McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Monotone and Nonmonotone Dataflow Networks

Marija Čubrić
Prakash Panangaden

ACAPS Technical Memo 81
December 8, 1993

Research supported in part by NSERC and FCAR.

Abstract

The expressiveness of concurrent dataflow primitives have been studied from the point of view of automata theory [19] as well as in terms of abstractly given processes defined in terms of traces [20]. The expressiveness of the merge primitives, fair merge, angelic merge and infinity-fair merge were shown to hinge on monotonicity properties of their trace languages. From the point of view of concurrent automata an important class of automata called monotone automata were defined and it was shown that fair merge could not be implemented by such an automaton.

In this paper we study the relationship between monotonicity properties of automata and of trace languages. We characterize the trace languages computed by monotone automata and we show that the characterization does not correspond to the existing monotonicity properties of trace languages. These, relatively easy, results led us to discover a new indeterminate primitive, the fair stack, that turns out to be nonmonotone and to have interesting expressiveness properties. We show that a fair stack is strictly weaker than fair merge but provably incomparable with angelic merge. These results suggest that it is time to study seriously the class of nonmonotone automata.

Contents

List of Figures

List of Tables

1 Introduction

The general theory of asynchronous systems has reached a state of maturity comparable to the much more extensively studied theory of process calculi. The fundamental studies of Stark [27, 28], Arnold and Nivat [2], Mazurkiewicz [15], Pratt [22], Kok [12], Kwiatkowska [13], Jonsson [7] and Rabinovich and Trakhtenbrot [23], just to mention an arbitrary subset, have placed the theory on firm foundations.

Among the possible models of asynchronous computation a particularly pleasing one is the dataflow model. We have a clear conceptual model, originally due to Kahn and McQueen [9, 8], we understand how to formalize the model in operational terms using port automata [14] and show that the Kahn principle is correct with respect to this model. We have generalizations to indeterminate networks [10, 1] and we have expressiveness results that classify the power of different primitives [19, 20, 17]. The result of Jonsson [7, 24] shows that traces form a fully-abstract semantics for such networks.

The most important expressiveness result shows that there is a class of automata, called "monotone" automata that enjoy nice properties but which exclude simple primitives such as fair merge [19]. One of the main results of the paper by Panangaden and Stark was to show that the input-output relation of monotone automata satisfied a monotonicity property. Since fair merge fails to have this property the expressiveness result follows. However, there was no precise characterization of what trace languages are computed by these networks. In this paper we settle this by defining a new monotonicity property of trace languages and showing that all monotone automata produce trace languages satisfying this property and also the reverse, namely that any trace language satisfying this property is indeed the trace language of some monotone automaton.

In an analysis of processes defined by trace languages, Panangaden and Shanbhogue classified the relative expressive power of different variants of fair merge, namely angelic merge and infinity-fair merge. Those results hinged on two different monotonicity properties of trace languages, called Hoare and Smyth monotonicity. It turns out that the monotonicity property that we introduce in this paper is different from either of these. To illustrate this we exhibit a new primitive called "fair stack" which has a Hoare monotone trace language but which cannot be implemented by any monotone automaton. The expressive power of fair stack turns out to be curious, it is incomparable to angelic merge; in other words neither one can implement the other. Both fair stack and angelic merge are weaker than fair merge and stronger than infinity-fair merge. These expressiveness results are proved using a new compositional property of dataflow networks called *f-continuity.*

In the context of asynchronous transition systems (ATS) Shields [26] has studied the relation between trace languages and ATSs. A key point to be noted is that with ATSs one does not have a notion of input and output thus one does not have the same class of phenomena that one studies in the case of IO or port automata. The classifications of behaviors of various types of automata in terms of fairness [13, 11] address orthogonal issues to the properties that we discuss. Finally classification based on geometrical properties [6] address general issues of concurrency and causality and not the interaction between input and output events which seem to lie at the heart of our studies.

To conclude this introductory section we recapitulate the informal definitions of the merge primitives. A merge primitive has two input ports and a single output port. The tokens that arrive at the input ports are transmitted unchanged to the output ports and these are the only tokens produced. The ordering of the tokens at a given port is preserved on the output but there is no ordering between the tokens that arrive on different input ports. If both input ports receive tokens output must be produced. In a *fair merge* every input token appears on the output port eventually. In *angelic merge*, if one of the input ports receives a finite number of tokens then every token on the other port is guaranteed to be output, thus if both ports receive finite sequences the angelic merge behaves exactly as fair merge would but if one of the input sequences is infinite the behavior may not be equitable. The *infinity-fair merge* is dual to angelic merge. If a port receives infinitely many tokens all the input at the other port is guaranteed to be output. If both inputs are infinite it will behave as fair merge does. If one of the inputs is finite it will produce a finite output. The terminology is due to David Park [21] who was implicitly aware that they must be different.

2 Background

The basic automata that we work with are called *port automata*. Port automata [14, 20, 18] are a special class of *input/output automata* [14]. They communicate with the environment through a fixed set of input and output ports. The automata have states and transitions between states. The observable aspects of the behavior correspond to the *events*. These are pairs (p, v), where p is a port name and v is a value from some set of possible values. Such an event corresponds to the appearance of the datum v at the port p and represents the observable communication between an automaton and its environment. The set of possible events forms the *alphabet* of an automaton. In a general IO automaton the alphabet is not assumed to be structured into port-value pairs. Port automata are more concrete than IO automata and none of the discussion that we have in the present paper would be fundamentally altered by working with general IO automata. To capture the behavior of such automata the following properties are usually assumed:

- All events on a single port are either causally dependant or disable each other.
- The order of events on different input ports cannot be observed, hence, all events on different input ports are concurrent.
- If two concurrent events are enabled at the same state of an automaton, they will both be executed eventually. (This property is sometimes [13] called *concurrency fairness*).

- The environment cannot be prevented from supplying a new input to an automaton. This is called *receptivity.*

The upshot of these assumptions is that processes are reactive and receptive. The input is assumed to be buffered, thus the fact that processes are receptive does not mean that a given process needs to react in a nontrivial way to all input. The basic causality and conflict relations between events are axiomatized below.

Definition 1 A *concurrent alphabet* is a pair $(E, \|)$, where E is any set, and $\|$ is a symmetric and irreflexive binary relation on E, called the concurrency relation.

We will use the notation $\mathcal{E}$ for the concurrent alphabet $(E, \|)$.

Definition 2 Let V be a set of data values, and $P^{inp} \cap P^{out} = \emptyset$ two finite sets of names. A *port signature* is a set E of the form $E = E^{inp} \cup E^{out}$ where $E^{inp} = P^{inp} \times V$ and $E^{out} = P^{out} \times V$.

Elements of E^{inp},E^{out}, P^{inp}, P^{out} are called respectively, input events and output events, input ports and output ports. For any event $e = (p, v)$ we write $\mathrm{port}(e) = p$ and $\mathrm{value}(e) = v$ for its corresponding port and value component. We use the notation E^p to denote all events on the port p and we use a letter P for the union $P^{inp} \cup P^{out}$.

Next we define some operations with concurrent alphabets.

Definition 3 If $\mathcal{E}_1$ and $\mathcal{E}_2$ are concurrent alphabets, then their union $\mathcal{E}_1 \cup \mathcal{E}_2$ is a concurrent alphabet $(E_1 \cup E_2, \|_1 \cup \|_2 \cup \{(e_1, e_2) \in (E_1^{inp} \times E_2^{inp}) \cup (E_1^{out} \times E_2^{out}) : \mathrm{port}(e_1) \neq \mathrm{port}(e_2)\})$.

Sometimes we may want to conceal ports.

Definition 4 If $\mathcal{E}$ is a concurrent alphabet and X is a set of ports, then $(E \backslash X, \|)$ is the concurrent alphabet obtained by removing all events with the port in X from E and all pairs involving such events from $\|$.

2.1 Port Automata

The following definition of port automata is similar to the one given in [14, 20, 18]. For the sake of simplicity, we omit the notion of internal events from the definition of port automata.

Definition 5 A *port automaton* is a 4-tuple $M = (\mathcal{E}, Q, q^\imath, \rightarrow)$ where

- E is a port signature.
- $\mathcal{E} = (E, \|)$ is a concurrent alphabet, which satisfies the following properties:

Single port dependency For all $e, e' \in E$: if $e \| e'$ and $\text{port}(e) = \text{port}(e')$, then $e = e'$.

Buffered input For all input events e, e': if $\text{port}(e) \neq \text{port}(e')$, then $e \| e'$.

- Q is a set of states, and $q^\imath \in Q$ is a distinguished initial state,
- $\rightarrow$ is a transition function; i.e. a partial function of type $Q \times E \rightarrow Q$, which satisfies the following properties:

Receptivity For any state q and any input event a, there exists a state r such that $q \xrightarrow{a} r$.

Commutativity For any state q and any events a, b such that $a \| b$, if $q \xrightarrow{a} r$ and $q \xrightarrow{b} r'$ then, there exists a state q' such that $r \xrightarrow{b} q'$ and $r' \xrightarrow{a} q'$.

We say that event e is *enabled* at the state q if there exists a state r, such that $q \xrightarrow{e} r$. In that case, we sometimes write qa for the state r. Note that if two events a and b are concurrent we **do not imply** that if one is enabled in a given state the other has to be as well. A key difference between asynchronous transition systems [26] and port automata can be indicated here. In ATSs one assumes that if a and b are concurrent and the sequence ab is possible in a given state then ba must also be possible. A simple example of a dataflow primitive that violates this is *left-strict OR*.

A very important point is the following. The concurrency relation is part of the definition of an automaton. It is not a derived concept obtained from the transition function. Thus, if, for example, there are two output events a and b such that when either one is executed both a and b are enabled in the resulting state it does not necessarily follow that a and b are concurrent. It could be the case that a conflict and that the choice between a and b is being made available in the new state. This makes phrases like "the same transition is enabled in the next state" quite subtle. In order to make such statements rigorous one must appeal to the concurrency relation. If a and b are concurrent then, in the situation just outlined, the "same" b transition is enabled after the a transition but not otherwise. A rigorous treatment is based on the notion of *residual* [19, 27, 3] adapted from the lambda calculus [4]. For our purposes we will not need the formal definitions of residuals or the algebra of residuals but the concept is important.

We now introduce two important classes of automata. We say that a port automaton is *monotone* if it satisfies the following property:

Monotonicity If some output event b is enabled at the state q, then for any input event a, event b is also enabled at the state qa.

The intuition here is that "arrival of new input cannot disable an already enabled output transition". Thus monotone automata rule out timeouts and interrupts. These are important in real programs but are painful to reason about; for a very good reason, they introduce nonmonotonicity in a fundamental way.

We say that a port automaton is *determinate* if it is monotone and it satisfies the following property in addition:

Determinacy If two different output events b and b', are both enabled at the same state then, $b \| b'$.

This prevents the presence of internal choice in an automaton. It is straightforward to show that if an automaton satisfies this property the input-output relation is indeed a (continuous) function. From now on, we will use the word *automaton* to mean a *port automaton.*

2.2 Networks of Automata

Networks of automata are built by linking ports of different automata. Here (as in [20, 19]) we use three operations on automata to build more complex (networks of) automata. The question of what exactly constitutes the "right" set of primitive operations has been carefully addressed by Stark [29].

Aggregation of automata M and N is a new automaton $M \| N$ obtained by putting M and N side by side (no shared ports).

Feedback of automata M is a new automaton $loop(M, p_1, p_2)$ obtained from M by linking its output port p_1 to its input port p_2.

Hiding Automaton $M \backslash X$ is obtained by hiding a set of output ports X of automaton M.

Ordinary sequential composition can be obtained by aggregation followed by feedback. For every of this three operations, certain compatibility conditions have to be defined, otherwise, the operation will not necessarily produce a port automaton. These compatibility conditions are spelled out, for example, in [20]. They merely serve to ensure that there are no conflicts in port names and that interconnection is port-to-port and there is no implicit merging of output ports.

Definition 6 Let M_1, M_2 be a two automata. Then, the *aggregation* of M_1 and M_2 is the automaton $M_1 \| M_2 = (\mathcal{E}_1 \cup \mathcal{E}_2, Q_1 \times Q_2, (q_1^\imath, q_2^\imath), \rightarrow)$ where: $q \xrightarrow{a} r$ iff $\forall i\ ((a \in E_i \Rightarrow q_i \xrightarrow{a} r_i) \wedge (a \notin E_i \Rightarrow q_i = r_i))$.

Definition 7 Let $M = (\mathcal{E}, Q, q^\imath, \rightarrow)$ be an automaton and let $\theta : E^{p_1} \rightarrow E^{p_2}$ be an isomorphism of events on ports p_1 and p_2. Then the *feedback* of M on (p_1, p_2) is the automaton $loop(M, p_1, p_2)$ $= (\mathcal{E} \backslash (E^{p_2} \times V), Q, q^\imath, \rightarrow_1)$ where: $q \xrightarrow{a}_1 r$ iff $(\mathrm{port}(a) \neq p_1 \wedge q \xrightarrow{a} r) \vee (\mathrm{port}(a) = p_1 \wedge \theta(a) = a' \wedge \exists s (q \xrightarrow{a} s \xrightarrow{a'} r))$

Definition 8 Let $M = (\mathcal{E}, Q, q^\imath, \rightarrow)$ be an automaton and let $X \subseteq P^{out}$. Then the *output-hiding* of X in M is the automaton $M \backslash X = (\mathcal{E} \backslash (X \times V), Q, q^\imath, \rightarrow)$.

Proposition 1 [18] The classes of monotone and determinate port automata are both closed under the operations of aggregation, hiding and feedback.

From now on, we will use the term *network of automata* to denote an automaton resulting from composing finitely many automata, using the above three operations. We write $\prod M_i$ for network consisting of finitely many automata $M_1, M_2, \ldots, M_n$. It is important to keep in mind that a network of automata is just another automaton.

2.3 Completed computations and traces

In this section we give the basic definitions of computations and traces.

Definition 9 A *computation sequence*, γ, of an automaton M is any finite or infinite sequence of transitions $q^\imath \xrightarrow{e_1} q_2 \xrightarrow{e_2} \ldots$ where each transition is a transition of M.

The set of all computations of an automaton M is written $\mathrm{Comp}(M)$. A compositional description of the processes described by automata is given by ignoring the intermediate states and recording only the events.

Definition 10 The *trace of a computation sequence* $\gamma = q^\imath \xrightarrow{e_1} q_2 \xrightarrow{e_2} \ldots$ is a sequence $\mathrm{tr}(\gamma) = e_1 e_2 \ldots$ consisting of the input and output events in γ.

The set of all traces of an automaton M is written $\mathrm{Traces}(M)$. If γ is a finite computation sequence of length n, we use the following notation: $q^\imath \xrightarrow{\mathrm{tr}(\gamma)} q_n$ to indicate that γ starts at the state $q^\imath$ and finishes at the state q_n, having a trace $\mathrm{tr}(\gamma)$. [1] We say that some event e is

[1] We will make always clear from the context, whether the symbol $\rightarrow$ is used for a transition function, or for a finite derivation.

enabled at the state r of some trace $t = \mathrm{tr}(\gamma)$ if e is enabled at r in the computation sequence γ. Similarly, we say that event e is executed in the trace t if it is executed in γ.

The following definition captures the notion of so-called "concurrency fairness". Notice that we do not expect fair behavior with respect to all continuously enabled events, but only for those which are continuously enabled **and concurrent with** all other enabled events.

Definition 11 A computation sequence γ is *completed* iff it is either finite, and no output event is enabled at its final state, or it is infinite and for no event e, does there exist a finite computation sequence δ such that $\gamma = \delta\gamma'$ and e is enabled at every state of γ' but never executed or disabled in γ'.

According to this definition it is possible that an output event a may be in conflict with an output event b on a different output port. It may happen that when a occurs the events a and b are both enabled in the new state and the choice between them has to be made afresh. If you just look at the transition table, it may seem that a and b are concurrent. In fact, the concurrency relation has to be given as primitive. Just because two transitions appear to commute does not mean they are concurrent. In this circumstance it is possible for a to happen repeatedly and for b to never happen and for the computation to be deemed completed. In this case we take the view that b is getting disabled and reenabled every time a is executed. It is only if a and b are deemed to be concurrent that b must happen eventually. An example of this phenomenon is a dataflow primitive called a *splitter*. Here one has a single input port and two output ports. Each input token is placed on one or other of the two output ports. It is entirely possible for the choice to be made consistently in favor of one of the two output ports.

Next, we define a *completed computation sequence of a network of automata*. We use the notation $\gamma|_X$ for the projection of the computation sequence γ onto the set of events X. We use the same notation for the projection of a trace. Also, we use the notation $t|_{\sim X}$ to stand for $t|_{E \setminus X}$. The set of all completed computations of an automaton M is written $\mathrm{Compl}(M)$.

Proposition 2 A computation sequence γ of the network of automata $\prod M_i$ is completed iff for all i, $\gamma|_{E_i} \in \mathrm{Compl}(M_i)$.

In [19] it was shown that for monotone automata the set of completed computations is the set of maximal computations in a certain preorder, to be defined below. Unfortunately this does not carry over to networks that may include non-monotone automata.

Definition 12 The *trace set* of an automaton M is the following set:

$$\mathrm{Tr}(M) = \{t \in E^\infty : t = \mathrm{tr}(\gamma),\ \gamma \in \mathrm{Compl}(M)\}.$$

2.4 The Permutation Preorder

In reasoning about computations or traces or concurrent alphabets in general the usual order structure on sequences, i.e. the prefix ordering, is not appropriate because of the ability to commute concurrent events. One can define a preordering (and induced equivalence relation) on sequences by using the permutation of concurrent events. In fact the actual sequences are not as important as the equivalence classes under permutation. The basic ideas here are due to Mazurkiewicz.

We use A^* and A^∞ to denote the set of all finite and respectivelly - finite and infinite words over the alphabet A.

We define *permutation equivalence* [15], on the set of finite strings over the concurrent alphabet $\mathcal{E} = (E, \|)$, to be the least congruence $\equiv_\mathcal{E}$ with respect to concatenation, such that $ab \equiv_\mathcal{E} ba$ iff $a \| b$, where $a, b \in E$. We say that two strings $w, w' \in E^*$ are permutation equivalent $w \equiv_\mathcal{E} w'$ iff one can be obtained from the other by permuting adjacent, concurrent symbols. From now on, we use the notation $\sqsubseteq$ for the usual prefix ordering and we omit the subscript $\equiv_\mathcal{E}$ when the concurrent alphabet is clear from context.

We first define the permutation preorder on finite words as: $w \precsim w'$ iff $\exists w''(ww'' \equiv w')$ and then we extend it to infinite words as follows: $w \precsim w'$ iff $\forall w_1 \sqsubseteq w (\exists w_1' \sqsubseteq w'(w_1 \precsim w_1'))$. Now, we extend the relation $\equiv_\mathcal{E}$ on infinite words: $w \equiv w'$ iff $w \precsim w' \wedge w' \precsim w$ From now on, $\overline{E}$ will denote the quotient set $E^\infty/_\equiv$ with the induced partial order. It is well known [18] that $\overline{E}$ is a *dI domain*.

The following key fact, which we use later on, was established using the calculus of residuals and it follows from the Levy's lemma in trace theory ([15]).

Proposition 3 [18] For all $t, t' \in E^\infty$ and for all $a, b \in E$, if $ta, tb \precsim t'$ then $a \| b$.

We use $\pi_\sqsubseteq(X)$ to denote the set of all $\sqsubseteq$-prefixes for the set $X \subseteq E^\infty$, and similarly for $\pi_\precsim(X)$.

It has been proved in [19] that maximality with respect to the permutation order coincides with the notion of fairness for monotone automata. There, fairness is defined to be a *weak fairness*, meaning, roughly speaking, the following: if a component is continuously willing to make progress, then it will be allowed to do so eventually. It would be pleasant if the definition of completed computation sequence (11) allows one to prove an analogous result for network of general automata. Unfortunately, however, this is true only for a network of a single, (non-monotone) automaton. Roughly speaking the difficulty is that when one has nonmonotone components in the network, then permutations that are harmless in the monotone case may cause the disabling of events in the nonmonotone case. The following proposition states what one can do with a single automaton.

Proposition 4 [19, 25] The computation sequence γ of M is completed iff γ is $\precsim$-maximal among all computation sequences of M having the same input history.

Unfortunately, the same is not true for a network consisting of arbitrary automata, thus, the definition 2. (For more details see [20]). Another pleasant consequence of the definition of completed computation sequences is the following proposition:

Proposition 5 [25] If γ is a computation sequence of M, then it can be $\precsim$- extended to a completed computation sequence with the same input history. Moreover, if γ is finite, then the extension can be chosen to be an extension in the prefix ordering [25].

This gives us a handle on dealing with the computations of nonmonotone automata.

2.5 Traces are Fully Abstract

The fact that traces give a fully-abstract semantics for dataflow networks was first established by Jonsson in his doctoral dissertation but was largely overlooked until Kok developed a different fully-abstract semantics [12]. The result was proved independently by several other groups, for example Rabinovich and Trakhtenbrot proved it with a different notion of observation. Kok's contribution was a simple universal context, built using fair merge, that served to distinguish any two networks with different traces; this was used by Jonsson to simplify his original result. Jim Russell showed that one could construct a universal discriminating context using purely determinate primitives. In this section we quickly review the presentation from Shanbhogue's thesis which differs in minor details from Jonsson's presentation.

The key point is that in order to have full abstraction one should close the set of traces under permutations. Consider the following two primitives. In each case there is one input port and two output ports called b and c. The output ports each receive a copy of the input tokens. In the first case the automaton always performs the output event on port b first whereas in the second case the automaton performs the output event first on b or c alternately. These are both determinate and quite distinct as automata and they have different trace sets. However, a moment's thought shows that no context can possibly distinguish these two automata if we are observing the input-output relation. Thus the set of traces needs to be closed under certain permutations.

Definition 13 Let $\mathcal{E}$ be a concurrent alphabet and let $X \subseteq E^{\infty}$. Then, $\mathrm{Cl}(X) = \{w \in E^{\infty} : w \equiv_{\mathcal{E}} w' \text{ for some } w' \in X\}$.

It is not difficult to see that Cl is a closure operator on $(E^{\infty}, \subseteq)$. It has been proved in [25] that closure sets of traces are compositional i.e. the following equalities hold:

$$\mathrm{Cl}(\mathrm{Tr}(M\|N)) = \{t \in E^{\infty} : t|_M \in \mathrm{Cl}(\mathrm{Tr}(M)) \wedge t|_N \in \mathrm{Cl}(\mathrm{Tr}(N))\}$$
$$\mathrm{Cl}(\mathrm{Tr}(loop(M, p_1, p_2))) = \{(t|\ _{E^{p_2}} : t \in \mathrm{Cl}(\mathrm{Tr}(M)) \wedge t|_{\theta(E^{p_1})} = t|_{E^{p_2}} \wedge prec(p_1, p_2, t)\}$$
$$\mathrm{Cl}(\mathrm{Tr}(M \backslash X)) = \{t|\ _{E^X} : t \in \mathrm{Cl}(\mathrm{Tr}(M))$$

where, we use the abbreviation $prec(p_1, p_2, t)$ to denote that occurrence of every event a' such that $\mathrm{port}(a') = p_2$ is preceded by the corresponding occurrence of event a on the port p_1, where $\theta(a) = a'$.

The observable aspect of the behavior of automata is the *input-output relation.* A port history of M is a vector of values over the set V^{∞}, indexed by port names. Every trace t determines a port history H_t, where for each port p, the projection of H_t onto the port p is a sequence of values $H_t^p = \mathrm{value}(t|_{E^p})$. We write H_t^{inp}, H_t^{out} for the port histories of t restricted to input and output ports respectively. We will use the notation $\sqsubseteq$ for the pointwise ordering among port histories. Also, if $\vec{x} \in (V^{\infty})^n$, and $b \in E$, we will often use the notation $\vec{x}b$ to denote the port history $\vec{x}'$ such that $\vec{x}'_i = \vec{x}_i$, for all $i \neq \mathrm{port}(b)$, and $\vec{x}'_j = \vec{x}_j\mathrm{value}(b)$ for $\mathrm{port}(b) = j$.

Since the domain $\overline{E}$ is isomorphic to the cartesian product $\overline{V}^n$, (see [18]), where $\overline{V}$ is a domain of value sequences and n is the number of input and output ports, it is clear that $\overline{V}^n$ is a dI domain. If $L \subseteq E^{\infty}$, then we write $\mathrm{Rel}(L)$ to denote the following set: $\{(H_w^{inp}, H_w^{out}) : w \in L\}$.

Definition 14 The *input-output relation* of an automaton M is the following set: $\mathrm{IO}(M) = \mathrm{Rel}(\mathrm{Tr}(M))$

The following propositions are due to Lynch and Stark.

Proposition 6 [14] If M is a determinate automaton, then $\mathrm{IO}(M)$ is continuous function.

Proposition 7 [18] If function $f : (V^{\infty})^m \rightarrow (V^{\infty})^n$ is continuous, then there exists a determinate automaton M such that $\mathrm{IO}(M) = f$.

It was shown in [12, 7, 23, 24] that two automata have the same input-output relation in every network context iff the closure of their trace sets are the same. This result is known as the *full-abstraction* theorem for trace semantics.

3 Monotone Automata and Monotone Trace Languages

In this section we state and prove the characterization theorem for the trace languages generated by monotone automata. We begin by introducing the notion of Hoare-monotone trace languages which was used to show that angelic merge was strictly less expressive than fair merge. We will show that monotone trace languages are always Hoare monotone.

In what follows, we assume that $L = \mathrm{Tr}(M)$ for some automaton M.

Definition 15 L is *Hoare-monotone* iff $\forall w \in L(\forall \alpha \in (E^{inp})^{\infty}(w|_{E^{inp}} \precsim \alpha \Rightarrow \exists w' \in L(w'|_{E^{inp}} \equiv \alpha \ \wedge \ w \precsim w')))$.

The intuition is that given any trace for any process, one can always increase (in the portwise prefix ordering) the input and ask about the resulting traces. For a Hoare-monotone process there will always be some trace with the increased input which is above the original trace in the permutation preorder on traces. Since the input and output are monotone functions of the trace we see that such a trace will have output that is extended as well. It is easy to see that fair merge is not Hoare monotone. The above property was defined, in an equivalent but more clumsy fashion in [20, 25] using an extension of $\sqsubseteq$- ordering on ports as follows:

$$w \prec w' \text{ iff } \ \forall p \in P((H_w^p \sqsubseteq H_{w'}^p) \wedge (\exists \phi : w \overset{1-1}{\rightarrow} w' \text{ which is order preserving}))$$

One can similarly define a limit-completeness property:

Definition 16 L is *continuous* iff L is Hoare-monotone and for every set $\{t_i\}_{i \in I} \in L$, the following property holds : $\forall i(t_i \prec t_{i+1}) \Rightarrow \exists t \in L(H_t = \bigsqcup H_{t_i} \wedge \forall i(t_i \prec t))$.

It has been proved in [20] that *continuity* defined as above is a compositional property with respect to aggregation and feedback. But, as it is pointed out in the same paper, hiding of ports in a network consisting of continuous components, will result in a network which is Hoare-monotone, but not necessarily continuous.

Proposition 8 If, for all i, $\mathrm{Tr}(M_i)$ is continuous trace sets, then $\mathrm{Tr}(\prod M_i)$ is a Hoare-monotone trace set.

Knowing this allows one to establish that no network using angelic merge (which can be defined as a Hoare-monotone and continuous process) can emulate fair merge.

In [19] it was shown, in effect, that all monotone automata have Hoare-monotone input-output relations; below we will show that they, in fact, have Hoare-monotone trace sets. It is not true, however, that any Hoare-monotone trace set correspond to the trace set of some automaton. First we characterize the trace sets of monotone automata via the following definition.

Definition 17 A trace language L is *monotone* iff $\forall a \in E^{inp}(\forall b \in E^{out}(wb, wa \in \pi_{\sqsubseteq}(L) \Rightarrow wab \in \pi_{\sqsubseteq}(L)))$.

This definition emulates the automata-theoretic definition of monotone automaton. The fact that all monotone automata turn out to have monotone trace sets is not surprising. It is less obvious that all monotone trace languages have to be the trace set of some monotone automaton.

Theorem 1 M is a monotone automaton iff $\mathrm{Tr}(M)$ is a monotone trace language.

Proof: Direction ($\Rightarrow$) is trivial. Now, suppose that $\mathrm{Tr}(M)$ is a monotone trace language. Let input event a and output event b be both enabled at some state q of M, and let $w \in \mathrm{Traces}(M)$ be any trace of M having q as its final state. Then, using 5, we have that $wa, wb \in \pi_{\sqsubseteq}(\mathrm{Tr}(M))$, which implies that $wab \in \pi_{\sqsubseteq}(\mathrm{Tr}(M))$. Now, using 3, it must be that $a \| b$. Using commutativity of M, it implies that b is enabled at the state qa ∎

Compositionality of this property follows directly from 1 and 1.

Proposition 9 If, for all i, $\mathrm{Tr}(M_i)$ is monotone trace sets, then $\mathrm{Tr}(\prod M_i)$ is a monotone trace set.

Next, we prove that monotonicity is a stronger property than Hoare-monotonicity. Later on, we will see that this inclusion is strict. Before doing this we prove an auxiliary lemma. This lemma gives an important characterization of monotone automata. It says that in monotone automata, portwise prefix ordering induces a $\precsim$- ordering.

Lemma 1 *M is a monotone automaton iff $(\forall w, w' \in \mathrm{Traces}(M))(\forall p \in P(H_w^p \sqsubseteq H_{w'}^p \Rightarrow w \precsim w')))$.*

Proof: Suppose input event a and output event b are enabled at some state q of M. Let w be any trace with q as its final state. Then, $H_{wa} \sqsubseteq H_{wba}$ and $wa \precsim wba$. Using the property 3 we have that $a \| b$. But this is what it means for M to be monotone.
Now suppose that M is known to be monotone. Suppose that for some traces w, w', we have $H_w \sqsubseteq H_{w'}$. Suppose w is finite. Let $w_0 \sqsubseteq w'$ be the minimal prefix of w' such that $H_w^{inp} = H_{w_0}^{inp}$. Since M is monotone, we can move to the right all output events from w_0 which are not in w starting from the last one. Thus, we obtain $w'' \equiv w'$ and $w \sqsubseteq w''$. So we have that $w \precsim w'$. If w is infinite, then for every finite prefix $w_1 \sqsubseteq w$, using the same procedure as in the finite case, we can find $w_1' \equiv w_1$ such that $w_1 \precsim w_1'$, which implies $w \precsim w'$ ∎

Proposition 10 If $\mathrm{Tr}(M)$ is monotone, then it is Hoare-monotone.

Proof: It has been proved in [19] that the input-output relation of a monotone automaton is monotone. Now, Hoare-monotonicity of the corresponding trace set is a direct consequence of lemma 1 ∎

4 Nonmonotone Automata and their Expressiveness

In order to show that Hoare monotonicity fails to capture the behavior of monotone automata we have to have an example of an automaton with a Hoare-monotone trace set but which does not have any monotone automaton equivalent to it; i.e. with the same (permutation closure of its) trace set. Clearly such a demonstration requires the type of characterization theorem we established in the previous section.

The example we discovered is called a *fair stack.* A stack is a port automaton with one input and one output port. Values, from some value set V are pushed into the stack at the input port and popped out at the output port, using the Last-In-First-Out rule. It is not difficult to see that a stack violates the monotonicity requirement. The execution of the input event "push a" will disable the already enabled, output event "pop b", where b is the value on the top of the stack.

Although a stack satisfies the determinacy property, since the concurrency relation is empty, and at every state of the stack, only one output event is enabled ("pop a", where "a" is the value on the top of the stack), because of the nonmonotone behavior, a stack has an input-output relation which is not a function.

For infinite input sequences, one can impose a variety of fairness constraints. We define completed computational sequences of a fair stack to be either finite traces which finish at the initial state (empty stack), or infinite traces, for which, the initial state is visited infinitely many times i.e the stack is emptied infinitely many times.

To define the trace set of a fair stack we use an auxiliary (partial) function depth : $E^* \to \mathbf{N}$. We write a to denote an input event (i, a), and $\overline{a}$ to denote the output event (o, a), where i, o are input and output port names respectively. This function simply computes the depth of the stack, with respect to the given trace. It is defined as follows: $\mathrm{depth}(\epsilon) = 0$, $\mathrm{depth}(ua\overline{a}v) = \mathrm{depth}(uv)$, $\mathrm{depth}(w) = |w|$, where: $u, v \in E^*$ and $w \in (E^{inp})^*$. Now, we define the trace set of fair stack (for which we use the abbreviation FS) to be the following set:

$$\mathrm{Tr}(\mathrm{FS}) = \{t \in E^* : \mathrm{depth}(t) = 0\} \cup \{t \in E^\omega : \forall u \sqsubseteq t(\exists v(u \sqsubseteq v \sqsubseteq t \wedge \mathrm{depth}(v) = 0))\}$$

In the following, we summarize some interesting properties of the fair stack.

Lemma 2 If $t, t' \in \mathrm{Tr}(FS)$ and $H_t \sqsubseteq H_t'$, then $t \sqsubseteq t'$.

Proof: The proof is by induction on the number of input events in t ∎

One should not think that the above lemma is in the conflict with lemma 1, since here we use only completed traces, not arbitrary traces as in 1. Take, for example, traces ab and $\overline{a}b$. Obviously, $ab \sqsubseteq ab$ and $\epsilon \sqsubseteq \overline{a}$ but $ab \not\sqsubseteq \overline{a}b$.

Proposition 11 Tr(FS) is Hoare-monotone, continuous, but not monotone.

Proof: First we show that Tr(FS) is Hoare-monotone. Let $t \in$ Tr(FS) and $\alpha' = t|_{E^{inp}} a_1 a_2 \ldots$, where $\{a_i\}$ is a (possibly infinite) sequence of input events. Let $t' = t a_1 \overline{a}_1 a_2 \overline{a}_2 \ldots$. Then, obviously $t|_{E^{inp}} \sqsubseteq t'|_{E^{inp}}$, $t \sqsubseteq t'$ and $t' \in$ Tr(FS)

Next, we show that $Tr(M)$ is continuous. Let $\{t_i\}$ be an infinite sequence from Tr(FS) such that $t_i \prec t_{i+1}$, for all i. Using the lemma 2, we have that $\{t_i\}$ is a prefix increasing sequence. By the definition of Tr(FS) for infinite traces, it follows that $\bigsqcup t_i \in$ Tr(FS). Since $\sqsubseteq$ ordering implies $\prec$- ordering, we have that $t_i \prec t$, for all i

The fact that Tr(M) is not monotone is immediate from the following: $w\overline{b}, wa \in \pi$(Tr(FS)), and $wa\overline{b} \notin \pi$(Tr(FS)) ∎

Corollary 1 The trace language Tr(FS) cannot be generated by any monotone automaton.

Corollary 2 Hoare-monotone trace languages are a proper subclass of monotone trace languages.

Given that FS is not a monotone automaton one is tempted to assume that it is more "expressive" than any monotone automaton. In fact the situation is quite complicated. In order to analyze the expressive power of FS we will need a new compositional property.

Definition 18 A trace language L is *f-continuous* iff $\exists f \subseteq$ Rel(L) which is continuous.

Similarly, we can define an *f-monotone* language.

Theorem 2 If, for all i, Tr(M_i) is f-continuous trace sets, then Tr($\prod M_i$) is a f-continuous trace set.

To prove the above theorem, we need to prove few auxiliary lemmas. In what follows, we assume that M, M', N, N' are arbitrary automata.

Lemma 3 Let Tr(M) be f-continuous and let $f \subseteq IO(M)$ be a continuous function. Then, there exists a determinate automaton M_1 such that Tr(M_1) $\subseteq$ Cl(Tr(M)) and $f = IO(M_1)$.

Proof: Let $M = (\mathcal{E}, Q, q^\imath, \rightarrow)$ and $M_1 = (\mathcal{E}, Q_1, q_1^\imath, \rightarrow_1)$, where $Q_1 = \mathrm{Traces}(M)/\equiv_{\mathcal{E}}$, $q_1^\imath = [\epsilon]$, and transition function is defined as follows. If $a \in E^{inp}$ then $[w] \xrightarrow{a}_1 [wa]$. If $b \in E^{out}$ then $[w] \xrightarrow{b}_1 [wb]$ providing that $H_{wb}^{out} \sqsubseteq f(H_{wb}^{in})$. It is not difficult to prove that M_1 is a determinate port automaton and that $\mathrm{IO}(M_1) = f$. Now, we prove that $\mathrm{Tr}(M_1) \subseteq \mathrm{Cl}(\mathrm{Tr}(M))$. Suppose $t \in \mathrm{Tr}(M_1)$ and t is finite. By definition of M_1 we have that there exists $w \in \mathrm{Traces}(M)$ and $w \equiv t$. Since $f(H_t^{inp}) = H_t^{out}$ we have that $f(H_w^{inp}) = H_w^{out}$. But, $f \in \mathrm{IO}(M)$, so it must be that $w \in \mathrm{Tr}(M)$. Suppose that t is infinite. Then $t = \bigsqcup w_i$ and for all i, $w_i \in \mathrm{Traces}(M_1)$. By definition of M_1 we have that there exists an infinite sequence $w_i' \in \mathrm{Traces}(M)$ and $w_i' \equiv w_i$, for all i. Using residuals technique [18], we can even find a prefix increasing sequence $w_i'' \in \mathrm{Traces}(M)$ and $w_i'' \equiv w_i$. Let $w = \bigsqcup w_i''$. It is clear from the construction of w that $w \equiv t$ and $w \in \mathrm{Traces}(M)$. Suppose that w is not completed. For this, let b be an output event which is continuously enabled in w, starting from some finite prefix w_1, where $w = w_1 w'$, concurrent with all other events in w', but never executed in w'. By the construction of trace w we have that the same must be true for b in t, starting from some prefix $w_i \sqsubseteq t$. But, this is not possible since t is completed in M_1. So, $w \in \mathrm{Tr}(M)$ ∎

Lemma 4 If, for every i, $\mathrm{Tr}(M_i') \subseteq \mathrm{Cl}(\mathrm{Tr}(M_i))$, then $\mathrm{Tr}(\prod M_i') \subseteq \mathrm{Cl}(\mathrm{Tr}(\prod M_i))$

Proof: Straightforward, using the equalities on the page 9 ∎

Lemma 5 For every M there exists M_1 such that $\mathrm{Cl}(\mathrm{Tr}(M)) = \mathrm{Tr}(M_1)$

Proof: Let $M = (\mathcal{E}, Q, q^\imath, \rightarrow)$ and $M_1 = (\mathcal{E}, Q_1, q_1^\imath, \rightarrow_1)$, where $Q_1 = \mathrm{Traces}(M)/\equiv_{\mathcal{E}}$, $q_1^\imath = [\epsilon]$, and transition function is defined as follows: $[w] \xrightarrow{a}_1 [w']$, for every $a \in E$, providing that $wa \equiv w'$. It is not difficult to verify that $\mathrm{Cl}(\mathrm{Tr}(M)) = \mathrm{Tr}(M_1)$ ∎

Lemma 6 If $\mathrm{Tr}(M) \subseteq \mathrm{Cl}(\mathrm{Tr}(N))$ then $\mathrm{IO}(M) \subseteq \mathrm{IO}(N)$

Proof: Using the lemma 5 there exists an automaton N_1 such that $\mathrm{Cl}(\mathrm{Tr}(N)) = \mathrm{Tr}(N_1)$. So, $\mathrm{Tr}(M) \subseteq \mathrm{Tr}(N_1)$. Then, $\mathrm{IO}(M) \subseteq \mathrm{IO}(N_1)$ and $\mathrm{IO}(N_1) = \mathrm{IO}(N)$ ∎

Proof of the theorem 2
Suppose that $f \subseteq \mathrm{IO}(M)$ and $g \subseteq \mathrm{IO}(N)$ are continuous functions. Then, using the lemma 3 there exist determinate automata M' and N' such that $\mathrm{Tr}(M') \subseteq \mathrm{Cl}(\mathrm{Tr}(M))$ and $\mathrm{IO}(M') = f$ and similarly for N' and g. Using the lemma 4 we have that $\mathrm{Tr}(M' \times N') \subseteq \mathrm{Cl}(\mathrm{Tr}(M \times N))$. Then, using the last lemma 5 we have that $\mathrm{IO}(M' \times N') \subseteq \mathrm{IO}(M \times N)$. Since M' and N'

are determinate, so is their composition $M' \times N'$, which implies (see 6) that $\mathrm{IO}(M' \times N')$ is a continuous function ∎

We are now ready to analyze the expressiveness of FS. We use the abbreviations: FM, AM, IFM, to denote respectively, *fair merge, angelic merge* and *infinity fair merge* (see [20, 21]) We use the notation $A > B$ to denote the following relation: $A \rightarrow B$ and $B \not\rightarrow A$, where the symbol "$\rightarrow$" stands for "implements", meaning that one can find a network consisting of finitely many copies of A and finitely many determinate automata, which has the same input-output relation as B. Note that the notion of "implements" is as weak as possible which means that if one proves a nonimplementability result it will automatically imply nonimplementability with respect to any stronger relation that one is interested in[2]

In the table1 we show which primitives have which properties. Since all the properties listed in the table 1 are compositional, (see 9,2,[20]) they can be used to prove "non-implementability" results.

	FM	AM	FS	IFM
Hoare-monotonicity	no	yes	yes	yes
Continuity	no	yes	yes	yes
Monotonicity	no	yes	no	yes
f-continuity	no	no	yes	yes

Table 1: Monotonicity properties for different indeterminate dataflow primitives

The easy expressiveness result is the following.

Proposition 12 FM > FS > IFM

Proof: Figure 1 shows how one can implement FS using FM. (Symbol "p" is interpreted as the "pop" operation and symbol "c" as "clear until the end of the stack" operation). Now, suppose that FS can implement FM. Since, the trace set of FS is continuous, as well as the trace set of any determinate primitive, this will imply that FM has a Hoare-monotone trace set, by 8. But this is not true [19, 20]. Thus, we have FM > FS. In the figure 2 we can see an implementation of IFM using FS. (Every continuous sequence of zeros of the length n is interpreted as the number n. For nonimplementability part, we use the fact that IFM is monotone [19], while FS is not 11 ∎

The interesting question is the relation between *angelic merge* and *fair stack*. Using the fact that monotonicity and f-continuity are compositional properties, we have proved the following:

[2]This obvious remark is made because in the past we have heard comments like "why not study bisimulation instead of just the IO relation?" thus showing that it needs to be said.

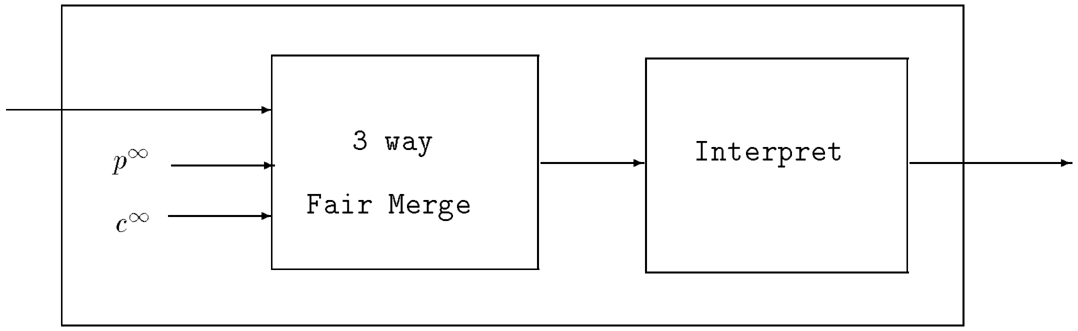

Figure 1: FM $\rightarrow$ FS

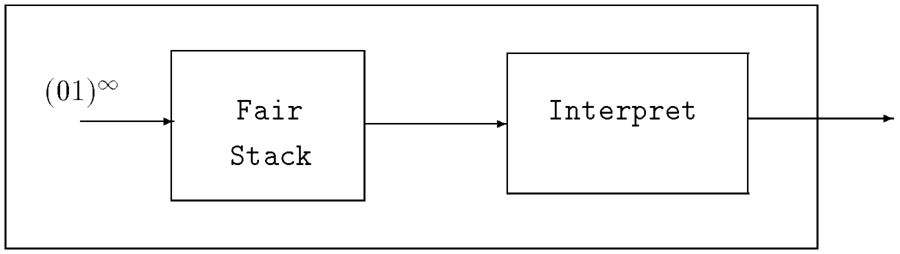

Figure 2: FS $\rightarrow$ IFM

Proposition 13 FS $\not\rightarrow$ AM and AM $\not\rightarrow$ FS.

Proof: Suppose that AM can implement FS. Since AM is monotone [19], then it must be that FS is monotone too, which is not true 11. Since IO(FS) includes the identity function, it is clear that FS is f-continuous. Now, to prove that FS cannot implement AM, it is enough to show that AM is not f-continuous. Suppose the opposite, i.e there exists $f \in$ IO(AM) which is continuous. For any input events a, b we have the following: $(a, \epsilon, a), (\epsilon, b, b) \in f$ (because of receptivity), but neither $(a, b, ab) \in f$ nor $(a, b, ba) \in f$, since f is monotone. Thus, AM cannot be f-monotone, so it cannot be f-continuous either. ∎

5 Conclusions

In this paper we have examined a new indeterminate dataflow primitive and studied its expressive power. The interesting aspect of this primitive is that it has a Hoare-monotone trace set but cannot be expressed by any monotone automaton. The expressiveness results show that we are not close to understanding the structural theory of the class of nonmonotone automata. So far we have generalized Kahn's principles for various classes of monotone automata only [1, 28] and not for nonmonotone automata without introducing auxiliary constructs like hiatons [5, 21, 16]. Perhaps such an approach is the only one possible but we need theorems that say that. The fact that common systems programming idioms like timeouts and interrupts are nonmonotone shows that one really needs to provide mechanisms to reason about these kinds of primitives. The results of this paper provide a first step towards organizing our classification of nonmonotone automata. The concept of f-continuity may suggest some inductive reasoning principles that may work for at least some nonmonotone automata.

References

[1] S. Abramsky. A generalized Kahn principle for abstract asynchronous networks. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the Fifth Workshop on Mathematical Foundations of Programming Semantics*, number 442 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[2] A. Arnold and M. Nivat. Formal computations of non-deterministic recursive program schemas. *Math. System Theory*, 13:219–236, 1980.

[3] P. Bachmann and Phan Minh Dung. Nondeterministic computations - structure and axioms, Elektron. *Inf. verarb. Kybern.*, (22):243–261, 1986.

[4] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* Studies in Logic. North-Holland Publishing Company, Amsterdam, revised edition, 1984.

[5] F. Boussinot. Proposition de sémantique dénotationelle pour des réseaux de processus avec operateur de mélange équitable. *Theoretical Computer Science*, 18:173–206, 1982.

[6] Jeremy Gunawardena. Causal automata. *Theoretical Computer Science*, 101(2):265–288, 20 July 1992.

[7] Bengt Jonsson. A fully abstract trace model for dataflow networks. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–165, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[8] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, 1977.

[9] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 471–475, 1974.

[10] Robert M. Keller and Prakash Panangaden. Semantics of networks containing indeterminate operators. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science, pages 479–496, Carnegie-Mellon University, Pittsburgh, Pennsylvania, July 9–11, 1984. Springer-Verlag.

[11] N. Klarlund. *Progress Measures and Finite Arguments for Infinite Computations.* PhD thesis, Cornell University, 1990.

[12] Joost N. Kok. Denotational semantics of nets with nondeterminism. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming, ESOP '86*, number 213 in Lecture Notes in Computer Science, pages 237–249, Saarbrücken, West Germany, March 17–19, 1986. Springer-Verlag.

[13] Marta Kwiatkowska. *Fairness for Non-Interleaving Concurrency.* PhD thesis, University of Leicester, May 1989.

[14] N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, July 1989.

[15] Antoni Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Application and Relationships to Other Models of Concurrency*, number 255 in Lecture Notes in Computer Science, pages 279–324. Springer-Verlag, 1986.

[16] P. Panangaden and V. Shanbhogue. McCarthy's AMB cannot implement fair merge. In K. V. Nori, editor, *Proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 338 in Lecture Notes in Computer Science, pages 348–363. Springer-Verlag, 1988.

[17] P. Panangaden, V. Shanbhogue, and E. W. Stark. Stability and sequentiality in dataflow networks. In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium On Automata, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 308–321, Warwick University, England, July 16–20, 1990. Springer-Verlag.

[18] P. Panangaden and E. W. Stark. Computations, residuals and the power of indeterminacy. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium On Automata, Languages and Programming*, number 317 in Lecture Notes in Computer Science, pages 348–363. Springer-Verlag, Tampere, Finland, July 11–15, 1988.

[19] Prakash Panangaden. Abstract interpretation and indeterminacy. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, number 197 in Lecture Notes in Computer Science, pages 497–511, Carnegie-Mellon University, Pittsburgh, Pennsylvania, July 9–11, 1984. Springer-Verlag.

[20] Prakash Panangaden and Vasant Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1):99–131, May 1992.

[21] D. Park. The "fairness problem" and non-deterministic computing networks. In J. deBakker and L. van Leeuwen, editors, *Proceedings of the Fourth Advanced Course on Foundations of Computer Science: Distributed Systems*, pages 133–161. Mathematisch Centrum, 1982.

[22] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[23] A. Rabinovich and B. A. Trakhtenbrot. Nets of processes and dataflow. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency – Proceedings of a School/Workshop (REX Workshop)*, number 354 in Lecture Notes in Computer Science, pages 574–602. Springer-Verlag, Noordwijkerhout, The Netherlands, May 30–June 3, 1988.

[24] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium of Foundations of Computer Science*, pages 170–177, 1989.

[25] V. Shanbogue. *The Expressivness of Indeterminate Dataflow Primitives.* PhD thesis, Cornell University, 1990.

[26] M. W. Shields. Behavioural presentations. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency – Proceedings of a School/Workshop (REX Workshop)*, number 354 in Lecture Notes in Computer Science, pages 673–689, Noordwijkerhout, The Netherlands, May 30–June 3, 1988. Springer-Verlag.

[27] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64(3):221–269, May 1989.

[28] E. W. Stark. Connections between a concrete and an abstract model of concurrent systems. In *Mathematical Foundations of Programming Language Semantics*, number 442 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[29] E. W. Stark. Calculus of dataflow networks. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 125–136, Santa Cruz, California, June 22–25, 1992. IEEE Computer Society and ACM/SIGACT.