton in which $e$ always occurs before $e'$ from the automaton in which $e$ may precede $e'$, or $e'$ may precede $e$. So, as far as the external observer is concerned, these two automata should have the same set of "traces" – abstractions from computation sequences.

For this reason, we will describe **processes** as automata with buffers attached to each of the input and output ports. As we shall see, this will allow us to commute events on different ports. A buffer was described as a specific automaton in example 1. Having buffers in the description of a process will also make the arbitrary delay between the emission of a value and its observation explicit in computation sequences of the network.

We now formally describe what we mean by a "buffered automaton."

*Definition* 13. A **process** is the composition of

(i) an automaton, called the **central automaton** of the process, with input ports $i_1, i_2, \ldots i_n$ and output ports $o_1, o_2, \ldots o_m$, and

(ii) $m + n$ buffers, called the **process buffers**, $n$ of them having output ports $i_1, i_2, \ldots i_n$ respectively, and the remaining $m$ of them having input ports $o_1, o_2, \ldots o_m$ respectively. The rest of the ports of the buffers are disjoint from the set $\{i_1, i_2, \ldots i_n, o_1, o_2, \ldots o_m\}$.

with the ports $\{i_1, i_2, \ldots i_n, o_1, o_2, \ldots o_m\}$ hidden.

Figure 3 shows a process; we may also refer to this as a buffered automaton.

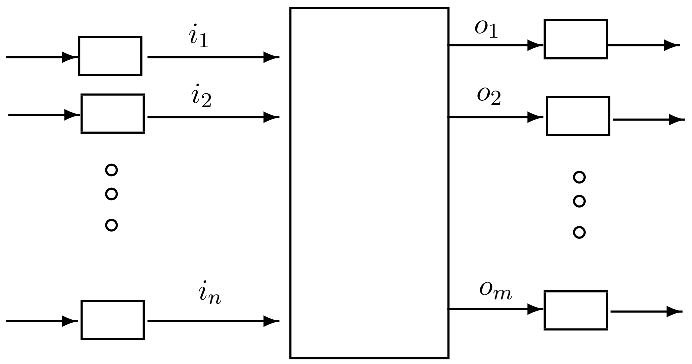

Figure 3: The Structure of a Process

By our definition of completed computation sequences, we ensure that every value input to the process will arrive at an input port of the central automaton, and every value output by the central automaton will be output by the process.

Just as we defined a compatible set of automata in the previous section, we can similarly define a compatible set of processes – the set consisting of the automata that make up these processes must be compatible. The *composition* of a compatible set of processes is the network obtained by composing the set of network automata representing the individual processes. A network of processes may also be obtained by hiding some of the output ports of the composition of a compatible set of processes.

We can now talk about notions of implementability and non-implementability.

*Definition* 14. A set $S$ of processes can **implement** a relation $R$ if there is a finite network $N$, built out of copies of processes in $S$, such that $R$ is the input-output relation of $N$.

*Definition* 15. A set $S$ of processes can **implement** a process or a network $M$ if there is a finite network $N$, built out of copies of processes in $S$, such that $N$ and $M$ have the same input-output relation.

In this paper, we differentiate between classes of processes, as exemplified by the merge processes, by proving non-implementability results between these classes.

## 3 Trace sets

We can describe the behaviour of a network by describing its input-output behaviour. The input-output relation of a network of processes is the input-output relation of the associated network of automata. Since "buffering" an automaton does not affect the input-output relation, we have a process with the input-output relation of *poll* (example 2). But note that we do not consider "arbiters," which can distinguish between the order of arrival of values at distinct input ports. This is because, for the processes we consider, we cannot distinguish between the order of arrival of values at distinct input ports, because the central automaton of the process may see the values in either order, as the buffers at the input ports of the process may produce the corresponding output events in either order.

The input-output relation semantics for processes and networks fails to be compositional. The reason for this is that the input-output relation describes which sequences of values can be obtained at the output ports for given sequences of values at the input ports. So this view describes the *entire* output when given the *entire* input. It does not describe which particular values in the output sequences *really depended* on the presence of which particular values in the input sequences. Briefly, there is no causality information between events encoded in the input-output relation. One way to express causality information is to consider sequences of input and output events.

*Definition* 16. If $\gamma$ is a computation sequence, then we define $tr(\gamma)$ to be the subsequence of $ev(\gamma)$, consisting of all the input and output events in $ev(\gamma)$.

*Definition* 17. A **trace** of a network $N$ of processes is a sequence $t$ of input events and output events of $N$, such that $t = tr(\gamma)$ for some completed computation sequence $\gamma$ of $N$. We write $Trset(N)$ for the set of traces of a network $N$.

In an earlier discussion of network semantics, trace sets were called archives [14]. Presented in this way, the traces appear as an abstraction of computation sequences that were defined using an operational formalism. The important point is that we can define composition rules directly on trace sets, and this allows us to build up trace sets of complex networks structurally. If $t$ is a sequence of events and $P$ is a process or a network, $\Pi_P(t)$ will represent the subsequence of $t$ consisting of all the events on the input and output ports of $P$ in $t$.

We emphasize here that due to the description of processes as automata with buffers, the trace set of a process satisfies natural closure conditions – if two events $e_1$ and $e_2$ are adjacent in a trace $t$, and either (i) both are input events at different ports, or (ii) both are output events at different ports, or (iii) $e_1$ is an output event and $e_2$ is an input event, then flipping these two events gives us a trace of the process. This property of trace sets of processes is natural, as discussed in the subsection on observability earlier.

We would like to prove a theorem that the trace set of a network may be described in terms of the trace sets of its subnetworks. In order to do this we need to formalize some preliminary notions. We define the notion of corresponding events in a trace or a computation sequence.

*Definition* 18. In a trace or computation sequence of a network with two of its ports being $p_1$ and $p_2$, a $p_1$-event occurrence $e_1$ and a $p_2$-event occurrence $e_2$ are said to be *corresponding events* if $e_1$ is the $i$th event on port $p_1$, and $e_2$ is the $i$th event on port $p_2$ for some $i > 0$.

We use the predicate "precedes" to indicate precedence between events in a trace.

*Definition* 19. $\mathit{precedes}(p_1, p_2, t) =_{def}$ for every finite prefix of $t$, $\mathit{value}(\Pi_{p_2}(t))$ is a prefix of $\mathit{value}(\Pi_{p_1}(t))$.

Informally, this means that every $p_2$-event occurrence is preceded by a corresponding $p_1$-event occurrence with the same value. This predicate is useful for defining the trace sets of networks built using feedback.

The following theorem states that traces are compositional.

*Theorem* 2. Consider any decomposition of a network $N$ into subnetworks $\{N_i\}$ for $i$ in some finite index set $I$, where the $N_i$ are not necessarily individual processes. Then the set of traces $Trset(N)$ of the network is exactly the set of sequences $\Pi_N(t)$, such that

(i) $t$ is a sequence of events at the input and output ports of the $N_i$'s, distinguishing between an output port and an input port even if they have the same name $p$ by calling them $p_{out}$ and $p_{in}$ respectively,

(ii) for all $i \in I$, $\Pi_{N_i}(t)$ is a trace of $N_i$, and

(iii) for every port $p$ that is an output port of $N_i$, as well as an input port of $N_j$, $i$ not necessarily distinct from $j$, $\mathit{precedes}(p_{out}, p_{in}, t)$ and $\Pi_{p_{out}}(t) = \Pi_{p_{in}}(t)$.

**Proof :** Let $t$ be a sequence of events, satisfying conditions (i), (ii) and (iii) above. We show that $\Pi_N(t)$ is a trace of $N$. Let $t_i = \Pi_{N_i}(t)$. For each $t_i$, there is a completed computation sequence $\gamma_i$ of $N_i$, such that $t_i = tr(\gamma_i)$.

Then we can dovetail among the computation sequences $\{\gamma_i : i \in I\}$ to obtain a computation sequence $\gamma$, such that $t$ is the subsequence of $ev(\gamma)$ consisting of all the input and output events of the $N_i$. Now, using condition (iii) in the statement of the theorem, and by the property of non-disabling inputs of processes, we obtain a new computation sequence $\gamma''$ in which for every shared port $p$, the output events at $p_{out}$ immediately precede the corresponding input events at $p_{in}$. Replacing each such adjacent pair of events at $p_{out}$ and $p_{in}$ by a single $p$-event, we obtain a computation sequence $\gamma'$ of the network $N$ in which the output events on the shared port are also the input events on the shared port. Any automaton $M$ in the network is in one of the subnetworks, say $N_i$, and so the restriction of $\gamma'$ to $M$ is the same as the restriction of $\gamma_i$ to $M$. Since the latter is completed, because $\gamma_i$ is a completed computation sequence of $N_i$, so is the restriction of $\gamma'$ to $M$. Therefore $\gamma'$ is a completed computation sequence of the whole network, and so $tr(\gamma')$ is a trace and it is exactly the restriction of $t$ to the events on the input and output ports of the network.

Let $t'$ be a trace of $N$ corresponding to the completed computation sequence $\gamma$ of the network, composed of subnetworks $N_i$. The restriction $\gamma_i$ of $\gamma$ onto any subnetwork $N_i$ is then a completed computation sequence of $N_i$ by the definition of a completed computation sequence of a network. Let $t''$ be the subsequence of $ev(\gamma)$ consisting of all the input and output events of the $N_i$ in $ev(\gamma)$. Now, for every port that is shared by an $N_i$ and an $N_j$, $i$ not necessarily distinct from $j$, replace an event on that port in $t''$ by two events – an output event on the output port followed by an input event on the input port. The new event sequence $t$ is a sequence of events such that its restriction to the events of any subnetwork $N_i$ is a trace of that subnetwork, and moreover, the restriction of $t$ to events on the input and output ports of the network is the same as the trace $t' = tr(\gamma)$. ∎

It is possible to have processes 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 [28].

We will use vector notation to talk about collections of ports. We project traces onto vectors of ports and assume that we get a vector of event sequences. Thus, we may write, "a trace $t$ when projected onto ports $\vec{a}$ produces the sequences $\vec{\alpha}$." This notation allows us to write many projections at once and also allows us to name the components of the vectors as the need arises.

We described the construction of networks in the previous section. We use the notation $N \| M$ to represent the network built by aggregation from the two subnetworks $N$ and $M$. We use the notation $loop(p_1, p_2, N)$ to represent the network built by identifying an output port $p_1$ of the network $N$ with the input port $p_2$ of $N$. Lastly, the traces of a network built by hiding some output ports of another network are obtained by simply taking the traces of the original network and projecting out the events on the hidden output ports. We use the notation $N \backslash S$ for the network $N$ with the ports $S$ of $N$ hidden.

*Theorem* 3.

$$\text{Trset}(N \| M) = \{t \in (P \times V)^\infty | \Pi_N(t) \in \text{Trset}(N) \wedge \Pi_M(t) \in \text{Trset}(M)\}$$

where $P$ is the set of input and output ports of $N$ and $M$.

*Theorem* 4. $Trset(loop(p_1, p_2, N))$ consists of every sequence $\Pi_{\sim p_2}(t)$ such that $t \in Trset(N)$ and $precedes(p_1, p_2, t)$ and $(\Pi_{p_1}(t) = \Pi_{p_2}(t))$.

The statement of the theorem expresses the idea that in the network with the feedback, the events on $p_2$ arise from events on $p_1$.

Let us define a predicate similar to *precedes*, that captures "immediate precedence" of events.

*Definition* 20. $immprecedes(p_1, p_2, t) =_{def} precedes(p_1, p_2, t) \wedge$ (every $p_2$-event is immediately preceded by the corresponding $p_1$-event in $t$)

Then it follows by monotonicity (definition 4) that for any trace $t'$ of $loop(p_1, p_2, N)$, there is a trace $t \in Trset(N)$ such that $immprecedes(p_1, p_2, t) \wedge (\Pi_{p_1}(t) = \Pi_{p_2}(t))$.

*Theorem* 5. Suppose $S_O$ is a subset of the set of output ports of a network $N$. Then

$$Trset(N \backslash S_O) = \{\Pi_{\sim S_O}(t) | t \in Trset(N)\}$$

All these proofs follow directly from Theorem 2 that establishes compositionality of trace sets.

The fact that we have a compositional description of network behavior allows one to prove properties of networks by structural induction on the network.

## 4 The Merge Primitives

In this section, we describe the merge primitives fair merge, angelic merge and infinity-fair merge. We describe both their input-output behavior and their trace sets. The point of introducing these primitives is to study and compare the interesting classes of processes in which they lie. We emphasize

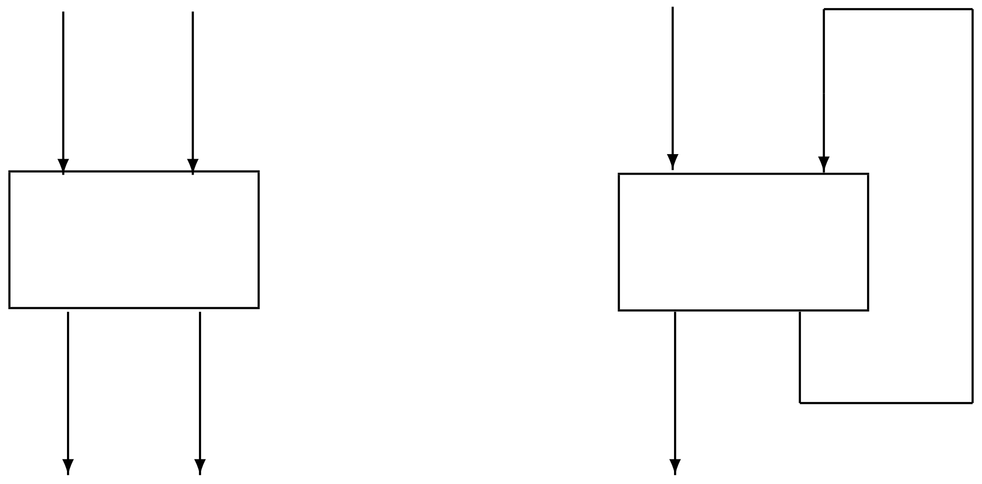

Figure 4: Loop formation

that we wish to make interesting distinctions between the input-output relations of these primitives. We use the compositionality of trace sets to make these distinctions. Informal descriptions of the merge primitives were given in the introduction. Now we formally describe their input-output relations, and specific trace sets.

We first describe angelic merge. Intuitively, angelic merge can avoid getting stuck at input ports with no input, but need not be fair on infinite input streams. The input-output relation of angelic merge consists of triples $(s_i, s_{i'}, s_o)$ of value sequences, such that $s_o$ can be broken up into two subsequences $s_1, s_2$ such that

(i) $s_1$ is a prefix of $s_i$ and $s_2$ is a prefix of $s_{i'}$,

(ii) if $s_1$ is finite, then $s_2 = s_{i'}$, and if $s_2$ is finite, then $s_1 = s_i$.

We now give a specific trace set that has the above IO-relation associated with it. If the angelic merge process has input ports $i, i'$ and output port $o$, then the trace set consists of all event sequences $t \in (\{i, i', o\} \times V)^\infty$ such that $\Pi_o(t)$ can be broken up into two subsequences $s_1, s_2$ such that

(i) $\mathit{value}(s_1)$ is a prefix of $\mathit{value}(\Pi_i(t))$ and $\mathit{value}(s_2)$ is a prefix of $\mathit{value}(\Pi_{i'}(t))$,

(ii) if $\Pi_i(t)$ is finite, then $\mathit{value}(s_2) = \mathit{value}(\Pi_{i'}(t))$,

(iii) if $\Pi_{i'}(t)$ is finite, then $\mathit{value}(s_1) = \mathit{value}(\Pi_i(t))$, and

(iv) for every prefix $t'$ of $t$, the prefix of $s_1$ in $t'$ is a prefix of $\Pi_i(t')$, and the prefix of $s_2$ in $t'$ is a prefix of $\Pi_{i'}(t')$.

Before describing infinity-fair merge, we first describe a primitive *anyints* that embodies unbounded indeterminacy. It has one input port and one output port. It consumes any value on the input port and produces any positive integer on the output port. In particular, we could feed it an infinite stream of zeros and have it produce an infinite stream of arbitrary positive integers on its output port. With *anyints* we can easily program a merge process that is fair if both its input streams are infinite. We use the *anyints* to produce an arbitrary stream of integers. We can then use this as an "oracle" for a determinate process that has three input ports and a single output port. The determinate process uses the arbitrary stream of integers to determine how many tokens to read off each of its input ports before switching to the other input port. The adjective *infinity-fair* was coined by Park [22] for such a process.

The intuition behind infinity-fair merge is captured by the above simulation. If both input sequences are infinite, then it reads both of them completely and outputs an interleaving of them. The process will however get stuck at an input port with no more values left to read, if either of the input sequences is finite. The input-output relation of infinity-fair merge consists of triples $(s_i, s_{i'}, s_o)$ of value sequences, such that $s_o$ can be broken up into two subsequences $s_1, s_2$ such that

(i) $s_1$ is a prefix of $s_i$ and $s_2$ is a prefix of $s_{i'}$,

(ii) if $s_1$ is infinite, then $s_2 = s_{i'}$, and if $s_2$ is infinite, then $s_1 = s_i$.

We now give a specific trace set that has the above IO-relation associated with it. If the infinity-fair merge process has input ports $i, i'$ and output port $o$, then the trace set of the process consists of all sequences $t \in (\{i, i', o\} \times V)^\infty$ such that $\Pi_o(t)$ can be broken up into two subsequences $s_1, s_2$ such that

(i) $\mathit{value}(s_1)$ is a prefix of $\mathit{value}(\Pi_i(t))$ and $\mathit{value}(s_2)$ is a prefix of $\mathit{value}(\Pi_{i'}(t))$,

(ii) if $\Pi_i(t)$ is infinite, then $\mathit{value}(s_2) = \mathit{value}(\Pi_{i'}(t))$,

(iii) if $\Pi_{i'}(t)$ is infinite, then $\mathit{value}(s_1) = \mathit{value}(\Pi_i(t))$,

(iv) if either of $\Pi_i(t)$ or $\Pi_{i'}(t)$ is finite, then $\Pi_o(t)$ is finite, and either $\mathit{value}(s_1) = \mathit{value}(\Pi_i(t))$ or $\mathit{value}(s_2) = \mathit{value}(\Pi_{i'}(t))$, and

(iv) for every prefix $t'$ of $t$, the prefix of $s_1$ in $t'$ is a prefix of $\Pi_i(t')$, and the prefix of $s_2$ in $t'$ is a prefix of $\Pi_{i'}(t')$.

It is also true that infinity-fair merge can implement anyints. To exhibit this, we use an infinity-fair merge with fixed input sequences $0^\infty$ and $1^\infty$. Since the output must contain infinitely many 0's and infinitely many 1's, we can read off the lengths of the blocks of 1's and the blocks of 0's, and make the output of **anyints** be the sequence consisting of these lengths. Therefore **anyints** and infinity-fair merge can implement each other, and we can use either one interchangeably in implementability and non-implementability proofs.

Intuitively, fair merge requires sensitivity to the presence or absence of values at input ports. Keller [12] considered the addition of a command with syntax **poll** $a$, where $a$ is an input port, to the usual language of determinate node programs consisting of read, write and internal commands. The command has the effect of indicating whether there is a value available to be read on port $a$. Thus it is sensitive to the presence or absence of values at input ports. With this command, it is easy to program a fair merge. We can repeatedly poll the two input ports and read a value each time if it is present. This process never waits to read data that is not going to appear nor will it consistently favour a particular port. We introduce a primitive called poll that has similar behaviour. A poll primitive has one input port and one output port. It outputs a shuffle of the input sequence and an infinite sequence of $\star$s, where $\star$ is a special value that cannot be produced by other processes. With poll, one can easily implement fair merge. All one needs is a determinate zipper. This is a process with two input ports and one output port that reads values from each input port in strict alternation, and copies the values onto its output port. To implement a fair merge one puts a poll in front of each input port to the zipper. Then the resulting output can be filtered to remove the $\star$'s. It is trivial to implement poll with a fair merge – just merge the input stream with an infinite sequence of $\star$'s. Therefore poll and fair merge can implement each other.

Formally, the input-output relation of fair merge consists of triples $(s_i, s_{i'}, s_o)$ of value sequences, such that $s_o$ is an interleaving of $s_i$ and $s_{i'}$. We can give a specific trace set with this IO-relation associated with it. If the fair merge process has input ports $i, i'$ and output port $o$, then the trace set consists of all event sequences $t \in (\{i, i', o\} \times V)^\infty$ such that $\Pi_o(t)$ can be broken up into two subsequences $s_1, s_2$ such that

(i) $\mathit{value}(s_1) = \mathit{value}(\Pi_i(t))$, and

(ii) $value(s_2) = value(\Pi_{i'}(t))$, and

In the next section, we will use properties of trace sets of these merge processes to distinguish between them.

## 5 Inexpressiveness results

In this section, we study properties of the merge processes defined in the previous section, and prove that the different merge processes actually form different levels in a hierarchy of expressiveness. The properties that we define and study in this section are two different notions of "monotonicity." We will refer to these as "Hoare-monotonicity" and "Smyth-monotonicity," and these are properties of trace sets. We will prove that Smyth-monotonicity of trace sets is preserved under network composition, but it turns out to be exceedingly difficult to prove that Hoare-monotonicity of trace sets is preserved under network composition. So a stronger property of trace sets, that implies Hoare-monotonicity, needs to be proved. This is a "continuity" kind of property.

### 5.1 Hoare-monotonicity

The intuition behind the definition of Hoare-monotonicity is that, in a monotone network, i.e. satisfying the non-disabling input property (definition 4), arrival of values at input ports cannot disable output of values that were already enabled before the arrival of the values at the input ports. We formulate this property as a property of trace sets and also describe a stronger property - limit-closure - that is preserved by network composition.

*Definition* 21. Let $t_1$ and $t_2$ be two traces of a network $N$. Then the **relative order of events** in $t_1$ is said to be preserved in $t_2$ if the following two conditions hold :

(i) for every port $p$ of $N$, $\Pi_p(t_1) \sqsubseteq \Pi_p(t_2)$, and

(ii) for every pair $e_1, e_2$ of event occurrences in $t_1$, if $e_1$ is the $i$th event on port $p_1$, and $e_2$ is the $j$th event on port $p_2$, then $e_1$ precedes $e_2$ in $t_1$ iff the $i$th event on port $p_1$ in $t_2$ precedes the $j$th event on port $p_2$ in $t_2$.

*Definition* 22. Let $N$ be a network with input ports $\vec{a}$ and output ports $\vec{b}$. We say that $Trset(N)$ is *Hoare-monotone* if, for any trace $t \in Trset(N)$ and any finite prefix $t'$ of $t$ such that $\Pi_{\vec{a}}(t) = \vec{\alpha}$, and for any $\vec{\alpha}' \sqsupseteq \vec{\alpha}$, there is a trace $t'' \in Trset(N)$ with $\Pi_{\vec{a}}(t'') = \vec{\alpha}'$, with $\Pi_{\vec{b}}(t) \sqsubseteq \Pi_{\vec{b}}(t'')$, with $t'$ a prefix of $t''$ and with the relative order of the events in $t$ preserved in $t''$.

This says that, given a trace, we can always extend the input, and there will be some trace that represents the response of the network to the new input and in this new trace, the sequence of values on each port will be an extension of or equal to the sequence of values seen before. Clearly we cannot expect that every response will be an extension (in the above sense) of or equal to the old response since the networks are indeterminate. This definition captures the idea that if there was an enabled output then adding new input will not disable this output.

The definition of the Hoare-monotonicity of a trace set is quite involved since it deals with the orderings of events in traces, and not just with the input and output sequences. We now define a predicate on the input-output relations of networks, such that Hoare-monotonicity of the trace set of a network implies the truth of this predicate on the input-output relation of that network. We will give the same name Hoare-monotonicity to this predicate.

*Definition* 23. The input-output relation of a network $N$ is said to be **Hoare-monotone**, if whenever for some input $\vec{\alpha}$, there is an output $\vec{\beta}$, and $\vec{\alpha'} \sqsupseteq \vec{\alpha}$, then there is a possible output $\vec{\beta'} \sqsupseteq \vec{\beta}$.

Note that since we are only talking of the input-output relation here, we do not have requirements as for Hoare-monotonicity of trace sets, such as the preservation of relative order of events in traces.

The reason that we could not work with Hoare-monotonicity of the input-output relation is that this does not imply Hoare-monotonicity of the trace set, as the example below shows.

*Example* 5. We describe a process with input ports $a', b'$ and output port $c'$, and with the central automaton of the process having input ports $a, b$ and output port $c$. The central automaton repeatedly executes the following conditional statement : if $a$ has no values to be read, then write a 1 on $c$ and then read from $b$, otherwise first read a value from $a$, then read a value from $b$ and then write a 1 on $c$. The input-output relation of this process has the property that when there are no values at $a'$, then the output at $c'$ is of the same length as the input at $b'$. Moreover, when there is input at $a'$, then the length of the output at $c'$ is either the length of the input at $b'$ or 1 more than the length of the input at $b'$. It can be easily checked that this is a Hoare-monotone input-output relation. But the trace set of this process is not Hoare-monotone, as the sequence $(c', 1), (b', 1), (c', 1), (b', 1)...$ is a trace, but when the input at $a'$ is extended to a non-empty sequence, then the relative order of events in the above trace cannot be preserved, and for some $i > 0$, the $i$th $c'$-event follows the $i$th $b'$-event.

However, we justify the use of our definition of Hoare-monotonicity of trace sets by arguing that all processes whose central automata are monotone have a Hoare-monotone trace set. The point here is that for such processes, input events commute with all other events. So if $t$ is a trace and $\gamma$ is the corresponding completed computation sequence, then adding new input events after any prefix $\gamma_p$ of $\gamma$ gives us a new computation sequence extending $\gamma_p$ and with the relative order of events in $\gamma$ preserved. This can now be $\sqsubseteq$-extended to a completed computation sequence [21]. It is of course possible that one could have a process with a non-monotone central automaton that has a Hoare-monotone trace set. Thus our proof is more general than that of Panangaden and Stark [21].

To prove the preservation of Hoare-monotonicity of trace sets for network composition, we need to consider aggregation and feedback. Of these, aggregation is more or less trivial whereas feedback, not surprisingly, involves an inductive proof. In order for the inductive proof to go through, we need to introduce limit-closure.

*Definition* 24. $Trset(N)$ is said to be **limit-closed** if it satisfies the following property. Suppose that $\{t_i | i < \omega\}$ is a set of traces of $N$ such that for any port $p$, $\Pi_p(t_i)$ is a chain in the prefix ordering, and the relative order of events is preserved along this chain. Then there is a trace $t \in Trset(N)$ such that, for any port $p$ we have $\Pi_p(t) = \sqcup \Pi_p(t_i)$ and the relative order of events in $t_i$ is preserved in $t$ for every $i$.

We now show that a network composed of components with Hoare-monotone and limit-closed trace sets has to have a Hoare-monotone and limit-closed trace set. This is the fundamental theorem on which our first expressiveness result rests.

*Theorem* 6. The aggregate of two networks, $N$ and $M$, with Hoare-monotone and limit-closed trace sets has a Hoare-monotone and limit-closed trace set.

**Proof:** Let $M$ and $N$ be networks with Hoare-monotone and limit-closed trace sets. Let $t$ be a trace of $N \| M$. Then $t$ is a shuffle of $t_1$ and $t_2$, where $t_1$ is a trace of $N$ and $t_2$ is a trace of $M$. Extending the input streams of $N \| M$ extends the input streams of $N$ and $M$. By Hoare-monotonicity of $N$ and $M$, there are traces $t'_1$ and $t'_2$ of $N$ and $M$ respectively, with the extended input and output and with relative order of events in $t_1$ and $t_2$ preserved in $t'_1$ and $t'_2$ respectively. Then $t'$, an appropriate shuffle of $t'_1$ and $t'_2$, is the desired trace of $N \| M$.

The proof that limit-closure is preserved is very similar. One can decompose traces of $N \| M$ to obtain traces of $N$ and $M$. One can obtain the required limiting vector by using the limit-closure of the trace sets of the components and appropriately shuffling the corresponding traces. ∎

Now we consider the case of feedback. For definiteness, let us suppose that the network in question is $M$ and that the output port $b$ of $M$ is connected to the input port $c$ forming a feedback loop. Let the network $M$ without this feedback loop be called $N$. Note that $N$ has an extra input port $c$. Recall that the relationship between the trace sets of the two networks is

$$Trset(M) = \{\Pi_{\sim c}(t) | t \in Trset(N) \wedge \mathit{precedes}(b, c, t) \wedge (\Pi_c(t) = \Pi_b(t))\}.$$

We will use the following convenient notation: suppose that $\vec{\alpha}$ is a vector of $n$ sequences, then $\vec{\alpha}; \gamma$ will mean the vector consisting of the sequences in $\vec{\alpha}$ and the sequence $\gamma$ in the $n+1$th position.

*Theorem* 7. Suppose the networks $M$ and $N$ are as above. If $Trset(N)$ has a Hoare-monotone and limit-closed trace set, then so does $Trset(M)$.

**Proof :** Suppose $t \in Trset(M)$, $\Pi_I(t) = \vec{\alpha}$ and $\Pi_O(t) = \vec{\beta}$, where $I$ and $O$ are the sets of input ports and output ports, respectively, of $M$. Now we consider extending the input to $\vec{\alpha}'$, and we pick a finite prefix $t_p$ of $t$. By theorem 3, there is some trace $t_0 \in Trset(N)$ such that $\Pi_I(t_0) = \vec{\alpha}; \Pi_c(t_0)$ and $\Pi_O(t_0) = \vec{\beta}$ and $\mathit{precedes}(b, c, t_0)$ and $\Pi_c(t_0) = \Pi_b(t_0)$. Let $t'_p$ be the finite prefix of $t_0$ such that $\Pi_{\sim c}(t'_p) = t_p$. Since the original network $N$ is Hoare-monotone, we can extend the input to $\vec{\alpha}'; \Pi_c(t_0)$ and there will be some new trace of $N$, call it $t_1$, such that $t_1 \sqsupseteq t'_p$ and $\Pi_I(t_1) = \vec{\alpha}'; \Pi_c(t_0)$ and $\Pi_O(t_1) = \vec{\beta}_1 \sqsupseteq \vec{\beta}$ and with the relative ordering of events of $t_0$ preserved in $t_1$, and hence $\mathit{precedes}(b, c, t_1)$. If $\Pi_c(t_0)$ is infinite, then $\Pi_c(t_1) = \Pi_b(t_1)$, but if $\Pi_c(t_0)$ is finite, then we have no guarantee that $\Pi_c(t_1) = \Pi_b(t_1)$. In the rest of the discussion we describe how to build a trace $t' \in Trset(N)$, such that $t' \sqsupseteq t'_p$, $\Pi_O(t') = \vec{\beta}'$, $\Pi_I(t') = \vec{\alpha}'; \Pi_c(t')$, $\Pi_b(t') = \Pi_c(t')$ and with $\vec{\beta} \sqsubseteq \vec{\beta}'$ and $\mathit{precedes}(b, c, t')$. $\Pi_{\sim c}(t')$ will then be present in $Trset(M)$ and will demonstrate that $M$ is also Hoare-monotone.

The construction of $t'$ is carried out using traces from $Trset(N)$. Since we know that $Trset(N)$ is Hoare-monotone, we can construct $t'$ in successive approximations knowing that at each stage we can do it in such a way that the sequences of events on the ports increase at each stage of the construction. Let $t'_1$ be a prefix of $t_1$ upto the first new $b$-event in $t_1$. Then, by Hoare-monotonicity, there is a trace $t_2$, extending $t'_1$, with the input on input port $c$ extended by a new $c$-event that matches the new $b$-event in $h_1$, and with an extended output.

We iterate this process by successively considering all the $b$-events, and obtain, in this way, a sequence of traces $t_0, t_1 \ldots t_n \ldots$ such that the event sequences on all the ports are increasing and for which $\mathit{value}(\Pi_c(t_{n+1}))$ is the prefix of $\mathit{value}(\Pi_b(t_n))$ of length $n+1+$ (length of $\Pi_c(t_0)$). At any stage, if $\Pi_b(t_n) = \Pi_c(t_n)$, then the trace $t_n$ has the required property. However, it is possible that, when we look at the sequence of traces projected onto $c$, we obtain an infinite properly increasing chain of finite sequences. In this case, we have a sequence of inputs $\vec{\alpha'}; \Pi_c(t_i)$ and a corresponding sequence of outputs $\vec{\beta_i}$. By the limit-closure of $N$, we have a trace $t_\infty$ with input $\sqcup(\vec{\alpha'}; \Pi_c(t_i)) = \vec{\alpha'}; \sqcup\Pi_c(t_i)$ and outputs $\sqcup\vec{\beta_i}$ with the relative order of events from each $t_n$ being preserved. The sequence corresponding to port $b$ is $\sqcup\Pi_b(t_i)$ in $\sqcup\vec{\beta_i}$.

Any event in $\sqcup\Pi_c(t_i)$ is in some $\Pi_c(t_i)$, hence preceded by a corresponding $b$-event in the trace $t_i$, and hence preceded by a corresponding $b$-event in $\sqcup\Pi_b(t_i)$ (because relative order is preserved). Therefore, $\mathit{precedes}(b, c, t_\infty)$ holds and so $t_\infty$ is the required trace.

Now we prove that $Trset(M)$ is limit-closed. Suppose $(\vec{\alpha_1}, \vec{\beta_1}), \ldots$ are input-output pairs corresponding to traces $t_1, \ldots$ in $Trset(M)$, such that the relative ordering of events is preserved. Let $t'_1, \ldots$ be the corresponding traces in $Trset(N)$ such that $\mathit{immprecedes}(b, c, t'_i)$ holds for every $i$. It is then easy to see that the relative ordering of events is preserved along $t'_1, \ldots$. By limit-closure of $Trset(N)$, $\exists$ a trace $t_\infty$ with input $\sqcup(\vec{\alpha_i}; \Pi_c(t_i))$ and output $\sqcup\vec{\beta_i}$ with relative order of events in every $t_i$ being preserved in $t_\infty$. Since $\Pi_c(t_i) = \Pi_b(t_i)$ for every $i$, $\sqcup\Pi_c(t_i) = \sqcup\Pi_b(t_i)$ and as before, $\mathit{precedes}(b, c, t)$ holds. So $\Pi_{\sim c}(t)$ is a trace of $M$ with the required property.
∎

We now deal with output hiding.

*Theorem* 8. Any network obtained by hiding some output ports of a network with a Hoare-monotone and limit-closed trace set has a Hoare-monotone trace set.

**Proof :** Let $N$ be a network with a Hoare-monotone and limit-closed trace set, and let $M$ be the network obtained by hiding some set $S$ of output ports of $N$. Let $t$ be a trace of $M$, and let $t_p$ be a finite prefix of $t$. Let the input be $\vec{\alpha}$ and the output be $\vec{\beta}$. By theorem 5, there is a trace $t'$ of $N$ such that $t = \Pi_{\sim S}(t')$. Let $t'_p$ be the prefix of $t'$ such that $\Pi_{\sim S}(t'_p) = t_p$. The output in $t'$ is $\vec{\beta}; \vec{\gamma}$ where $\vec{\gamma}$ consist of the sequences of events at the ports in $S$. When $\vec{\alpha}$ is extended to $\vec{\alpha}'$, then by Hoare-monotonicity of $N$, there is a trace $s' \sqsupseteq t'_p$ with input sequences $\vec{\alpha}'$ and output sequences $\vec{\beta}'; \vec{\gamma}'$, $\vec{\beta} \sqsubseteq \vec{\beta}'$ and $\vec{\gamma} \sqsubseteq \vec{\gamma}'$. Therefore $\Pi_{\sim S}(s')$ is a trace of $M$ with the desired property.

■

*Corollary* 1. Any network consisting of components with Hoare-monotone and limit-closed trace sets has a Hoare-monotone trace set.

**Proof :** Any network implemented by components with Hoare-monotone and limit-closed trace sets is obtained by composing the components and then hiding some output ports. As proved earlier, any network obtained by composing components with Hoare-monotone and limit-closed components has a Hoare-monotone and limit-closed trace set. Hiding some output ports preserves Hoare-monotonicity of trace sets. Hence the resulting network has a Hoare-monotone trace set. ■

*Theorem* 9. No total subset of the input-output relation of fair merge can be implemented by any finite network of Hoare-monotone and limit-closed processes.

**Proof :** The trace set of any finite network built out of Hoare-monotone and limit-closed processes by aggregation, feedback and output hiding is Hoare-monotone, and hence the input-output relation is also Hoare-monotone. But no total subset of the IO-relation of fair merge is Hoare-monotone. To see this, consider the case where the input sequences of fair merge are $1^\infty$ and $\Lambda$, the empty sequence. Then $1^\infty$ is the only possible output sequence. Now if the sequence on the second input port is extended to 2, then the output sequence must include the value 2, and no such output sequence is an extension of $1^\infty$. ■

We can therefore distinguish between angelic merge and fair merge, since the following corollary is immediate.

*Corollary* 2. Angelic merge cannot implement the IO-relation of fair merge.

Note that, although fair merge is also not limit-closed, and limit-closure is preserved under composition, it might seem that we could prove the expressiveness theorem using the limit-closure property instead of Hoare-monotonicity. This is not possible, because infinity-fair merge can be obtained, as in the previous section, by composing some networks with Hoare-monotone and limit-closed trace sets and then hiding some output ports. But it is not limit-closed. Consider the inputs $(1, 2), (1^2, 2), (1^3, 2), \ldots$ to the infinity-fair merge process. $1, 1^2, 1^3, \ldots$ are possible outputs respectively. The limit of the inputs is $(1^\infty, 2)$ and the limit of the outputs is $1^\infty$, which is not a valid output sequence because 2 has not been output. This shows that we cannot just consider limit-closure, and we really need to talk about Hoare-monotonicity.

## 5.2 Smyth-monotonicity

Hoare-monotonicity described the property that given a trace $t$, if we consider an increased input, then there is a trace $t'$ with that increased input and an equal or increased output, and further the relative ordering of events in $t$ is preserved in $t'$. We now define a "similar" property. Informally, this property says that given a trace $t$, if we consider a decreased input, then there is a trace $t'$ with that decreased input and an equal or decreased output, and further the relative ordering of events in $t'$ is preserved in $t$.

*Definition* 25. $Trset(N)$ is said to be *Smyth-monotone* if for any trace $t$ with input sequences $\vec{\alpha}$ and output sequences $\vec{\beta}$, if $\vec{\alpha}' \sqsubseteq \vec{\alpha}$, then there is a trace $t'$ with input $\vec{\alpha}'$ and output $\vec{\beta}'$ and $\vec{\beta}' \sqsubseteq \vec{\beta}$ and the relative order of events in $t'$ is preserved in $t$.

The next lemmas establish that Smyth-monotonicity is preserved by network composition.

*Lemma* 1. Suppose $N_1$ and $N_2$ are networks with Smyth-monotone trace sets. Then their aggregate $N$ also has a Smyth-monotone trace set.

**Proof :** Let $t$ be a trace of $N$. Then $t$ is an interleaving of a trace $t_1$ of $N_1$ and a trace $t_2$ of $N_2$. Let $\vec{\alpha}$ and $\vec{\beta}$ be the input sequences and output sequences respectively in $t_1$. Let $\vec{\gamma}$ and $\vec{\delta}$ be the input sequences and output sequences respectively in $t_2$. Let $\vec{\alpha}' \sqsubseteq \vec{\alpha}$ and $\vec{\gamma}' \sqsubseteq \vec{\gamma}$. Then, by Smyth-monotonicity of $N_1$ and $N_2$, there are traces $t_3$ and $t_4$, of $N_1$ and $N_2$ respectively, such that the relative order of events in $t_3$ and $t_4$ are preserved in $t_1$ and $t_2$ respectively. Then an appropriate interleaving of $t_3$ and $t_4$ is a trace of $N$, the relative order of events of which are preserved in $t$. ∎

*Lemma* 2. If $N$ is a network with a Smyth-monotone trace set, and has an output port $p_1$ and an input port $p_2$, then the network $loop(p_1, p_2, N)$ has a Smyth-monotone trace set.

**Proof :** Let $M$ be the network $loop(p_1, p_2, N)$. We recall that

$$Trset(M) = \{\Pi_{\sim p_2}(t) | t \in Trset(N) \wedge precedes(p_1, p_2, t) \wedge (\Pi_{p_1}(t) = \Pi_{p_2}(t))\}$$

Let $t'$ be a trace of $M$ with $\vec{\alpha}$ and $\vec{\beta}$ as the input sequences and output sequences respectively. There is a trace $t$ of $N$, such that $t' = \Pi_{\sim p_2} t$ and $precedes(p_1, p_2, t)$ and $\Pi_{p_1}(t) = \Pi_{p_2}(t)$. Let $\delta = value(\Pi_{p_1}(t))$. Then the input sequences in $t$ are $\vec{\alpha}; \delta$. Let $\vec{\alpha}' \sqsubseteq \vec{\alpha}$. By Smyth-monotonicity of $N$, there is a trace $r_1$ of $N$, such that the input sequences are $\vec{\alpha}'; \delta$ and $value(\Pi_{p_1}(r_1)) = \delta_1 \sqsubseteq \delta$ and the relative order of events in $r_1$ are preserved in $t$.

If $\delta = \delta_1$ then we are done because then $\Pi_{\sim p_2} r_1$ is the desired trace of $M$. If not, we iterate as outlined below until we finally obtain the desired trace. At each stage of the iteration, we construct new traces such that the relative order of events in newer traces is preserved in the older traces.

The iteration proceeds as follows: If $\delta \neq \delta_1$ then $\delta_1$ is a proper prefix of $\delta$. This implies that $\vec{\alpha}'; \delta_1$ is a prefix of $\vec{\alpha}'; \delta$. So, by Smyth-monotonicity of $N$, there is a trace $r_2$ with input sequences $\vec{\alpha}'; \delta_1$ and $\mathit{value}(\Pi_{p_1}(r_2)) = \delta_2 \sqsubseteq \delta_1$ and $\mathit{precedes}(p_1, p_2, r_2)$.

If we repeat the above step, we will reach $\delta_i = \delta_{i+1}$ and then we will be done. This procedure will terminate because the prefix ordering is a well-ordering and $\delta_i$ must certainly hit $\emptyset$ and stabilize if it does not stabilize earlier. ■

*Lemma* 3. Any network obtained by hiding some output ports of a network with a Smyth-monotone trace set has a Smyth-monotone trace set.

**Proof :** Let $N$ be a network with a Smyth-monotone trace set, and let $M$ be the network obtained by hiding some set $S$ of output ports of $N$. Let $t$ be a trace of $M$. Let the input be $\vec{\alpha}$ and the output be $\vec{\beta}$. By theorem 5, there is a trace $t'$ of $N$ such that $t = \Pi_{\sim S}(t')$. The output in $t'$ is $\vec{\beta}; \vec{\gamma}$ where $\vec{\gamma}$ consist of the sequences of events at the ports in $S$. Let $\vec{\alpha}' \sqsubseteq \vec{\alpha}$. Then by the Smyth-monotonicity of $N$, there is a trace $s' \in \mathit{Trset}(N)$ with input $\vec{\alpha}'$ and output $\vec{\beta}'; \vec{\gamma}'$, such that the relative order of events in $s'$ is preserved in $t'$. Then $\Pi_{\sim S}(s')$ is a trace of $M$ with the desired property. ■

We now define a predicate on input-output relations that is true for any network whose trace set has the Smyth-monotonicity property. We will refer to this predicate as Smyth-monotonicity too.

*Definition* 26. The input-output relation of a network $N$ is said to be **Smyth-monotone**, if whenever for some input $\vec{\alpha}$, there is an output $\vec{\beta}$, and $\vec{\alpha'} \sqsubseteq \vec{\alpha}$, then there is a possible output $\vec{\beta'} \sqsubseteq \vec{\beta}$.

*Lemma* 4. No total subset of the IO-relation of angelic merge is Smyth-monotone.

**Proof :** Consider the following three pairs of sequences as inputs:

1. $\langle 1^\infty, 2^\infty \rangle$,
2. $\langle 1^\infty, \Lambda \rangle$,
3. $\langle \Lambda, 2^\infty \rangle$.

In the second and third cases, the output of the angelic merge is determined. Thus, any total subset of the IO-relation of angelic merge must

$$\begin{tikzcd}
\text{fair} \arrow[d, shift right] \\
\text{angelic} \arrow[u, shift right, "/" marking] \arrow[d, shift right] \\
\text{infinity-fair} \arrow[u, shift right, "/" marking]
\end{tikzcd}$$

Figure 5: A Straight-line Picture of Merges

produce exactly the same output as angelic merge would; namely $1^\infty$ and $2^\infty$ respectively. Now we consider the possible outputs in the first case. Note that the input in this case extends the inputs in both cases two and three. Thus if the IO-relation is to be Smyth-monotone *all* the possible outputs must extend both $1^\infty$ and $2^\infty$; this is obviously impossible. ∎

*Theorem* 10. No total subset of the IO-relation of angelic merge can be implemented by any finite network of processes with Smyth-monotone trace sets.

**Proof :** Smyth-monotonicity of trace sets is preserved under aggregation, feedback and output hiding. Therefore the trace set of any finite network of Smyth-monotone processes must be Smyth-monotone, and hence the network must have a Smyth-monotone input-output relation. But no total subset of the IO-relation of angelic merge can have a Smyth-monotone input-output relation. ∎

We can therefore distinguish between infinity-fair merge and angelic merge, since the following corollary is immediate.

*Corollary* 3. Infinity-fair merge cannot implement the IO-relation of angelic merge.

## 5.3 Implementability results

The results in the previous subsection show that infinity-fair merge cannot implement angelic merge, and that angelic merge cannot implement fair merge. Eugene Stark showed how one could implement infinity-fair merge using angelic merge [33]. We will now show how one could implement angelic merge using fair merge, thus providing a very nice straight-line picture of expressibility (figure 5).

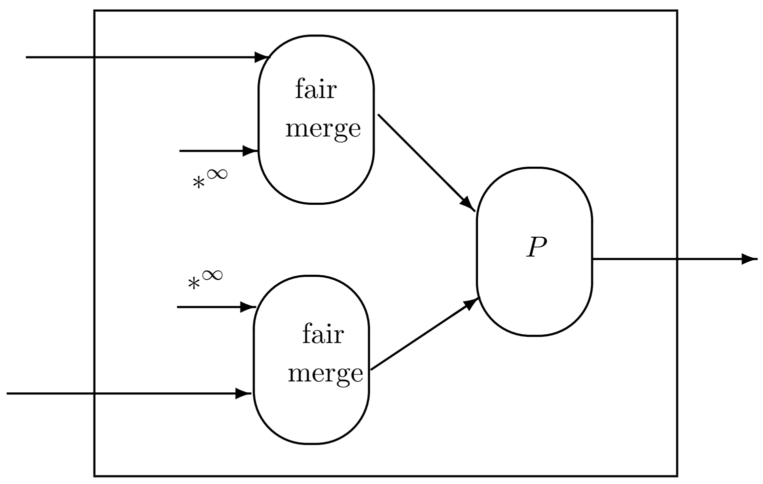

Figure 6: Angelic Merge from Fair Merge

Figure 6 shows the network that implements angelic merge. The process $P$ is a determinate process whose behaviour can be described as follows : it reads values from its first input port and outputs the values that are not $*$, until it reads two successive $*$'s, in which case it switches to reading from its second input port and reads and outputs values that are not $*$ until it reads two successive $*$'s, in which case it switches back to the first input port and continues as above. We could formally describe this as an automaton, but the behaviour should be clear from the given description.

## 6 Conclusions

In this paper we have described a trace semantics for indeterminate dataflow networks and used it to prove expressiveness theorems. We have shown that there are three levels of *unbounded indeterminacy* and these are characterized by different monotonicity properties. We make no claims that this is an exhaustive classification, indeed we have discovered related expressiveness results that refine the ones discussed in the present paper [19]. The main significance of the results derives from the significance for order theoretic approaches to fixed-point semantics for dataflow networks. The fact that different monotonicity properties are obeyed suggests that different approaches to fixed-point semantics might be appropriate at the different levels. Thus, for example, one may not need as elaborate an approach if one is only interested in modeling infinity-fair merge and not fair merge.

Recent work by Samson Abramsky [3] and E. W. Stark [32] has in fact shown that there are aproaches that give fixed-point semantics for dataflow networks that include infinity-fair merge and angelic merge respectively. The problem of describing a fixed-point principle for dataflow networks with fair merge remains open.

## Acknowledgements

We have benefitted from useful discussions with Samson Abramsky, Steve Brookes, Elsa Gunter, Nax Mendler, Albert Meyer, Alex Rabinovich, Jim Russell and especially E. W. Stark. We acknowledge the support of NSF grants DCR-8602072 and CCR-8817909.

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

[3] S. Abramsky. A generalized kahn principle for abstract asynchronous networks. In *Proceedings of the 1989 Symposium on Mathematical Foundations of Programming Language Semantics*, 1989.

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

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

[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 Bjoerner, editor, *Formal Description of Programming Concepts II*, pages 125–148. North-Holland, 1983.

[8] J. W. deBakker, J. A. Bergstra, J. W. Klop, and J.-J. Ch. Meyer. Linear time and branching time semantics for recursion with merge. *Theoretical Computer Science*, 34:134–156, 1984.

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

[10] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of the Sixteenth Annual ACM Symposium On Principles Of Programming Languages*, 1989.

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

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

[13] R. M. Keller and P. Panangaden. Semantics of networks containing indeterminate operators. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 479–496, 1985. LNCS 197.

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

[15] J. Kok. A fully abstract semantics for dataflow nets. In *Proceedings of Parallel Architectures And Languages Europe 1987*, pages 351–368, Berlin, 1987. Springer-Verlag.

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

[17] 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.

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

[19] D. McAllester, P. Panangaden, and V. Shanbhogue. Nonexpressibility of fairness and signaling. In *Proceedings of the 29th Annual Symposium of Foundations of Computer Science*, 1988.

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

[21] 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.

[22] 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.

[23] David Park. On the semantics of fair parallelism. In *Proceedings of the Winter School on Formal Software Specification*, pages 504–526, New York, 1980. Springer-Verlag. Lecture Notes In Computer Science 86.

[24] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[25] G. D. Plotkin. A powerdomain for countable nondeterminism. In *Proceedings of the Ninth ICALP*. Springer-Verlag, 1982. LNCS 140.

[26] G. D. Plotkin. Personal communication to the first author. Remark attributed to M. Smyth in 1976 by G. D. Plotkin., 1990.

[27] A. Rabinovich and B. A. Trakhtenbrot. Nets of processes and dataflow. To appear in Proceedings of ReX School on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS, 1988.

[28] J. Russell. *Full Abstraction and Fixed-Point Principles for Indeterminate Computation*. PhD thesis, Cornell University, 1989.

[29] V. Shanbhogue. *The Expressiveness of Indeterminate Dataflow Primitives*. PhD thesis, Cornell University, 1990.

[30] 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.

[31] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, pages 221–269, 1989.

[32] E. W. Stark. A simple generalization of Kahn's principle to indeterminate dataflow networks. Technical Report TR 89-29, SUNY at Stony Brook, Computer Science Dept, Dec 1989.

[33] E. W. Stark. On the relations computable by a class of concurrent automata. In *Proceedings Of The Fifteenth Annual ACM Symposium On Principles Of Programming Languages*, New York, 1990. IEEE. Also Technical Report TR 88-09, SUNY at Stony Brook, Computer Science Dept.