Computations, Residuals, and the Power of Indeterminacy

Prakash Panangaden[*]
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

Eugene W. Stark[†]
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA

Abstract

We investigate the power of Kahn-style dataflow networks, with processes that may exhibit indeterminate behavior. Our main result is a theorem about networks of "monotone" processes, which shows: (1) that the input/output relation of such a network is a total and monotone relation; and (2) every relation that is total, monotone, and continuous in a certain sense, is the input/output relation of such a network. Now, the class of monotone networks includes networks that compute arbitrary continuous input/output functions, an "angelic merge" network, and an "infinity-fair merge" network that exhibits countably indeterminate branching. Since the "fair merge" relation is neither monotone nor continuous, a corollary of our main result is the impossibility of implementing fair merge in terms of continuous functions, angelic merge, and infinity-fair merge.

Our results are established by applying the powerful technique of "residuals" to the computations of a network. Residuals, which have previously been used to investigate optimal reduction strategies for the λ-calculus, have recently been demonstrated by one of the authors (Stark) also to be of use in reasoning about concurrent systems. Here, we define the general notion of a "residual operation" on an automaton, and show how residual operations defined on the components of a network induce a certain preorder $\precsim$ on the set of computations of the network. For networks of "monotone port automata," we show that the "fair" computations coincide with $\precsim$-maximal computations. Our results follow from this extremely convenient property.

1 Introduction

We are concerned with networks of communicating processes, like those considered by Kahn [16, 17], but in which processes are allowed to have indeterminate behavior. Thus, we view a

*Research supported in part by NSF Grant DCR-8602072.
†Research supported in part by NSF Grant CCR-8702247.

network as a graph, whose vertices are processes, and whose arcs are FIFO communication channels. The processes communicate with each other by passing messages, which contain data values, along the arcs. We classify processes by the types of branching they may contain in their program. *Unrestricted* processes may contain arbitrary branching, including both the ability to make indeterminate internal choices between a countable number of computation paths (so-called "countable indeterminacy" or "countable nondeterminism"[2, 3, 4, 5]), and the ability to test for the presence and absence of input data. *Monotone* processes may make arbitrary internal choices, and although they may branch based on input data that has already arrived, they may not contain tests for the absence of input data. This restriction means that the behavior of monotone processes is completely independent of the times at which inputs arrive during computation.

An interesting class of processes with indeterminate behavior are the so-called *merge* processes. A *fair merge* process combines sequences of values arriving on two input channels into a single sequence, in such a way that the output sequence produced in any complete computation is always a fair shuffle of the two input sequences. The *angelic merge* and *infinity-fair* merge processes [23] perform a function similar to that of the fair merge, but do not necessarily transmit all values that arrive on both inputs. Instead, they both satisfy the basic requirement that the output sequence should be a fair shuffle of prefixes (possibly all) of the input sequences. In addition to this basic requirement, an angelic merge process guarantees that if the sequence arriving on one input is finite, then it will transmit the entire sequence of values that arrives on the *other* input. An angelic merge process therefore never gets "stuck" waiting for input that might never arrive. An infinity-fair merge process supplements the basic requirement with the guarantee that if the sequence arriving on one input is *infinite*, then it will transmit the entire sequence of values that arrive on the other input.

It is well known that the presence of fairness implies the ability to make countably indeterminate choices (see, for example, [3] or [24]), and it is easy to demonstrate this using König's Lemma arguments. The converse, whether fairness can be programmed if one has a primitive for countably indeterminate choice, is not so clear. It is not difficult to program infinity-fair merge with such a primitive [3]. In the case of nondeterministic recursive programs, which are closely related to the dataflow networks considered here, it is known that with McCarthy's **amb** primitive [10] one can produce countable branching and also angelic merge, but it has not been shown that one can program a fair merge with **amb** [1]. In this paper, we show that countable indeterminacy alone is not sufficient for fairness; it is also necessary to branch based on the availability of input data. More precisely, we show that networks of monotone processes can compute arbitrary continuous input/output functions, angelic merge, and infinity-fair merge (which requires countably indeterminate branching), but that no such network can implement fair merge.

We should emphasize that our results concern the *implementability* of relations by a certain class of mechanisms; they are not directly concerned with the *definability* of relations in terms of other relations. Indeed, a satisfactory abstract theory of definability of relations by nondeterministic recursive programs is not available at present, except in the special case that the underlying value domains are flat. However, if one accepts the premise that any such abstract theory ought to give results consistent with the operational model used here, then our results can be interpreted as saying something indirectly about definability.

Our results, concerning the relative power of merging primitives, are a byproduct of a more general study of networks of "dataflow-like" processes with indeterminate behavior. Our main tool in this study is a formalism developed in [25]. We define three classes of automata, starting with a very general, abstract class, and becoming successively more specialized and concrete. The first class, called simply *automata*, is essentially the same as the "labeled transition systems" that have been used in the study of CCS and CSP (*e.g.* in [8, 9, 13, 21]). Computations of such automata consist of sequences of "transitions," each of which is labeled with a symbol, called an "event." Next, we define *port automata*, which represent processes that receive data values from other processes through "input ports," and send data values to other processes through "output ports." Port automata are a special case of the "input/output automata" defined by Lynch and Tuttle [20], and the "I/O-systems" of Jonsson [15]. By imposing on port automata a condition stating that enabled output transitions cannot be disabled by the arrival of input, we obtain the class of *monotone port automata*. We shall see that monotone port automata are an extremely well-behaved class of indeterminate processes. Essentially the same class of automata was defined in a somewhat more abstract setting in [25], but was not thoroughly investigated there.

After defining the various kinds of automata, we show how to "compose" a collection of component automata into a *network automaton*, which represents a system of concurrently executing processes in which communication and synchronization takes place through shared events. Although the composition of an arbitrary collection of automata always results in an automaton, the same is not true for port automata. We therefore define a "compatibility" condition on collections of port automata, such that the composition of a compatible collection of port automata always results in a port automaton. Our notion of compatibility, and our definition of composition of port automata are special cases of the corresponding notions defined by Lynch and Tuttle [20] for input/output automata. Having defined the notion of a network of port automata, we define the "fair computations" and the "input/output relations" of such networks. We also define when a network "implements" a relation.

The heart of the paper is the definition of a *residual operation* on an automaton. A residual operation is a partial binary operation $\uparrow$ on the set of transitions of an automaton, subject to a few simple axioms. Such an operation serves, in essence, to point out which pairs of transitions from a state are "concurrent," and to show how concurrent transitions "commute." Naturally associated with each class of automata is a corresponding kind of residual operation, whose definition exploits the particular commutativity properties of that class. We show how a residual operation on an automaton induces a preorder $\precsim$, extending the usual prefix relation, on the set of its computations. The main result of this section shows that the set of equivalence classes of computations of an automaton, under the induced partial order, is an algebraic cpo whose finite elements are exactly the equivalence classes of finite computations.

We then use residuals as a tool to investigate the properties of networks of monotone port automata. For such networks, we show that the fair computations are exactly the computations that are $\precsim$-maximal among all computations with the same input history. A corollary of this result states that every computation $\precsim$-extends to a fair computation with the same input history. We apply these results to show that the input/output relations of networks of monotone automata are total and monotone relations. Conversely, every relation that is total, monotone, and continuous in a suitable sense is the input/output relation of a network of monotone port automata.

Residuals have been used previously in the investigation of optimal reduction strategies for the λ-calculus [6, 19] and term-rewriting systems [7, 14]. In that work, residuals are used to keep track of what happens to one redex in a term while others are contracted. Our use is entirely analogous—a residual operation lets us keep track of what happens to one transition while other transitions are executed concurrently. The use of residuals in reasoning about concurrent systems was demonstrated in [25].

Before proceeding with the presentation of our results, we comment on notation. In this paper, all sets whose cardinality is left unspecified are assumed to be at most countable. If V is a set, then V^* and V^∞ denote, respectively, the of all finite sequences from V, and the set of all finite and infinite sequences from V. The set V^* is a monoid under concatenation, and a partially ordered set under the prefix relation $\preceq$. The set V^∞ is a *Scott domain* (*i.e.* an ω-algebraic, bounded-complete poset) under the prefix ordering. If the notation V^U denotes, as usual, the set of all functions from U to V, then the set $(V^*)^U$ inherits the monoid structure and partial order "argumentwise" from V^*, and the set $(V^\infty)^U$ similarly inherits the structure of a domain from V^∞. We use the symbol $\preceq$ to denote the argumentwise ordering on V^* and V^∞.

2 Automata

An *automaton* is a tuple $M = (A, Q, q^\iota, \rightarrow)$, where

- A is a set of *events*, called the *event signature* of M.
- Q is a set of *states*, with $q^\iota \in Q$ a distinguished *initial state*.
- $\rightarrow$ is the *transition relation*, and is a subset of the set $Q \times (A \cup \{\epsilon\}) \times Q$, whose elements are called *transitions*. Here ϵ is a special symbol, not in A, called the *identity event*. If $t = (q, a, r) \in \rightarrow$, then we write $t : q \xrightarrow{a} r$, or $q \xrightarrow{a} r$, or just $t : q \rightarrow r$, when the event a is unimportant. The transition relation of M is required to satisfy:

  (Identity) For all states $q, r \in Q$, we have $(q, \epsilon, r) \in \rightarrow$ iff $q = r$.

Intuitively, a transition $t : q \xrightarrow{a} r$, with $a \neq \epsilon$, represents a step in which M performs event a in state q, and changes state from q to r. If $a = \epsilon$, then t does not represent a step of M. Although the condition (Identity) ensures that the presence of such transitions has no computational significance other than providing a way to "pad" computations, these transitions will play an indispensible technical role in the sequel.

If $t : q \xrightarrow{a} r$, then the state q is called the *domain* of t, and is denoted $dom(t)$. The state r is called the *codomain* of t, and is denoted $cod(t)$. We use $event(t)$ to denote the event a of t. A transition $q \xrightarrow{\epsilon} q$ is called an *identity transition*, and is denoted by id_q. Transitions t and u are called *coinitial* if $dom(t) = dom(u)$. We say that a set T of transitions is *enabled* in state q if there exists a transition $t \in T$ with $dom(t) = q$. An event a is *enabled* in state q if the set of all transitions for event a is enabled in state q.

A *finite computation* for an automaton is a finite sequence γ of transitions in $\rightarrow$, of the form:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n.$$

The number n is called the *length* $|\gamma|$ of γ. Similarly, an *infinite computation* is an infinite sequence of transitions in $\rightarrow$:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots.$$

We extend notation and terminology for transitions to computations, so that if γ is a computation, then the *domain* $dom(\gamma)$ of γ is the state q_0, and if γ is finite, then the *codomain* $cod(\gamma)$ of γ is the state q_n. We write $\gamma : q \rightarrow r$ to assert that γ is a finite computation with domain q and codomain r. A computation γ is *initial* if $dom(\gamma)$ is the distinguished start state q^ι. The *trace* of γ is the subsequence of $a_1 a_2 \dots$ consisting of the non-identity events in γ. If $\gamma : q \rightarrow r$ and $\delta : q' \rightarrow r'$ are finite computations, then γ and δ are called *composable* if $q' = r$, and we define their *composition* to be the finite computation $\gamma\delta : q \rightarrow r'$, obtained by concatenating γ and δ and identifying $cod(\gamma)$ with $dom(\delta)$.

2.1 Port Automata

We now define a particular kind of automaton, called a "port automaton," that communicates by sending "data values" through "ports."

Formally, let V be a fixed set of *data values*, which we assume to contain at least the set of natural numbers. A *port signature* is an event signature A of the form $A = A^{\text{in}} \cup A^{\text{out}}$, where $A^{\text{in}} = X \times V$, $A^{\text{out}} = Y \times V$, and $X \cap Y = \emptyset$. The elements of $P = X \cup Y$ are called *ports*, with the elements of X called *input ports* and the elements of Y called *output ports*. Similarly, the elements of A^{in} are called *input events* and the elements of A^{out} are called *output events*. If $a = (p, v) \in A$, then we write $port(a)$ for the port component p, and $value(a)$ for the value component v, of a.

A *port automaton* is an automaton $M = (A, Q, q^\iota, \rightarrow)$, where

- A is a port signature.
- The transition relation of M satisfies:

  (Receptivity) For all states q and input events a, there exists a unique state r such that $q \xrightarrow{a} r$.

  (Commutativity) For all states q and all input events a and b with $port(a) \neq port(b)$, if $q \xrightarrow{a} r \xrightarrow{b} s$ and $q \xrightarrow{b} r' \xrightarrow{a} s'$, then $s = s'$.

Intuitively, an output event a for a port automaton represents the transmission of value $value(a)$ on output port $port(a)$. An input event a represents the receipt of value $value(a)$ on input port $port(a)$. The receptivity condition means that a port automaton is always willing to accept input. The commutativity condition means that a port automaton is insensitive to the order of arrival of successive inputs at distinct ports.

A port automaton is *monotone* if it satisfies the additional property:

(Monotonicity) For all states q and r, all input events a, and all output events b, if $q \xrightarrow{b} r$, $q \xrightarrow{a} q'$, and $r \xrightarrow{a} r'$, then $q' \xrightarrow{b} r'$.

Intuitively, this condition means that output transitions, once enabled, are never disabled by the arrival of additional input.

For the purposes of this paper, it is convenient to state the receptivity, commutativity, and monotonicity conditions in the somewhat abstract, but relatively simple form above, rather than in terms of somewhat messy concrete assumptions about the structure of states. As a particular concrete model of the axioms, we think of an automaton whose state set is of the form $Q^o \times (V^*)^X$, where Q^o is a set of "internal states," and $(V^*)^X$ is a set of "input buffer states." Although we allow arbitrary changes of state to be associated with output transitions, the only effect allowed for an input transition $a = (p, v)$ is to append the value v to the end of the input buffer for port p. It is easy to see that such a model satisfies the receptivity and commutativity conditions. The monotonicity condition can be satisfied by defining the automaton in a programming language that contains no primitive for testing for the emptiness of an input buffer.

Suppose M is a port automaton, with port set P. Define a *port history* for M to be an element of the set $(V^\infty)^P$. Each computation γ for M determines a corresponding port history H_γ, where for each $p \in P$, if $a_1 a_2 \ldots$ is the subsequence of those non-identity events a in γ with $port(a) = p$, then $H_\gamma(p)$ is the corresponding sequence $value(a_1)value(a_2)\ldots$ of values. The restrictions H_γ^{in} and H_γ^{out} to the sets of input and output ports, respectively, are called the *input history* and *output history* of γ.

2.2 Networks of Automata

In this section, we define "network automata," which are systems of communicating, concurrently executing, component automata. Communication and synchronization between component automata are performed through shared events. That is, if the event signatures of two component automata have a nonempty intersection, then a transition of one component for an event in the intersection must always occur simultaneously with a transition, for the same event, of the other component. No restriction is placed on the number of component automata that may share an event.

Since dataflow networks are not usually modeled using shared events, a few remarks are in order concerning networks of port automata. Communication between components of a network of port automata occurs when an output transition of one component, with a particular port and data value, occurs simultaneously with input transitions, with the same port and data value, for a number of other components. To ensure that an event shared by two component automata is never an output event for both of them, we define below a notion of "compatibility" of a collection of port automata. We allow arbitrary "fanout" in the sense that a single event may be shared by more than two component automata, as long the event is an output event for at most one of them. This is a bit more general than the usual definition of "linking" in the dataflow literature, in which each port of a process may be connected to at most one port of another process. We do not clutter our theory with any notion of "input buffers" or "channel processes." Rather, we think of an input buffer for a port as already incorporated into the state of each component automaton that inputs from that port. The receptivity, commutativity, and monotonicity conditions in the definition of port automata are, in a sense, abstract descriptions of the properties of these buffers.

Formally, suppose $\mathcal{M} = \{M_i : i \in I\}$ is a collection of automata, where $M_i = (A_i, Q_i, q_i^\iota, \rightarrow_i)$. The *composition* of $\mathcal{M}$ is the automaton

$$\prod \mathcal{M} = (\bigcup_{i \in I} A_i, \prod_{i \in I} Q_i, (q_i^\iota : i \in I), \rightarrow),$$

where $\rightarrow$ is the set of all $((q_i : i \in I), a, (r_i : i \in I))$ such that

- For all $i \in I$, if $a \in A_i$, then $(q_i, a, r_i) \in \rightarrow_i$, and if $a \notin A_i$, then $r_i = q_i$.

We call $\prod \mathcal{M}$ a *network automaton*, the collection $\{M_i : i \in I\}$ a *decomposition* of $\prod \mathcal{M}$, and each of the M_i a *component* of $\prod \mathcal{M}$. Associated with a network automaton $\prod \mathcal{M}$ are *projections*

$$\alpha_i : A \cup \{\epsilon\} \rightarrow A_i \cup \{\epsilon\}, \qquad \sigma_i : (\prod_{i \in I} Q_i) \rightarrow Q_i$$

where the σ_i's are the usual projections associated with the cartesian product, and α_i is defined as follows:

$$\alpha_i(a) = \begin{cases} a, & \text{if } a \in A_i, \\ \epsilon, & \text{otherwise.} \end{cases}$$

These projections lift to projections π_i on computations of $\prod \mathcal{M}$ as follows: If $\gamma = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$ is a finite or infinite computation of M, then

$$\pi_i(\gamma) = \sigma_i(q_0) \xrightarrow{\alpha_i(a_1)} \sigma_i(q_1) \xrightarrow{\alpha_i(a_2)} \ldots.$$

Lemma 1 *Suppose $M = \prod\{M_i : i \in I\}$. Then a sequence γ of transitions is a computation of M iff $\pi_i(\gamma)$ is a computation of M_i, for all $i \in I$.*

Proof – Straightfoward. ∎

A collection $\mathcal{M} = \{M_i : i \in I\}$ of port automata, where $M_i = (A_i, Q_i, q_i^\iota, \rightarrow_i)$, is called *compatible* if for all $i, j \in I$ with $i \neq j$ we have $Y_i \cap Y_j = \emptyset$. If $\mathcal{M}$ is compatible, then $\bigcup_{i \in I} A_i$ is a port signature with

$$Y = \bigcup_{i \in I} Y_i, \qquad X = (\bigcup_{i \in I} X_i) \setminus Y.$$

Our definition of compatibility is a strengthened version, suitable for port automata, of the compatibility condition defined by Lynch and Tuttle in [20] for their input/output automata.

Lemma 2 *If M is the composition of a compatible collection $\{M_i : i \in I\}$ of port automata (resp. monotone port automata), then M is a port automaton (resp. monotone port automaton).*

Proof – Straightforward. ∎

Next, we define the notion of a "fair" computation of a network of port automata. Intuitively, a computation is fair if no component automaton ever fails to produce output, if it tries for a sufficiently long, uninterrupted interval. Formally, if M is the composition of a compatible collection $\{M_i : i \in I\}$ of port automata, then a finite computation γ of M is *fair* if no output events are enabled in state $cod(\gamma)$. An infinite computation γ is *fair* if for each $i \in I$, either there exist infinitely many transitions in γ whose actions are in A_i^{out}, or else there exist infinitely many states in γ for which A_i^{out} is not enabled.

Fairness is a "finite delay" property that is essential to operational models of dataflow networks, and appears in some form in all such models. However, we note that many distinct notions of fairness have been defined in the literature [11]. Ours, which might be referred to as "weak process fairness," admits the situation in which a transition or event becomes repeatedly enabled and disabled during a fair computation, but never appears in that computation. Thus, it is possible in a fair computation for an automaton with two output ports to repeatedly choose one port over another for output.

The following result is a kind of "compositionality" result for networks of port automata. Intuitively, it shows that "networks of networks" are no different than "networks." Although we do not require this result for any theorems stated in this paper, it is important as justification of our interpretation of these theorems as statements about the "implementability" of various relations in terms of "primitives."

Lemma 3 *Suppose M is the composition of a compatible collection $\{M_{ij} : i \in I, j \in J_i\}$ of port automata, and for each $i \in I$, let M_i be the composition of the compatible collection $\{M_{ij} : j \in J_i\}$. Then γ is a fair computation of M iff $\pi_i(\gamma)$ is a fair computation of M_i, for all $i \in I$.*

Proof – See [20], where a similar theorem is proved about input/output automata. ∎

If M is a network of port automata, then the *input/output relation* of M is the set of all pairs $(H_\gamma^{\mathrm{in}}, H_\gamma^{\mathrm{out}})$, such that γ is a fair initial computation of M.

Suppose $R \subseteq (V^\infty)^X \times (V^\infty)^Y$. We say that M *strongly implements* R if the event signature A of M has the form $A = (X \times V) \cup (Y' \times V)$, with $Y \subseteq Y'$, and R is the restriction to $(V^\infty)^X \times (V^\infty)^Y$ of the input/output relation of M. We say that M *weakly implements* R if M strongly implements a subset of R.

We conclude this section by defining some relations of interest.

- If $F : (V^\infty)^X \to (V^\infty)^Y$ is a function, then the *graph* of F is the subset of $(V^\infty)^X \times (V^\infty)^Y$ that contains all pairs $(x, F(x))$ with $x \in (V^\infty)^X$.
- **fmerge** (*fair merge*) is the set of all $((x_1, x_2), y) \in (V^\infty)^2 \times (V^\infty)$, such that y is a shuffle of x_1 and x_2.
- **amerge** (*angelic merge*) is the set of all $((x_1, x_2), y) \in (V^\infty)^2 \times (V^\infty)$ such that y is a shuffle of a prefix x_1' of x_1 and a prefix x_2' of x_2, and such that
    1. If x_1 is finite, then $x_2' = x_2$.
    2. If x_2 is finite, then $x_1' = x_1$.
    3. If both x_1 and x_2 are infinite, then either $x_1' = x_1$ or $x_2' = x_2$.
- **imerge** (*infinity-fair merge*) is the set of all $((x_1, x_2), y) \in (V^\infty)^2 \times (V^\infty)$ such that y is a shuffle of a prefix x_1' of x_1 and a prefix x_2' of x_2, and such that
    1. If x_1 is infinite, then $x_2' = x_2$.
    2. If x_2 is infinite, then $x_1' = x_1$.
    3. If one of x_1 or x_2 is finite, then y is finite.
- **uchoice** is the set of all $(x, y) \in (V^\infty)^\emptyset \times (V^\infty)$, such that y is an infinite sequence of natural numbers.
- **poll** is the set of all $(x, y) \in (V^\infty) \times (V^\infty)$, such that y is a shuffle of x with the infinite sequence $\tau\tau\tau\ldots$ of special values τ.

The relation **uchoice** describes the behavior of a process, with no inputs and one output, that repeatedly chooses an arbitrary natural number and outputs it.

The relation **poll** describes the behavior of a single-input, single-output process that repeatedly polls its input for the presents of data. If a data value is available, it is transmitted to the output channel, otherwise the special value τ is transmitted. Such a process provides the capability of branching on the availability of input data. In [22], a denotational semantics is given for networks that execute programs using polling. It is easy to see that with poll one can implement fair merge [18].

3 Residuals

In this section we develop the technical machinery needed for the expressiveness proofs.

A *residual operation* on an automaton M is a partial binary operation $\uparrow$ on the set of transitions of M, such that the following properties hold:

1. For all transitions t, u of M, if $t \uparrow u$ is defined, then $u \uparrow t$ is defined, $dom(t) = dom(u)$, $dom(t \uparrow u) = cod(u)$, and $cod(t \uparrow u) = cod(u \uparrow t)$. Moreover, either $event(t) \neq event(u)$ and $event(t \uparrow u) = event(t)$, or else $event(t) = event(u)$ and $event(t \uparrow u) = \epsilon$.

2. For all transitions $t : q \to r$ of M, $id_q \uparrow t = id_r$, $t \uparrow id_q = t$, and $t \uparrow t = id_r$.

3. For all transitions t, u, and v of M, $(t \uparrow u) \uparrow (v \uparrow u) = (t \uparrow v) \uparrow (u \uparrow v)$, whenever either side is defined.

Property (3) may be visualized by thinking of t, u, and v as emanating from one vertex of a cube, whose remaining edges are filled in by applying $\uparrow$. If $t \uparrow u$ (and hence $u \uparrow t$) is defined, then we say that transitions t and u are *consistent*. If $t \uparrow u$ is not defined for coinitial transitions t, u, then we say that t and u *conflict*.

Intuitively, a residual operation allows us to formalize the idea that certain pairs of transitions "commute," and thus the order in which they occur in a computation is immaterial. More precisely, if t and u are consistent, then we think of the two computations $t(u \uparrow t)$ and $u(t \uparrow u)$ as "commuting," or as two sequential representations of a single concurrent computation.

If M is any automaton, then there is an obvious residual operation $\uparrow$ on M, with respect to which the only consistent pairs of transitions t, u are the trivial ones with either $t = u$ or one of t, u an identity. Formally, given coinitial transitions $t : q \xrightarrow{a} r$ and $u : q \xrightarrow{b} s$, let $t \uparrow u$ and $u \uparrow t$ be defined exactly when one of the following clauses holds:

1. $t = u$, $t \uparrow u = id_s$, and $u \uparrow t = id_r$.

2. t is an identity transition and u is not, $t \uparrow u = id_s$ and $u \uparrow t = u$.

If M is a port automaton, then its additional structure makes it possible to obtain a residual operation with a larger domain of definition, by adding the clause:

3. If t and u are input transitions, and $port(a) \neq port(b)$, then $t \uparrow u$ and $u \uparrow t$ are the unique transitions for events a and b, respectively, such that $dom(t \uparrow u) = cod(u)$, $dom(u \uparrow t) = cod(t)$, and $cod(t \uparrow u) = cod(u \uparrow t)$.

For a monotone port automaton, we may add another clause:

4. If t is an input transition and u is an output transition, then $t \uparrow u$ is the unique transition for event a such that $dom(t \uparrow u) = cod(u)$, and $u \uparrow t$ is the unique transition for event b such that $dom(u \uparrow t) = cod(t)$ and $cod(u \uparrow t) = cod(t \uparrow u)$.

Lemma 4 *If M is an arbitrary (resp. port, monotone port) automaton, and $\uparrow$ is defined by clauses (1)-(2) (resp. (1)-(3), (1)-(4)), then $\uparrow$ is a residual operation on M.*

Proof – It is obvious in each case that $\uparrow$ satisfies the first two conditions in the definition of a residual operation. It remains to verify the third condition, that $(t \uparrow u) \uparrow (v \uparrow u) = (t \uparrow v) \uparrow (u \uparrow v)$, whenever either side is defined. Suppose, without loss of generality, that $(t \uparrow u) \uparrow (v \uparrow u)$ is defined. We first note that

- In case $u = v$, the result is obvious.

- In case $t = u$, then $t \uparrow u$, and hence $(t \uparrow u) \uparrow (v \uparrow u)$, is an identity. Since $u \uparrow v$ is defined by hypothesis, so is $t \uparrow v$, and $(t \uparrow v) \uparrow (u \uparrow v) = (u \uparrow v) \uparrow (u \uparrow v)$, which is an identity, proving the result.

- In case $t = v$, then $v \uparrow u = t \uparrow u$, hence $(t \uparrow u) \uparrow (v \uparrow u)$ is an identity. Since $t \uparrow v = t \uparrow t$ is an identity, so is $(t \uparrow v) \uparrow (u \uparrow v)$.

- In case one or more of t, u, v is an identity, the result is trivial by clauses (1) and (2).

For the remainder of the proof, we assume that t, u, v are all distinct, and that none of them is an identity. Let a, b, and c be the respective events. Examination of clauses (3)-(4) shows that $t \uparrow u$ and $(t \uparrow u) \uparrow (v \uparrow u)$ must be transitions for event a, $u \uparrow v$ must be a transition for event b, and $v \uparrow u$ must be a transition for event c. Similarly, if $t \uparrow v$ and $(t \uparrow v) \uparrow (u \uparrow v)$ are defined, then they must both be transitions for event a. Thus, to complete the proof, we need only show that $t \uparrow v$ and $(t \uparrow v) \uparrow (u \uparrow v)$ are defined, and that the codomain of $(t \uparrow v) \uparrow (u \uparrow v)$ equals that of $(t \uparrow u) \uparrow (v \uparrow u)$, for then these two transitions, having the same domain, codomain, and event, must be identical. Furthermore, we may assume that the three events a, b, and c are all for different ports, since it follows from clauses (1)-(4) that the only way that transitions for the same port can be consistent is if they are equal.

We consider the various cases:

- Suppose M is an arbitrary automaton, and $\uparrow$ is defined by clauses (1)-(2). Then we have already eliminated all the possible ways in which $t \uparrow u$ can be defined, so there is nothing more to prove.

- Suppose M is a port automaton, and $\uparrow$ is defined by clauses (1)-(3). The only remaining way that $t \uparrow u$ can be defined is if both are input transitions. Then for $u \uparrow v$ to be defined, it must be that v is also an input transition. Thus, all three transitions are input transitions for different ports, and the result follows immediately by the receptivity and commutativity properties of port automata.

- Next, suppose M is a monotone port automaton, and $\uparrow$ is defined by clauses (1)-(4). There are two additional ways in which $t \uparrow u$ can be defined: either t is an input transition and u is an output transition, or t is an output transition and u is an input transition.

  If t is an input transition and u is an output transition, then for $u \uparrow v$ to be defined, it must be the case that v is also an input transition. But then $t \uparrow v$ is defined by clause (3), and $(t \uparrow v) \uparrow (u \uparrow v)$ is defined by clause (4). The result now follows by the fact that there is exactly one transition from state $\mathit{cod}(u \uparrow v) = \mathit{cod}(v \uparrow u)$ for input event a.

  If t is an output transition and u is an input transition, then for $(t \uparrow u) \uparrow (v \uparrow u)$ to be defined, it must be the case that $v \uparrow u$, and hence v, is an input transition. Then $t \uparrow v$ and $(t \uparrow v) \uparrow (u \uparrow v)$ are defined by clause (3). Now, $u \uparrow t$ and $v \uparrow t$ are defined, and are transitions for events b and c, respectively, hence $(u \uparrow t) \uparrow (v \uparrow t)$ is defined by clause (3). Moreover, $(u \uparrow t) \uparrow (v \uparrow t) = (u \uparrow v) \uparrow (t \uparrow v)$, and $(v \uparrow t) \uparrow (u \uparrow u) = (v \uparrow u) \uparrow (t \uparrow u)$, by the fact that input transitions are uniquely determined by their domain and event. Then the codomain of $(t \uparrow u) \uparrow (v \uparrow u)$ must equal that of $(u \uparrow t) \uparrow (v \uparrow t)$. Similarly, the codomain of $(t \uparrow v) \uparrow (u \uparrow v)$ must equal that of $(v \uparrow t) \uparrow (u \uparrow t)$. But the codomains of $(u \uparrow t) \uparrow (v \uparrow t)$ and $(v \uparrow t) \uparrow (u \uparrow t)$ must be equal, proving the result.

∎

Next, we show that residual operations defined on the elements of a collection $\mathcal{M}$ of automata induce "componentwise" a residual operation on the composition $\prod \mathcal{M}$.

Lemma 5 *Suppose $M = \prod\{M_i : i \in I\}$, and suppose that $\uparrow_i$ is a residual operation on M_i, for each $i \in I$. If t, u are transitions of M, and $\pi_i(t) \uparrow_i \pi_i(u)$ is defined for all $i \in I$, then there is a unique transition $t \uparrow u$ of M such that $\pi_i(t \uparrow u) = \pi_i(t) \uparrow_i \pi_i(u)$ for all $i \in I$. Moreover, $\uparrow$ is a residual operation on M.*

Proof – Suppose t, u are such that $\pi_i(t) \uparrow_i \pi_i(u)$ is defined for all $i \in I$. Let $a = \mathit{event}(t)$, $b = \mathit{event}(u)$, and $c_i = \mathit{event}(\pi_i(t) \uparrow_i \pi_i(u))$ for all $i \in I$. The condition, $\pi_i(t \uparrow u) = \pi_i(t) \uparrow_i \pi_i(u)$ for all $i \in I$, ensures that if $t \uparrow u$ exists, then it must satisfy:

$$\begin{aligned} \mathit{dom}(t \uparrow u) &= (\mathit{dom}(\pi_i(t_i) \uparrow_i \pi_i(u_i)) : i \in I) \\ \mathit{cod}(t \uparrow u) &= (\mathit{cod}(\pi_i(t_i) \uparrow_i \pi_i(u_i)) : i \in I) \end{aligned}$$

Moreover, if $\uparrow$ is to be a residual operation, we must also have

$$\mathit{event}(t \uparrow u) = \begin{cases} a, & \text{if } a \neq b \\ \epsilon, & \text{otherwise.} \end{cases}$$

Clearly, $\sigma_i(\mathit{dom}(t \uparrow u)) = \mathit{dom}(\pi_i(t) \uparrow_i \pi_i(u))$ and $\sigma_i(\mathit{cod}(t \uparrow u)) = \mathit{cod}(\pi_i(t) \uparrow_i \pi_i(u))$ for all $i \in I$. To show $\alpha_i(\mathit{event}(t \uparrow u)) = c_i$ for all $i \in I$, let $i \in I$ be arbitrary. Note that either $c_i = \alpha_i(a)$ or $c_i = \epsilon$, because $c_i = \mathit{event}(\pi_i(t) \uparrow_i \pi_i(u))$. There are two cases:

1. If $a = b$, then $\alpha_i(\mathit{event}(t \uparrow u)) = \epsilon$. But $\alpha_i(a) = \alpha_i(b)$, so $c_i = \epsilon$ as well.

2. If $a \neq b$, then $\alpha_i(\mathit{event}(t \uparrow u)) = \alpha_i(a)$. If $\alpha_i(a) \neq \alpha_i(b)$, then we must have $\alpha_i(a) = c_i$. If $\alpha_i(a) = \alpha_i(b)$, then $a \notin A_i$, so we must have $\alpha_i(a) = \epsilon = c_i$.

Thus, $\pi_i(t \uparrow u) = \pi_i(t) \uparrow_i \pi_i(u)$ for all $i \in I$. Since $\pi_i(t) \uparrow_i \pi_i(u)$ is a transition of M_i for all $i \in I$, it follows that $\pi_i(t \uparrow u)$ is a transition of M.

It is now straightforward to check that $\uparrow$ satisfies the axioms for a residual operation. ∎

Finally, we show how to extend a residual operation $\uparrow$ on M to an operation $\Uparrow$ on the finite computations of M. We do this by double induction on the length of the finite computations involved. To understand this definition, it is helpful to think of computations γ and δ as being the bottom two sides of a diamond-shaped lattice, which we try to fill in so that for each small diamond in the lattice, if t and u are the bottom two sides, then t and u are consistent, and the top two sides are $t \uparrow u$ and $u \uparrow t$. The computations γ and δ are consistent if the lattice can be completely filled in, and if so, then $\gamma \Uparrow \delta$ is the side opposite γ, and $\delta \Uparrow \gamma$ is the side opposite δ.

Formally, if $k \geq 0$ then let id_q^k denote the length-k computation sequence of M that contains only transitions id_q. (If $k = 0$, then id_q^k consists of a single state, and no transitions.) Suppose $\gamma : q \to r$ and $\delta : q \to s$ are coinitial finite computations of M. Then

1. If $\gamma = id_q^0$, then $\gamma \Uparrow \delta = id_s^0$.
2. If $\gamma \neq id_q^0$ and $\delta = id_q^0$, then $\gamma \Uparrow \delta = \gamma$.
3. If $\gamma = t\gamma'$, δ is the single transition u, $t \uparrow u$ is defined, and $\gamma' \Uparrow (t \uparrow u)$ is defined, then
$$\gamma \Uparrow \delta = (t \uparrow u)(\gamma' \Uparrow (u \uparrow t)).$$
4. If $|\gamma| > 0$, $\delta = u\delta'$ with $|\delta'| > 0$, $\gamma \Uparrow u$ is defined, and $(\gamma \Uparrow u) \Uparrow \delta'$ is defined, then
$$\gamma \Uparrow \delta = (\gamma \Uparrow u) \Uparrow \delta'.$$

To state the properties of $\Uparrow$, it is convenient to define one more construction, which we call "completion," on automata. Formally, suppose $M = (A, Q, q^\iota, \to)$ is an automaton. The *completion* of M is the automaton $M^* = (A^+, Q, q^\iota, \Rightarrow)$, where A^+ is the set of all finite, nonempty sequences of elements of A, and $\Rightarrow$ contains all triples (q, ϵ, q), and all triples $(q, a_1 \dots a_n, r)$ such that there exists a finite computation $\gamma : q \to r$ of M, with trace $a_1 \dots a_n$.

Lemma 6 *Suppose M is an automaton, and $\uparrow$ is a residual operation on M. Then $\Uparrow$ is a residual operation on M^*. Moreover, for all transitions γ, δ, ξ of M^*, with γ and δ coinitial and δ and ξ composable:*

1. *$\gamma \Uparrow \delta\xi = (\gamma \Uparrow \delta) \Uparrow \xi$, whenever either side is defined.*
2. *$\delta\xi \Uparrow \gamma = (\delta \Uparrow \gamma)(\xi \Uparrow (\gamma \Uparrow \delta))$, whenever either side is defined.*

Proof – Straightforward inductive arguments. ∎

We are now ready to define an "extension" preorder on the set of computations of an automaton M. Intuitively, if γ and δ are coinitial computations, then δ is an "extension" of γ iff every finite prefix of γ can be transformed into a finite prefix of δ by a series of steps in which either adjacent "concurrent" transitions are "permuted," or "padding" (identity transitions) is inserted or deleted.

Formally, for finite computations γ and δ of M, define $\gamma \precsim \delta$ to hold iff γ and δ are consistent and $\gamma \Uparrow \delta$ is a sequence of identities. Next, extend the relation $\precsim$ to infinite computations by defining $\gamma \precsim \delta$ iff for every finite prefix γ' of γ, there exists a finite prefix δ' of δ, such that $\gamma' \precsim \delta'$.

Lemma 7 *Suppose γ and δ are finite computations. If γ and δ are consistent, then $\gamma(\delta \Uparrow \gamma)$ is a $\precsim$-supremum of $\{\gamma, \delta\}$. Conversely, if $\{\gamma, \delta\}$ is $\precsim$-bounded, then γ and δ are consistent.*

Proof – Since $\gamma \Uparrow (\gamma(\delta \Uparrow \gamma)) = (\gamma \Uparrow \gamma) \Uparrow (\delta \Uparrow \gamma)$ and $\delta \Uparrow (\gamma(\delta \Uparrow \gamma)) = (\delta \Uparrow \gamma) \Uparrow (\delta \Uparrow \gamma)$, both of which are sequences of identities, it is clear that $\gamma(\delta \Uparrow \gamma)$ is a $\precsim$-upper bound of $\{\gamma, \delta\}$. Suppose ξ is any $\precsim$-upper bound of $\{\gamma, \delta\}$. Then $\gamma(\delta \Uparrow \gamma) \Uparrow \xi = (\gamma \Uparrow \xi)((\delta \Uparrow \gamma) \Uparrow (\xi \Uparrow \gamma)) = (\gamma \Uparrow \xi)((\delta \Uparrow \xi) \Uparrow (\gamma \Uparrow \xi))$, which is a sequence of identities, so $\gamma(\delta \Uparrow \gamma) \precsim \xi$.

Conversely, if $\{\gamma, \delta\}$ is $\precsim$-bounded, then there exists a finite ξ such that $\gamma \precsim \xi$ and $\delta \precsim \xi$. Then $\gamma \Uparrow \xi$ and $(\gamma \Uparrow \xi) \Uparrow (\delta \Uparrow \xi)$ are sequences of identities. By Lemma 6, $(\gamma \Uparrow \delta) \Uparrow (\xi \Uparrow \delta)$ is defined, hence $\gamma \Uparrow \delta$ is defined. ∎

Theorem 1 *Suppose M is an automaton, and $\uparrow$ is a residual operation on M. The relation $\precsim$ is a preorder, on the set of all computations of M, which extends the prefix ordering. Moreover, the set of all $\precsim$-equivalence classes of computations, with the induced partial order, is a Scott domain whose finite elements are exactly the equivalence classes of finite computations.*

Proof – For finite computations, the relation $\precsim$ extends the prefix ordering, since if γ is a prefix of δ, then $\delta = \gamma\xi$ for some ξ, hence $\gamma \Uparrow \delta = (\gamma \Uparrow \gamma) \Uparrow \xi$, which is a sequence of identities. Reflexivity holds because if $cod(\gamma) = q$, $\gamma \Uparrow \gamma = id_q^{|\gamma|}$. To show transitivity, suppose $\gamma \precsim \delta$ and $\delta \precsim \xi$. Then $\gamma \Uparrow \delta$ is a sequence of identities, so $(\gamma \Uparrow \delta) \Uparrow (\xi \Uparrow \delta)$ is a sequence of identities. Since $(\gamma \Uparrow \delta) \Uparrow (\xi \Uparrow \delta) = (\gamma \Uparrow \xi) \Uparrow (\delta \Uparrow \xi)$, it follows that $(\gamma \Uparrow \xi) \Uparrow (\delta \Uparrow \xi)$ is a sequence of identities. But $\delta \Uparrow \xi$ is a sequence of identities because $\delta \precsim \xi$, hence $\gamma \Uparrow \xi$ is a sequence of identities.

Next, consider the extension to infinite computations. That $\precsim$ is reflexive and transitive is immediate. The fact that $\precsim$ extends the prefix ordering is also clear, since γ is a prefix of δ iff every finite prefix of γ is also a prefix of δ.

Now, by standard results (see, *e.g.* [12]), the ideal completion $\mathcal{I}$ of the set of finite computations, with respect to the $\precsim$ preorder, is a Scott domain whose finite elements are exactly the principal ideals. We claim that the map h, taking each $\precsim$-equivalence class $[\gamma]$ to $\{\delta : \delta \text{ finite}, \delta \precsim \gamma\}$ (which is clearly an element of $\mathcal{I}$), is an order-isomorphism. Since each equivalence class $[\gamma]$ with γ finite maps to the principal ideal generated by γ, we then have the desired result. Obviously h is well-defined, and satisfies $h([\gamma]) \subseteq h([\delta])$ iff $\gamma \precsim \delta$. Note that h is injective, because if $[\gamma] \neq [\gamma']$ then either γ has a finite prefix δ such that $\delta \notin h([\gamma'])$, or else γ' has a finite prefix δ' such that $\delta' \notin h([\gamma])$. To complete the proof, we must show that h is also surjective; that is, every $\precsim$-ideal Γ of the set of finite computations is $h([\gamma])$ for some computation γ.

Suppose $\Gamma \in \mathcal{I}$. We first observe that Γ is at most countable (because the set of all finite computations is countable), hence has an enumeration (perhaps with repetition) $\delta_0, \delta_1, \ldots$. Next, we inductively construct a sequence $\gamma_0, \gamma_1, \ldots$ of elements of Γ, forming a chain under the prefix ordering, such that $\delta_k \precsim \gamma_{k+1}$ for all $k \geq 0$. For the basis step, let γ_0 be the empty computation, which is in Γ because Γ is an ideal. For the induction step, suppose $\gamma_k \in \Gamma$ has been defined for some $k \geq 0$. Since $\delta_k, \gamma_k \in \Gamma$, and Γ is directed, it follows by Lemma 7 that δ_k and γ_k are consistent. Define $\gamma_{k+1} = \gamma_k(\delta_k \Uparrow \gamma_k)$. Clearly, γ_k is a prefix of γ_{k+1}. Since

γ_{k+1} is a $\precsim$-supremum of $\{\gamma_k, \delta_k\} \subseteq \Gamma$, and since the ideal Γ is closed under suprema of finite subsets, it follows that $\gamma_{k+1} \in \Gamma$. Also, $\delta_k \precsim \gamma_{k+1}$, since $\delta_k \Uparrow \gamma_{k+1} = (\delta_k \Uparrow \gamma_k) \Uparrow (\delta_k \Uparrow \gamma_k)$, which is a sequence of identities.

Let γ be the supremum of the chain $\gamma_0, \gamma_1, \ldots$ with respect to the prefix ordering. We claim that $h([\gamma]) = \Gamma$. Clearly, if $\delta \in \Gamma$, then $\delta = \delta_k$ for some $k \geq 0$, hence $\delta \precsim \gamma_{k+1}$. This shows $\Gamma \subseteq h([\gamma])$. Conversely, if δ is a finite computation with $\delta \precsim \gamma$, then $\delta \precsim \xi$ for some finite prefix ξ of γ. But this means $\delta \precsim \gamma_k$ for some $k \geq 0$, hence $h([\gamma]) \subseteq \Gamma$ because $\gamma_k \in \Gamma$ and Γ is an ideal. ∎

Lemma 8 *The map that takes each computation γ to its port history H_γ is continuous, with respect to the $\precsim$ preorder on computations, and the prefix ordering $\preceq$ on port histories.*

Proof – We first show monotonicity. A straightforward induction shows that if γ and δ are consistent finite computations, hence having supremum $\xi = \delta(\gamma \Uparrow \delta)$, then $H_{\gamma \Uparrow \delta}$ is the unique history such that $H_\xi = H_\delta H_{\gamma \Uparrow \delta}$. A special case of this result is: For finite γ and δ, if $\gamma \precsim \delta$, then $H_\gamma \preceq H_\delta$. The extension to infinite γ and δ is straightforward. Continuity then follows from the fact that the ordering $\preceq$ on port histories is algebraic. ∎

4 Application: Networks of Monotone Port Automata

We now consider the special properties enjoyed by the $\precsim$ preorder for networks of monotone port automata. Throughout this section, let M be the composition of a compatible collection $\{M_i : i \in I\}$ of monotone port automata. For each $i \in I$, let $\uparrow_i$ be the residual operation appropriate for the monotone port automaton M_i, and let $\uparrow$, $\Uparrow$, and $\precsim$ be defined for M as in the previous section.

Theorem 2 *A computation γ of M is fair iff it is $\precsim$-maximal among all computations γ' of M with $H^{\mathrm{in}}(\gamma') = H^{\mathrm{in}}(\gamma)$.*

Proof – Suppose γ is not fair. Then for some $i \in I$ we must have $\gamma = \delta\xi$, where A_i^{out} is enabled in every state of ξ, but no event in A_i^{out} appears in ξ. Let a be an event in A_i^{out} that is enabled in state $dom(\xi)$, and let t be a transition for event a from that state. We shall write t for the transition t as well as the computation consisting of the single transition t. Since ξ contains no events in A_i^{out}, it follows by the monotonicity property of M_i that t and ξ' are consistent whenever ξ' is a finite prefix of ξ. Let γ' be the supremum, with respect to the prefix ordering, of $\{\delta t(\xi' \Uparrow t) : \xi' \text{ finite}, \xi' \preceq \xi\}$. Then we have $\gamma \precsim \gamma'$, but not $\gamma' \precsim \gamma$. But since γ and γ' have the same input history, we have shown that γ is not maximal.

Conversely, suppose that γ is not maximal. Then there exists γ', with the same input history as γ, such that $\gamma \precsim \gamma'$ but not $\gamma' \precsim \gamma$. This means that for some finite prefix δ' of γ', it must be that $\delta' \Uparrow \delta$ is not a sequence of identities for any finite prefix δ of γ. Choose δ just long enough so that $\delta' \Uparrow \delta$ has as few nonidentity transitions as possible. Note that $\delta' \Uparrow \delta$ can contain only output or identity events, since otherwise γ and γ' would not have the same input history. Let ξ be the suffix of γ following δ, so that $\gamma = \delta\xi$. Let t be the first output transition in $\delta' \Uparrow \delta$, and suppose the output event of t is in A_i^{out}. Now, t must be consistent with every finite prefix ξ' of ξ, otherwise γ and γ' would not be consistent. However, the consistency of t with all such ξ' implies that ξ can contain no events in A_i^{out}. But then since for every finite prefix ξ' of ξ the transition $t \Uparrow \xi'$ is enabled in state $cod(\xi')$, it must be that γ is not fair. ∎

Lemma 9 *Every computation γ of M $\underset{\sim}{\sqsubset}$-extends to a fair computation with the same input history.*

Proof – The set of all computations γ' such that $\gamma \underset{\sim}{\sqsubset} \gamma'$ and such that γ and γ' have the same input history is nonempty, and has the property that every directed subset has a supremum. By Zorn's Lemma, it follows that this set has a maximal element δ. By Theorem 2, δ is fair. ∎

Lemma 10 *Suppose γ and δ are coinitial computations for M, such that δ consists entirely of input transitions, and such that the input histories H_γ^{in} and H_δ^{in} are consistent. Then there exists a computation ξ of M, with $H_\xi^{\mathrm{in}} = \sup\{H_\gamma^{\mathrm{in}}, H_\delta^{\mathrm{in}}\}$, such that $\gamma \underset{\sim}{\sqsubset} \xi$ and $\delta \underset{\sim}{\sqsubset} \xi$.*

Proof – A straightforward induction shows that γ' and δ' are consistent, hence by Lemma 7 have a $\underset{\sim}{\sqsubset}$-supremum ξ', whenever γ' is a finite prefix of γ and δ' is a finite prefix of δ. Let Ξ be the set of all such ξ'. Then it is easy to see that Ξ is directed, hence has a supremum ξ by Theorem 1. It follows from Lemma 8 that ξ has the required properties. ∎

Suppose $R \subseteq (V^\infty)^X \times (V^\infty)^Y$. If $\hat{x} \in (V^\infty)^X$, then define $R \ll \hat{x}$ (read R *finitely below* $\hat{x}$) to be the set of all finite (x, y) such that $x \preceq \hat{x}$ and $y \preceq \hat{y}$ for some $(\hat{x}, \hat{y}) \in R$. A chain $(x_0, y_0) \preceq (x_1, y_1) \preceq \ldots$ of elements of $R \ll \hat{x}$ is called *final* if for all $(x, y) \in R \ll \hat{x}$, there exists $k \geq 0$ such that either $(x, y) \preceq (x_k, x_k)$ or else (x, y) and (x_k, y_k) are incomparable under $\preceq$.

We say that relation R is

- *total* if for all $x \in (V^\infty)^X$, there exists $y \in (V^\infty)^Y$ such that $(x, y) \in R$.
- *monotone* if whenever $(x, y) \in R$ and $x \preceq x'$, then there exists y' with $y \preceq y'$ and $(x', y') \in R$.
- *continuous* if for all $x \in (V^\infty)^X$, whenever $(x_0, y_0) \preceq (x_1, y_1) \preceq \ldots$ is a final chain in $R \ll x$, then $\sup\{(x_k, y_k) : k \geq 0\} \in R$.

Theorem 3 *Suppose $R \subseteq (V^\infty)^X \times (V^\infty)^Y$. Then*

1. *If R is the input/output relation of a network automaton with monotone components, then R is total and monotone.*

2. *If R is is total, monotone, and continuous, then R is the input/output relation of a network automaton with monotone components.*

Proof – (1) Suppose R is the input/output relation of a network automaton M, with monotone components. To show R is total, we must show how, given an arbitrary input history $x \in (V^\infty)^X$, to construct a fair computation γ, with $H^{\mathrm{in}}_\gamma = x$. This is done by a simple dovetailing construction, which we omit. To show R monotone, suppose γ is a fair initial computation of M, and suppose $x \in (V^\infty)^X$ is an input history with $H^{\mathrm{in}}_\gamma \preceq x$. Let δ be an initial computation of M that consists only of input transitions, such that $H^{\mathrm{in}}_\delta = x$. By Lemma 10, there exists a computation ξ for M, with $H^{\mathrm{in}}_\xi = x$, such that $\gamma \precsim \xi$ and $\delta \precsim \xi$. By Lemma 9, ξ extends to a fair computation ξ' with the same input history, x. But then $(H^{\mathrm{in}}_{\xi'}, H^{\mathrm{out}}_{\xi'}) \in R$, and since $H^{\mathrm{out}}_\gamma \preceq H^{\mathrm{out}}_{\xi'}$ by Lemma 8, we have the desired result.

(2) Suppose R is total, monotone, and continuous. Construct a collection $\{M_p : p \in Y\}$ as follows:

- Let the input ports of M_p be X, and let M_p have the single output port p.
- Let M_p have internal states $(V^*)^X \times V^*$, with the least element of this set as the initial state. Intuitively, the $(V^*)^X$ component of the state of M_p records the input that has arrived so far, and the V^* component keeps track of the output that M_p has produced so far on port p.
- Let the transition relation of M_p contain:
  1. The identity transitions required by the definition of an automaton.
  2. All input transitions of the form $(x, q) \xrightarrow{a} (x', q)$, where if $a = (p, v)$, then $x'(p) = (x(p))v$ and $x'(p') = x(p')$ for $p' \neq p$.
  3. All output transitions of the form $(x, q) \xrightarrow{b} (x, qv)$, where if $b = (p, v)$, then $qv \preceq y(p)$ for some $y \in R(x)$.

Clearly, $\{M_p : p \in Y\}$ is compatible. It is also easy to use the monotonicity of R to verify that each M_p is a monotone port automaton.

Let $M = \prod\{M_p : p \in Y\}$. We claim that M has R as its input/output relation. Given $(x, y) \in R$, a straightforward dovetailing construction produces a fair initial computation γ of M, with $H^{\mathrm{in}}_\gamma = x$ and $H^{\mathrm{out}}_\gamma = y$.

Conversely, suppose $\gamma : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots$ is a fair initial computation of M. A simple induction shows that the $(V^*)^X$ components of the states of each of the M_p's are identical for each state q_k in γ. Let x_k be the common value of these components in state q_k. For each $k \geq 0$, let $y_k \in (V^*)^Y$ be defined so that for each $p \in Y$, $y_k(p)$ is the V^* component of the state of M_p in q_k. It is easy to see that this sequence $(x_0, y_0), (x_1, y_1), \dots$ is a $\preceq$-chain in $R \ll H^{\mathrm{in}}_\gamma$, with $(H^{\mathrm{in}}_\gamma, H^{\mathrm{out}}_\gamma)$ as its supremum. We claim that it is in fact a final chain in $R \ll H^{\mathrm{in}}_\gamma$, thus $(H^{\mathrm{in}}_\gamma, H^{\mathrm{out}}_\gamma) \in R$ follows by the continuity of R. If it were not final, then there would be some $(x, y) \in R \ll H^{\mathrm{in}}_\gamma$ and some $p \in Y$ such that $y' = \sup\{y_k(p) : k \geq 0\}$ is a proper prefix of $y(p)$. Thus, there would exist $K \geq 0$ such that $y_k(p) = y'$ for all $k \geq K$. Since this would mean that an output transition of M_p is enabled in state q_k, for all $k \geq K$, but that no output transitions for M_p appear in γ for $k \geq K$, we would have a contradiction with the assumed fairness of γ. ∎

Corollary 4 *The following relations are strongly implemented by network automata with monotone components:*

1. *The graph of any continuous function on port histories.*
2. *The relations* **amerge** *and* **uchoice***.*
3. *The relation* **imerge***.*

Proof – The relations in (1) are total, monotone, and continuous, as are the relations **amerge** and **uchoice**. The relation **imerge** is total and monotone, but not continuous. However, the relation **imerge** can be strongly implemented by a network that consists of a **uchoice** component and a component with functional behavior, which obeys the following algorithm: Use **uchoice** to select an arbitrary natural number n, then read $n+1$ values from the first input and transfer them to the output. If data is not available, then wait for it. Once $n+1$ values have been read and transferred, choose a new number n', read $n'+1$ values from the second input and transfer them to the output. Repeat this procedure forever. ∎

Corollary 5 *The following relations are not weakly implemented by any network automaton with monotone components.*

1. *The relation* **poll***.*
2. *The relation* **fmerge***.*

Proof – These relations have no total, monotone subsets.

For example, if **poll** had a total, monotone subset P, then P must contain (x, y), where x is the empty history and y is the history that consists of an infinite sequence of τ's. If x' is any nonempty history in which a value other than τ appears, then $x \preceq x'$, hence by monotonicity there would exist y', with $y \prec y'$ and $(x', y') \in P$. But this is impossible, since y is maximal with respect to $\preceq$. A similar argument works for **fmerge**. ∎

5 Discussion

We have shown that fair merge is strictly more powerful than the angelic merge and infinity-fair merge primitives. We accomplished this by identifying a class of networks capable of implementing the weaker primitives, but not fair merge. Although it is not really a surprise to find that fair merge is strictly more powerful than angelic merge, it is somewhat surprising to find that fair merge is strictly more powerful than infinity-fair merge. This is because the finding contradicts a dogma that holds fairness, countable indeterminacy, and the failure of continuity to be somehow equivalent. Notice in particular, that both **uchoice** and **imerge** exhibit countable indeterminacy, but **uchoice** is continuous and **imerge** is not, and both are strictly weaker than fair merge, even in combination with **amerge**.

The notion of a residual operation, which points out commutativity properties of transitions in an automaton, served as our main tool. Close examination of the proofs in Sections 4 and 5 will reveal that all the important properties used can be expressed abstractly as properties of a residual operation on an automaton, and that our assumptions about the concrete structure of the various kinds of automata can be replaced by axioms about a residual operation on an automaton. In fact, such an approach was taken in [25]. There, a "concurrent transition system" was defined to be an automaton plus a residual operation (called there a "translation operation"), and a class of automata, corresponding closely to the monotone port automata defined here, was defined axiomatically. What is missing from [25], though, is a concrete demonstration of the coincidence of fairness and maximality for networks of such automata. This defect is remedied in the present paper.

Our proof of Theorem 2 depends heavily on the properties of monotone automata. Since we have shown that there is no way to perform fair merging if one has only monotone automata, it would at first appear that the pleasant theory developed in this paper, in particular the coincidence of fair and $\precsim$-maximal computations, does not extend to networks capable of fair merge. However, we can define a port automaton $M_{\mathbf{poll}}$ that strongly implements the relation **poll**, by a construction similar to that used in part (3) of the proof of Theorem 3, except that we do not include transitions that output τ from any states in which a non-τ output is possible. Although the automaton $M_{\mathbf{poll}}$ is not monotone, since the arrival of input conflicts with the outputting of τ, it appears that Theorem 2 extends to networks of monotone processes and $M_{\mathbf{poll}}$, when the latter is equipped with the residual operation appropriate for non-monotone automata. Since fair merge can be implemented easily once **poll** is available [18], it would appear that the power of fair merge can be obtained, while retaining many of the pleasant theoretical properties of monotone networks. We are currently investigating the consequences of these observations.

We have not considered dynamic or recursively defined networks in this paper. However, our proofs do apply to networks that contain a countably infinite number of processes. This makes it seem likely that similar proofs could be given once a formalization of dynamic networks as the infinite limits of "finite unwindings" is carried out. Since angelic merge is essentially an iterated version of McCarthy's **amb**, such results would also imply the impossibility of performing fair merging in recursive programs with determinate primitives and **amb**.

References

[1] S. Abramsky. On expressing fair merge with amb. July 1984. Private communication.

[2] K. R. Apt and G. D. Plotkin. A cook's tour of countable nondeterminism. In S. Even

and O. Kariv, editors, *Proceedings of ICALP 81*, pages 479–494, Springer-Verlag, 1981. LNCS 115.

[3] K. R. Apt and G. D. Plotkin. Countable non-determinism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.

[4] R. J. Back. A continuous semantics for unbounded nondeterminism. *Theoret. Comput. Sci.*, 23(2):187–210, 1983.

[5] R. J. Back. Semantics of unbounded nondeterminism. In *Proceedings of 7th Colloquium on Automata, Languages and Programming*, pages 51–63, Springer-Verlag, 1980. LNCS 85.

[6] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *Journal of the ACM*, 26(1):148–175, January 1979.

[7] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 169–236, Cambridge University Press. 1985.

[8] S. D. Brookes. On the relationship of ccs and csp. In *Proceedings of ICALP 83*, Springer Verlag. *Lecture Notes in Computer Science*, 1983.

[9] S. D. Brookes and W. C. Rounds. Behavioral equivalence relations induced by programming logics. In *Proceedings of ICALP 83*, 1983.

[10] W. Clinger and C. Halpern. Alternative semantics for McCarthy's amb. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 467–478, Springer-Verlag, 1985. LNCS 197.

[11] Nissim Francez. *Fairness.* Springer-Verlag, 1986.

[12] I. Guessarian. *Algebraic Semantics.* Volume 99 of *Lecture Notes in Computer Science*, Springer Verlag, 1981.

[13] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, 1978.

[14] G. Huet. Formal structures for computation and deduction (first edition). May 1986. Unpublished manuscript. INRIA, France.

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

[16] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, North-Holland, 1974.

[17] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, North-Holland, 1977.

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

[19] J.-J. Lévy. *Réductions Correctes et Optimales dans le Lambda Calcul.* PhD thesis, Université Paris VII, 1978.

[20] 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.

[21] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer Verlag, 1980.

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

[23] 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.

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

[25] E. W. Stark. Concurrent transition system semantics of process networks. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 199–210, January 1987.