On the Relations Computable by a Class of Concurrent Automata

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

Abstract

We consider *monotone input/output automata*, which model a usefully large class of dataflow networks of indeterminate (or nonfunctional) processes. We obtain a characterization of the relations computable by these automata, which states that a relation $R : X \to 2^Y$ (viewed as a "nondeterministic function") is the input/output relation of an automaton iff there exists a certain kind of Scott domain D, a continuous function $F : X \to [D \to Y]$ and a continuous function $G : X \to \mathcal{P}(D)$, such that $R(x) = F(x)^\dagger(G(x))$ for all inputs $x \in X$. Here $\mathcal{P}$ denotes a certain powerdomain operator, and $\dagger$ denotes the pointwise extension to the powerdomain of a function on the underlying domain. An attractive feature of this result is that it specializes to two subclasses of automata, *determinate* automata, for which G is single-valued, and *semi-determinate* automata, for which G is a constant function. A corollary of the latter result is the impossibility of implementing "angelic merge" by a network of determinate processes and "infinity-fair merge" processes.

1 Introduction

Dataflow networks (see, *e.g.* [3, 4, 6, 7, 8]) consist of a collection of concurrently executing sequential processes that communicate by transmitting sequences or "streams" of "value tokens" over FIFO communication channels. Typically, a network is described as a directed graph, whose nodes are processes and whose arcs are communication channels. Each channel serves to connect an "output port" of one process to an "input port" of another process. "Determinate" (or functional) networks were first studied by Kahn [7], who gave an elegant fixed-point principle for determining the function computed by a network from the functions computed by the components. "Indeterminate" (or non-functional) networks remain less well understood, despite extensive study. An interesting class of indeterminate processes are the "merge" processes, which shuffle together sequences of values from two input channels onto a single output channel. *Fair merge* (**fmerge**) guarantees that every value arriving on either of the two inputs will eventually be transmitted to the output. *Angelic merge* (**amerge**) guarantees to transmit all values from one input channel only in case the sequence of values arriving on the *other* channel is finite. *Infinity-fair merge* (**imerge**) guarantees to transmit all values from one input channel only in case the sequence of values arriving on the other channel is *infinite*.

In previous papers [15, 21, 22], we identified the networks of "monotone" processes as an interesting and particularly well-behaved subclass of the class of indeterminate dataflow networks. The main result of [15] was that, although such networks can be used to perform the "unfair" merging operations **amerge** and **imerge**, no such network can implement **fmerge**. We obtained this result by showing that every network of monotone processes implements a monotone input/output relation, that **amerge** and **imerge** are implementable by networks of monotone processes, but that **fmerge** is not a monotone relation. Now, it is not difficult to see how to use **fmerge** to implement **imerge**, and Panangaden [13] has shown how, using **fmerge**, we can implement **amerge**. Thus the questions left unanswered in [15] are: (1) Can **amerge** be implemented by a network of **imerge** processes and processes with functional behaviors? (2) Can **imerge** be so implemented by **amerge**? Question (1) was answered in the negative by Panangaden and Shanbhogue [14], using *ad hoc* techniques. In this paper, we give an alternative proof by the more general methods of [15]. Contrary to our initial expectations, we have found that question (2) has an affirmative answer, and we exhibit a network with the stated property. This network can be viewed as a demonstration that there can be no notion of "finite indeterminacy" that is preserved by network construction. The three merge primitives, arranged in

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

0

order of their power to implement relations when used in combination with functional processes, thus form a strict hierarchy with **imerge** the weakest of the three primitives, **amerge** strictly stronger, and **fmerge** the strongest.

To obtain the negative answer to question (1), we refine the methods of [15], thereby obtaining a useful characterization of the relations implemented by the class of "monotone input/output automata." Our characterization theorem states that a relation $R : X \to 2^Y$ (viewed as a "nondeterministic function") is the input/output relation of such an automaton iff there exists a certain kind of Scott domain D, a continuous function $F : X \to [D \to Y]$ and a continuous function $G : X \to \mathcal{P}(D)$, such that $R(x) = F(x)^{\dagger}(G(x))$ for all inputs $x \in X$. Here $\mathcal{P}$ denotes a certain powerdomain operator, and $\dagger$ denotes the pointwise extension to the powerdomain of a function on the underlying domain. This characterization also specializes nicely to the classes of "determinate" and "semi-determinate" automata. For determinate automata, the theorem holds with the restriction that G be single-valued, and for semi-determinate automata, G is restricted to be a constant function. The latter result is interesting, since it shows that semi-determinate automata may be thought of as implementing a collection of continuous functions, indexed by an "oracle set" (the constant value of G). Since **imerge** is representable as such an oracleized set of functions, but **amerge** is not, we conclude that **imerge** is implementable by a semi-determinate automaton, but **amerge** is not. Since the class of semi-determinate automata is closed under network construction, it follows that **amerge** cannot be implemented by a network of **imerge** and functional components.

As in [15], our main tool is the notion of *permutation equivalence* of computation sequences. Intuitively, two finite computation sequences are permutation-equivalent if one can be transformed into the other by a finite sequence of steps in which the order of adjacent "concurrent pairs" of transitions is permuted. We use a result, shown elsewhere [20, 23], that the set of equivalence classes of computation sequences of an automaton, under the induced partial order, is a Scott domain whose finite elements are exactly the equivalence classes of finite computation sequences, and which has other useful properties. This result was established in the previous work by using the auxiliary notion of the *residual* of one finite computation sequence by another, to obtain an alternative characterization of permutation equivalence. For this paper, we simply quote the previous result without proof, and this permits us to prove our main result without having to develop the algebra of residuals.

2 Preliminaries

We require some concepts from trace theory [1, 12] and domain theory [16, 17, 18]. In the sequel, all sets whose cardinality is left unspecified are assumed to be at most countable.

2.1 Domains

A (Scott) *domain* is an ω-algebraic, consistently complete cpo. A *subdomain* of D is a subset U of D, which is a domain under the the ordering inherited from D, such that the inclusion of U in D is continuous. A subdomain U of D is *normal* if for every $d \in D$, the set $\{u \in U : u \sqsubseteq d\}$ is directed. Normal subdomains have the following property, which we use in the proof of Lemma 7.

Lemma 1 *Suppose U is a normal subdomain of a domain D. Then any continuous $F : U \to E$ extends to a least continuous $F' : D \to E$.*

Proof – Take $F'(d) = \bigsqcup\{F(u) : u \in U, u \sqsubseteq d\}$. ∎

An *interval* in a domain D is a pair $[d, d']$ with $d \sqsubseteq d'$. An interval $[d, d']$ is *prime* if d' *covers* d; that is, there does not exist d'' with $d \sqsubset d'' \sqsubset d'$.

2.2 The Fringe Set Powerdomain

Suppose D is a domain. Let $I_D : D \to 2^D$ denote the map that takes d to $\{d\}$. If $F : D \to E$ is any function, then its *pointwise extension* is the function $F^{\dagger} : 2^D \to 2^E$ defined by: $F^{\dagger}(U) = \{F(d) : d \in U\}$. Call a subset U of D *closed* if it is downward-closed, closed under lubs of directed subsets, and also closed under lubs of pairs of elements that are consistent in D. The *closure* U^c of U is the least closed set containing U. Define a nonempty subset U of D to be a *fringe set* if U is precisely the set of all maximal elements of U^c. It follows that fringe sets are *pairwise inconsistent*: if d, d' are two distinct elements of a fringe set U, then the set $\{d, d'\}$ has no upper bound in D. Define a binary relation $\sqsubseteq$ on fringe sets of D by defining $U \sqsubseteq V$ iff $U^c \subseteq V^c$. An equivalent characterization of $\sqsubseteq$ is the following: $U \sqsubseteq V$ iff $\forall x \in U\ \exists y \in V (x \sqsubseteq y)$. It is clear from this definition that $\sqsubseteq$ is a preorder, and since U and V are fringe sets, the relation $\sqsubseteq$ is a partial order. In fact, in view of the bijective correspondence between fringe sets and their closures, we have the following:

Lemma 2 *The set $\mathcal{P}(D)$ of all fringe sets of D, equipped with the ordering $\sqsubseteq$, is a domain.*

Indeed, this is nothing more than a slight variant of the "relational," "lower," or "Hoare" powerdomain of D [16, 17], whose elements we have chosen to represent by fringe sets, rather than by closed sets as is perhaps more usual. The fringe set representation proves more convenient for our purposes. Usually, the relational powerdomain of a domain D is defined in terms of sets that are closed in the Scott topology on D. The set of maximal elements of a Scott-closed set is pairwise incomparable under the ordering on D, but it is not necessarily pairwise inconsistent. The closed sets defined above are Scott-closed sets with the additional property of being closed under lubs of consistent pairs.

2.3 Traces

A *concurrent alphabet* is a set X, equipped with a symmetric, irreflexive binary relation $||$ on X, called the *concurrency relation.* The *direct product* of concurrent alphabets X and Y is the concurrent alphabet $X \otimes Y$ whose set of elements is the disjoint union of $X + Y$ of X and Y, and whose concurrency relation $||_{X\otimes Y}$ is defined to be

$$||_{X\otimes Y} \;=\; ||_X \,\cup\, ||_Y \,\cup (X \times Y) \cup (Y \times X).$$

Suppose X is a concurrent alphabet. Let X^* denote the free monoid generated by X, then there is a least congruence $\sim$ on X^* such that $a||b$ implies $ab \sim ba$ for all $a, b \in X$. The quotient $X^* /\!\sim$ is the *free partially commutative monoid* generated by X, and its elements are called *traces.* We use ϵ to denote the identity element (the empty trace). The monoid $X^* /\!\sim$ is partially ordered, with $x \sqsubseteq y$ iff $\exists z(xz = y)$. Let $\bar{X}$ denote the domain obtained by ideal completion of this poset. We call $\bar{X}$ the *trace domain* generated by the concurrent alphabet X. Notice that since the finite (=isolated=compact) elements of $\bar{X}$ are in bijective correspondence with the elements of $X^* /\!\sim$, they inherit the monoid operation of $X^* /\!\sim$, with the bottom element of $\bar{X}$ as the monoid identity. Let $\bar{X}^o$ denote the set of finite elements of $\bar{X}$. In the sequel, we identify the elements of $X^* /\!\sim$ with the corresponding elements of $\bar{X}^o$. If $Z \subseteq X$, then the monoid homomorphism from X^* to Z^* that deletes elements of $X \setminus Z$, respects $\sim$, hence induces a monoid homomorphism from $X^* /\!\sim$ to $Z^* /\!\sim$ and a continuous map from $\bar{X}$ to $\bar{Z}$. We write $x|Z$ for the application of this map to a trace $x \in \bar{X}$.

Lemma 3 *The following are equivalent statements about a domain D:*

1. *D is isomorphic, by a map that preserves prime intervals, to a normal subdomain of $\bar{X}$ for some concurrent alphabet X.*
2. *E is an* event domain *in the sense of [5, 24].*
3. *D is isomorphic to the domain of configurations of an event structure $(E, \vdash, \#)$, where E is a set of* events, *$\vdash$ is an* enabling relation *between finite subsets of E and elements of E, and $\#$ is a binary* conflict relation *on E.*

The equivalence of (2) and (3) is a theorem of Winskel [5, 24]. The equivalence of (1) and (2) is shown in [20]. We shall see that event domains arise naturally as the domains of computations of the kind of automata we consider.

3 Monotone Input/Output Automata

A *monotone input/output automaton* (henceforth simply "automaton") is a tuple

$$A = (E, X, Y, Q, q^{\mathrm{I}}, T),$$

where

- E is a concurrent alphabet of *actions*, and X, Y are disjoint subsets of E, called the sets of *input actions* and *output actions*, respectively. The elements of $E \setminus (X \cup Y)$ are called *internal actions.*
- Q is a set of *states.*
- $q^{\mathrm{I}} \in Q$ is a distinguished *initial state.*
- T is a function that maps each pair of states $q, r \in Q$ to a set $T(q, r) \subseteq E$.

These data are required to satisfy the following conditions:

(Disambiguation) $r \neq r'$ implies $T(q, r) \cap T(q, r') = \emptyset$.

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

(Monotonicity) $a||b$ whenever $a \in X$ and $b \in E \setminus X$.

(Receptivity) For all states q and input actions a, there exists a state r such that $a \in T(q, r)$.

Intuitively, if $a \in E$, then $a \in T(q, r)$ iff it is possible for A to take a step from state q in which action a occurs and the state changes to r. Input actions represent steps in which input is received from the external environment, and output actions represent steps in which output is transmitted to the external environment. We say that action a is *enabled* in state q if $a \in T(q, r)$ for some r. By the disambiguation condition, if a is enabled in state q, then there is a unique state r such that $a \in T(q, r)$. We sometimes denote this state by qa.

The *transitions* of an automaton are the triples (q, a, r) with $a \in T(q, r)$. We often use the notation $q \xrightarrow{a} r$ or $a : q \rightarrow r$ to denote the transition (q, a, r). If t is the transition (q, a, r), then q is called the *domain* $dom(t)$ of t and r is called the *codomain* $cod(t)$ of t. Transitions t and u are called *coinitial* if $dom(t) = dom(u)$.

Monotone input/output automata are closely related to the input/output automata defined by Lynch and Tuttle [10, 11]. If we ignore the distinction between input and output actions, and we delete the monotonicity and receptivity conditions, we obtain automata similar to those that have been studied by Bednarczyk [2], Kwiatkowska [9], and Shields [19]. To further motivate the definitions, we describe how monotone input/output automata can be used to model dataflow networks. For this, we use a restricted class of automata called "monotone port automata," which are similar to the monotone port automata defined in [15].

Formally, let us fix in advance countably infinite sets P of *ports* and V of *values*. We call elements of the set $P \times V$ *port actions*, and if e is the port action (p, v), then we write $port(e)$ for its port component p, and $value(e)$ for its value component v. Define an automaton $A = (E, X, Y, Q, q^{\mathrm{I}}, T)$ to be a *port automaton* if $X = P^{\mathrm{in}} \times V$ and $Y = P^{\mathrm{out}} \times V$ for some disjoint finite subsets P^{in} and P^{out} of P, and for all $e, e' \in X \cup Y$ we have $e || e'$ iff $port(e) \neq port(e')$. Such an automaton models a dataflow process or network with input port set P^{in} and output port set P^{out}. An input (output) action (p, v) corresponds to the receipt (transmission) of value v on port p, and internal actions correspond to internal computation steps in which communication with the environment does not take place. It follows from the definition of $||$ that the trace domains $\bar{X}$ and $\bar{Y}$ are, up to isomorphism, domains of "port histories" as in [15]. That is, the elements of $\bar{X}$ and $\bar{Y}$ are functions, mapping ports to finite and infinite sequences of values, and ordered argumentwise by prefix.

Notice that, in contrast to more typical models for dataflow networks, our definitions do not specify directly any particular concrete structure for the state sets of port automata, such as the existence of a FIFO "channel buffers" for each input or output port. However, the receptivity, commutativity, and monotonicity axioms can be interpreted as abstract statements about the properties of such buffers. The receptivity condition can be interpreted as stating that arriving input values can always be placed into an input buffer. The commutativity condition, together with the definition of the relation $||$, captures the notion that distinct ports have separate buffers, The monotonicity condition can be viewed as a restriction on the way input buffers can be accessed: since arriving input cannot disable transitions previously enabled, one can test for the *presence* of input in a buffer, but never for its *absence*.

4 Computations of Automata

A *finite computation sequence* for an automaton is a finite sequence γ of transitions of the form:

$$q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} q_n.$$

The number n is called the *length* $|\gamma|$ of γ. We call the computation sequence of length 0 from state q the *identity* computation sequence, and we denote it by id_q, or just id, when q is clear from the context. Also, it will sometimes be convenient to write $id_q = (q \xrightarrow{\epsilon} q)$. An *infinite computation sequence* is an infinite sequence of transitions:

$$q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \dots.$$

A computation sequence that contains only input actions is called a *pure-input* computation sequence; one containing no input actions is called a *non-input* computation sequence. The *trace* of γ is the element $\mathrm{tr}(\gamma) = e_1 e_2 \dots$ of the trace domain $\bar{E}$.

We extend notation and terminology for transitions to computation sequences, so that if γ is a computation sequence, 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 $\gamma : q \to r$ to assert that γ is a finite computation sequence with domain q and codomain r. A computation sequence γ is *initial* if $dom(\gamma)$ is the distinguished initial state I. If $\gamma : q \to r$ and $\delta : q' \to r'$ are finite computation sequences, then γ and δ are called *composable* if $q' = r$, and we then define their *composition* to be the finite computation sequence $\gamma\delta : q \to r'$, obtained by concatenating γ and δ and identifying $cod(\gamma)$ with $dom(\delta)$. The operation of composition of finite computation sequences is associative, and identity computation sequences behave as units for it. A finite computation sequence γ is a *prefix* of a computation sequence δ, and we write $\gamma \preceq \delta$, iff there exists a computation sequence ξ with $\gamma\xi = \delta$.

Define *permutation equivalence* to be the least congruence $\sim$, respecting concatenation, on the set of finite computation sequences of A such that:

- Computation sequences $q \xrightarrow{a} r \xrightarrow{b} p$ and $q \xrightarrow{b} s \xrightarrow{a} p$ are $\sim$-related if $a||b$.

Closely related to permutation equivalence is the *permutation preorder* relation $\precsim$ on finite computation sequences of A, which is defined to be the transitive closure of $\preceq \cup \sim$. That is, $\gamma \precsim \delta$ iff there is a way to perform a finite number of permutations of adjacent concurrent transitions in δ, and obtain a computation δ' that has γ as a prefix. It is not difficult to see that $\gamma \sim \delta$ iff $\gamma \precsim \delta$ and $\delta \precsim \gamma$. Permutation preorder extends in a straightforward way to infinite computation sequences as well: if γ' and δ' are coinitial finite or infinite computation sequences, then define $\gamma' \precsim \delta'$ to hold iff for every finite $\gamma \preceq \gamma'$ there exists a finite $\delta \preceq \delta'$, such that $\gamma \precsim \delta$. We may then extend permutation equivalence to infinite computation sequences by defining $\gamma' \sim \delta'$ iff $\gamma' \precsim \delta'$ and $\delta' \precsim \gamma'$.

A *computation* is a $\sim$-equivalence class of computation sequences. Obviously, all finite computation sequences that are representatives of the same $\sim$-equivalence class have the same length, so the notion of the *length* $|\gamma|$ of a finite computation γ makes sense. For each state q, the permutation preorder $\precsim$ on computation sequences from state q induces a partial order $\sqsubseteq$ on the set of all computations from state q. Coinitial computations γ and γ' are called *consistent* if they have an upper bound with respect to $\sqsubseteq$. For finite γ, γ', this is equivalent to the existence of a pair of finite computations δ, δ' such that $\gamma\delta' = \gamma'\delta$.

The following result, proved in [20], provides a great deal of information about the structure of the partially ordered set of computations of an automaton. Our main theorem is a direct consequence.

Lemma 4 *Suppose A is an automaton. For each state q, the set $\mathrm{Comp}_q(A)$ of computations of A with domain q, partially ordered by $\sqsubseteq$, is an event domain whose finite elements are exactly the equivalence classes of finite computation sequences. Moreover, the map $\mathrm{tr} : \mathrm{Comp}_q(A) \to \bar{E}$ that takes each equivalence class to the corresponding trace, is a prime-interval-preserving embedding of $\mathrm{Comp}_q(A)$ as a normal subdomain of $\bar{E}$.*

Rather than reproving this result here, we merely comment on the method used. It is fairly obvious from the definitions we have given that the map $\mathrm{tr} : \mathrm{Comp}_q(A) \to \bar{E}$ is a monotone injection that preserves prime intervals. To show $\mathrm{Comp}_q(A)$ is a domain, it is necessary to show that all directed subsets have lubs and that all consistent pairs of computations have lubs. To show that the map tr is an embedding of a normal subdomain, we may show that tr is strict, additive, continuous, and reflects consistency. To show these facts, we define the auxiliary notion of the *residual* of one computation sequence "after" another. Formally, we have the following: Suppose γ and γ' are consistent finite computations. Then there exists a unique pair of finite computations $\gamma\backslash\gamma'$ and $\gamma'\backslash\gamma$ (read "γ *after* γ'" and "γ' *after* γ"), such that $\gamma(\gamma'\backslash\gamma) = \gamma'(\gamma\backslash\gamma')$, and such that if δ and δ' are any finite computations with $\gamma\delta' = \gamma'\delta$, then there exists a unique finite computation ξ such that $(\gamma\backslash\gamma')\xi = \delta$ and $(\gamma'\backslash\gamma)\xi = \delta'$.

We may think of $\backslash$ as a partial binary operation on coinitial pairs of computations, where $\gamma\backslash\delta$ is defined exactly when γ and δ are consistent, or when they could both be part of the "same concurrent computation." In this case, δ may contain some transitions that "overlap" with γ and some that are "concurrent" with γ. Then $\gamma\backslash\delta$ should be thought of as what is "left" of γ after the part that overlaps with δ has been deleted. The residual $\gamma\backslash\delta$ is undefined exactly when γ contains some indeterminate choice that conflicts with a choice made in δ. A concrete construction of $\gamma\backslash\delta$ for arbitrary consistent finite computations γ and δ is performed by induction on their length.

A similar residual operation can be defined on finite traces (in fact more easily so, since there are no states to worry about). Formally, if x and x' are consistent traces then the *residual* of x after x' is the unique trace $x\backslash x'$ with the property that $x'(x\backslash x') = x \sqcup x'$. From this definition, one can easily see that $x \sqsubseteq y$ iff $x\backslash y = \epsilon$. A similar relation holds for computations: $\gamma \sqsubseteq \delta$ iff $\gamma\backslash\delta = id$. Thus, the partial ordering on computations has an equivalent characterization in terms of residuals. Using this observation, Lemma 4 may then be proved by using residuals to perform an inductive construction of lubs of directed collections of computations and of consistent pairs, and also to show that the map $\mathrm{tr} : \mathrm{Comp}_q(A) \to \bar{E}$ preserves residuals (hence is continuous) and reflects consistency.

We conclude this section with some additional facts about the computations of monotone input/output automata. We do not use these facts explicitly in this paper, but they are helpful in understanding the behavior of these automata. Proofs can be found (in a somewhat more abstract setting) in [21, 22].

Proposition 5 *A monotone automaton A has the following properties:*

1. *For every state q and input trace x, there exists a unique computation $\iota_x(q)$ with $\mathrm{dom}(\iota_x(q)) = q$ and $\mathrm{tr}(\iota_x) = x$. Moreover, the map $\iota : \bar{X} \to \mathrm{Comp}_q(A)$ that takes $x \in \bar{X}$ to $\iota_x \in \mathrm{Comp}_q(A)$, is a prime-interval-preserving embedding of $\bar{X}$ as a normal subdomain of $\mathrm{Comp}_q(A)$.*

2. *Suppose γ is a computation with $\mathrm{dom}(\gamma) = q$, and x is an input trace that is consistent with $\mathrm{tr}(\gamma)|X$. Then γ and $\iota_x(q)$ are consistent.*

3. *Every finite computation γ factors uniquely as $\gamma = \delta\xi$, where δ is a pure-input computation and ξ is a non-input computation. We call the pair (δ, ξ) a* pure-input/non-input factorization *of γ.*

4. *For every computation γ, there exists a computation δ that is $\sqsubseteq$-maximal among all computations γ' such that $\gamma \sqsubseteq \gamma'$ and $\mathrm{tr}(\gamma)|X = \mathrm{tr}(\gamma')|X$.*

5 Relations Computed by Automata

A computation is called *completed* if it is $\sqsubseteq$-maximal among all computations having the same input trace. In [15] it was shown that this notion coincides with an appropriate notion of "fairness" for monotone automata that are constructed as the parallel composition of a collection of "single-process" components. Specifically, a computation is "fair" if every component process having an enabled non-input action eventually performs some non-input action. Here, we consider a more general situation in which automata need not be networks of single-process components. Although fairness does not generalize directly to this situation, the notion "completed" does, and we therefore adopt it as the appropriate generalization of fairness.

The *input/output relation* of an automaton A is the function $R : \bar{X} \to 2^{\bar{Y}}$ that maps each $x \in \bar{X}$ to the set of all $y \in \bar{Y}$, such that for some some completed initial computation γ of A, we have $x = \mathrm{tr}(\gamma)|X$ and $y = \mathrm{tr}(\gamma)|Y$.

We can now state our main result:

Theorem 1 *A map $R : \bar{X} \to 2^{\bar{Y}}$ is the input/output relation of a monotone input/output automaton iff there exists an event domain D, a continuous function $F : \bar{X} \to [D \to \bar{Y}]$, and a continuous function $G : \bar{X} \to \mathcal{P}(D)$, such that $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$.*

Proof – Follows from Lemmas 6 and 7 below. ∎

Before proceeding to the proof, we give the definitions of **fmerge**, **amerge**, and **imerge**, and mention the implications of Theorem 1 for these. Intuitively, each of these relations takes as input finite or infinite sequences of values on two input ports, and produces a finite or infinite sequence of values on one output port. To formalize this, let p_0 and p_1 stand for the two input ports, and let p_2 stand for the single output port. Let V be a countably infinite set of values. Form a concurrent alphabet $X = \{p_0, p_1\} \times V$ with $(p, v)||(p', v')$ iff $p \neq p'$, and a concurrent alphabet $Y = \{p_2\} \times V$ with $|| = \emptyset$. Then the trace domain $\bar{X}$ is isomorphic to the domain of functions mapping $\{p_0, p_1\}$ to finite or infinite sequences of values with the prefix order; similarly, $\bar{Y}$ is isomorphic to the domain of functions from $\{p_2\}$ to value sequences. It will be convenient notationally to regard $x \in \bar{X}$ as a function on $\{p_0, p_1\}$, and similarly $y \in \bar{Y}$ as a function on $\{p_2\}$.

The three relations are defined as follows:

- **fmerge** is the set of all $(x, y) \in \bar{X} \times \bar{Y}$ such that $y(p_2)$ is a shuffle of $x(p_0)$ and $x(p_1)$.
- **amerge** is the set of all $(x, y) \in \bar{X} \times \bar{Y}$ such that $y(p_2)$ is a shuffle of a prefix x_0 of $x(p_0)$ and a prefix x_1 of $x(p_1)$, such that
  1. If $x(p_0)$ is finite, then $x_1 = x(p_1)$.
  2. If $x(p_1)$ is finite, then $x_0 = x(p_0)$.
  3. If both $x(p_0)$ and $x(p_1)$ are infinite, then either $x_0 = x(p_0)$ or $x_1 = x(p_1)$.
- **imerge** is the set of all $(x, y) \in \bar{X} \times \bar{Y}$ such that $y(p_2)$ is a shuffle of a prefix x_0 of $x(p_0)$ and a prefix x_1 of $x(p_1)$, such that
  1. If $x(p_0)$ is infinite then $x_1 = x(p_1)$.
  2. If $x(p_1)$ is infinite, then $x_0 = x(p_0)$.
  3. If one of $x(p_0)$ and $x(p_1)$ is finite, then both x_0 and x_1 are finite, and either $x_0 = x(p_0)$ or else $x_1 = x(p_1)$.

As a consequence of Theorem 1, if R is the input/output relation of an automaton, then R has the following *monotonicity* property: If $y \in R(x)$, and $x \sqsubseteq x'$, then there exists $y' \in R(x')$ with $y \sqsubseteq y'$. The relation **fmerge** is not monotone (consider $x(p_0) = \epsilon$, $x(p_1) = 555\ldots$, $y(p_2) = 555\ldots$, $x'(p_0) = 7$ and $x'(p_1) = 555\ldots$), hence is not the input/output relation of an automaton. In contrast, **amerge** can be expressed in the form of Theorem 1: assuming that the set of values that arrive on port p_0 is disjoint from those that arrive on p_1, we may take $D = \bar{Y}$, take $G =$ **amerge**, and take $F(x)(y) = y$ for all $x \in \bar{X}$ and $y \in \bar{Y}$. (If the value sets are not disjoint, then we may use a slight modification of this construction, in which we take D to be a "tagged" version of $\bar{Y}$, and the function $F(x)$ removes the tags.) Hence **amerge** is the input/output relation of an automaton. It is slightly more complicated to express **imerge** in the required form, and we postpone this to Section 6.2 below. The results of [15] are therefore a corollary of Theorem 1.

Lemma 6 *Suppose A is an automaton with input/output relation $R : \bar{X} \to 2^{\bar{Y}}$. Then there exists an event domain D, a continuous function $F : \bar{X} \to [D \to \bar{Y}]$, and a continuous function $G : \bar{X} \to \mathcal{P}(D)$, such that $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$.*

Proof – Let $D = \mathrm{Comp}_{q^{\mathrm{I}}}(A)$, then D is an event domain by Lemma 4. Define $F : \bar{X} \to [D \to \bar{Y}]$ by:

$$F(x)(\gamma) = \bigsqcup\{\mathrm{tr}(\delta)|Y : \delta \sqsubseteq \gamma, \mathrm{tr}(\delta)|X \sqsubseteq x\}.$$

That is, $F(x)(\gamma)$ is the maximum output that can be produced in a "prefix up to permutation" δ of γ, when the input is constrained to be less than x. It is straightforward to check that F is well-defined and continuous. Define $G : \bar{X} \to 2^D$ to map each $x \in \bar{X}$ to the set of all completed initial computations with input history x. By construction, $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$. It remains to be shown that $G : \bar{X} \to \mathcal{P}(D)$ and that G is continuous.

We first claim that $G(x)$ is a fringe set for all $x \in \bar{X}$. But this is clear from the fact that the set $\Gamma(x)$ of all initial computations γ with $\mathrm{tr}(\gamma)|X \sqsubseteq x$ is closed, and $G(x)$ is the set of its maximal elements. Next, we show that G is monotone. This is immediate from the fact that if $x \sqsubseteq x'$, then $\Gamma(x) \subseteq \Gamma(x')$. Finally, we show that G is continuous. Given a directed collection $\{x_i : i \in I\}$ of inputs, with supremum x, let $\Gamma = \bigcup \Gamma(x_i)$, and note that $\Gamma = \Gamma(x)$. Thus, $G(x) = \bigsqcup\{G(x_i) : i \in I\}$. ∎

Lemma 7 *Let $R : \bar{X} \to 2^{\bar{Y}}$ and suppose there exists an event domain D, a continuous function $F : \bar{X} \to [D \to \bar{Y}]$, and a continuous function $G : \bar{X} \to \mathcal{P}(D)$, such that $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$. Then R is the input/output relation of a monotone automaton.*

Proof – Since D is an event domain, by Lemma 4 it is isomorphic via a prime-interval-preserving map μ to a normal subdomain of $\bar{Z}$, for some concurrent alphabet Z. Define $C(x) = \mu(G(x)^c)$. Let $F' : \bar{X} \to [\bar{Z} \to \bar{Y}]$ be defined so that for each $x \in \bar{X}$, the map $F'(x) : \bar{Z} \to \bar{Y}$ is the least continuous function (which exists by Lemma 1) with the property that $F(x) = F'(x) \circ \mu$.

Next, we construct an automaton

$$A = (E, X, Y, Q, q^{\mathrm{I}}, T)$$

as follows:

- Let $E = X \otimes Y \otimes Z$.
- Let $Q = \bar{X}^{\circ} \times \bar{Y}^{\circ} \times \bar{Z}^{\circ}$, with $q^{\mathrm{I}} = (\epsilon, \epsilon, \epsilon)$.
- The map T is defined as follows:
  1. If $a \in X$, then $a \in T((x, y, z), (x', y', z'))$ iff $x' = xa$, $y' = y$, and $z' = z$.
  2. If $b \in Y$, then $b \in T((x, y, z), (x', y', z'))$ iff $x' = x$, $y' = yb$, $z' = z$, and $y' \sqsubseteq F'(x)(z)$.
  3. If $c \in Z$, then $c \in T((x, y, z), (x', y', z'))$ iff $x' = x$, $y' = y$, $z' = zc \in C(x)$.

Intuitively, the first component of the state will is used to keep track of the finite input received so far in a computation, the second component keeps track of the finite output issued so far, and the third keeps track of the history of how indeterminate choices have been resolved so far.

It is straightforward to check that the definition of A satisfies the requirements for a monotone input/output automaton. The commutativity property of A follows from the fact that $G(x)$, hence $C(x)$, is closed under lubs of consistent pairs of elements. We also observe, by a straightforward induction, that if γ is any finite initial computation sequence for A, then $\mathrm{tr}(\gamma)|Y \sqsubseteq F'(\mathrm{tr}(\gamma)|X)(\mathrm{tr}(\gamma)|Z))$ and $\mathrm{tr}(\gamma)|Z \in C(\mathrm{tr}(\gamma)|X)$. By continuity, these relationships extend also to infinite γ.

We claim that the automaton A has R as its input/output relation. There are two parts to the proof: (1) show that if $y \in R(x)$, then there exists a completed initial computation sequence γ with $x = \mathrm{tr}(\gamma)|X$ and $y = \mathrm{tr}(\gamma)|Y$; (2) show that if γ is a completed initial computation sequence, then $\mathrm{tr}(\gamma)|Y \in R(\mathrm{tr}(\gamma)|X)$.

(1) Suppose $y \in R(x)$. Then by hypothesis, there exists $z \in \bar{Z}$ such that $z \in \mu(G(x))$ and $y = F'(x)(z)$. Choose sequences $a_1, a_2, \ldots \in X \cup \{\epsilon\}$, $b_1, b_2, \ldots \in Y \cup \{\epsilon\}$, and $c_1, c_2, \ldots \in Z \cup \{\epsilon\}$, such that $x = a_1a_2\ldots$, $y = b_1b_2\ldots$, and $z = c_1c_2\ldots$. These sequences exist by the fact that trace domains are ω-algebraic and *finitary* (there are at most finitely many prefixes of any finite trace). Also, because the map $\mu : D \to \bar{Z}$ preserves prime intervals, we may choose $c_1, c_2, \ldots$ so that $c_1c_2\ldots c_k \in C(x)$ for all k.

Next, we use these sequences in a "scheduling argument" to construct a sequence $d_1, d_2, \ldots \in E \cup \{\epsilon\}$ such that if $w = d_1d_2\ldots$, then $x = w|X$, $y = w|Y$, and $z = w|Z$. We do this as follows: Suppose we have defined $d_1, d_2, \ldots, d_k$, for some $k \geq 0$. Suppose further that

$$\begin{aligned} x_k &= a_1a_2\ldots a_l = (d_1d_2\ldots d_k)|X, \\ y_k &= b_1b_2\ldots b_m = (d_1d_2\ldots d_k)|Y, \\ z_k &= c_1c_2\ldots c_n = (d_1d_2\ldots d_k)|Z. \end{aligned}$$

Define d_{k+1} as follows:

- Suppose $k \bmod 3 = 0$. Then let $d_{k+1} = a_{l+1}$.
- Suppose $k \bmod 3 = 1$. If $y_kb_{m+1} \sqsubseteq F'(x_k)(z_k)$, then let $d_{k+1} = b_{m+1}$, otherwise let $d_{k+1} = \epsilon$.
- Suppose $k \bmod 3 = 2$. If $z_kc_{n+1} \in C(x_k)$, then let $d_{k+1} = c_{n+1}$, otherwise let $d_{k+1} = \epsilon$.

It is clear from the construction that $w|X = x$, $w|Y \sqsubseteq y$, and $w|Z \sqsubseteq z$. Moreover, since the domain $\bar{Z}$ is algebraic, and the function G is continuous, given any finite $z' \sqsubseteq z$, there exists K such that $z' \in C(x_k)$ for all $k \geq K$, thus $z' \sqsubseteq z_k$ for sufficiently large k. Since z' may be arbitrary, we conclude $w|Z = z$. Similar reasoning shows that $w|Y = y$.

Now, it follows from the definition of A and the construction of the sequence $d_1, d_2, \ldots$, that there exist states $q^{\mathrm{I}} = q_0, q_1, \ldots$, such that for each $k \geq 0$ we have $q_k \xrightarrow{d_{k+1}} q_{k+1}$, and thus

$$\gamma = q_0 \xrightarrow{d_1} q_1 \xrightarrow{d_2} \ldots$$

is an initial computation for A. We claim that γ is completed. For if not, then there would exist δ with $\gamma \sqsubseteq \delta$ but $\gamma \neq \delta$, such that $\mathrm{tr}(\delta)|X = x = \mathrm{tr}(\gamma)|X$. By the injectiveness of the map tr, we would have either $\mathrm{tr}(\gamma)|Z \sqsubset \mathrm{tr}(\delta)|Z$ or else $\mathrm{tr}(\gamma)|Y \sqsubset \mathrm{tr}(\delta)|Y$. The former is impossible because $z = \mathrm{tr}(\gamma)|Z$ is maximal in $C(x) = \mu(G(x)^c)$, and then the latter is also impossible because $\mathrm{tr}(\delta)|Y$ is functionally determined (via F') by $\mathrm{tr}(\delta)|X = x$ and $\mathrm{tr}(\delta)|Z = z$.

(2) Suppose γ is a completed initial computation. Let $x = \mathrm{tr}(\gamma)|X$, $y = \mathrm{tr}(\gamma)|Y$, and $z = \mathrm{tr}(\gamma)|Z$. As previously observed, we know that $y \sqsubseteq F'(x)(z)$ and $z \in C(x)$. We claim that in fact $y = F'(x)(z)$ and $z \in \mu(G(x))$. First, suppose $y \sqsubset F'(x)(z)$. Then there is some finite $y'' \sqsubseteq F'(x)(z)$, such that $y'' \not\sqsubseteq y$. Assume y'' is chosen to be of minimal length, then $y'' = y'b$ where $y' \sqsubseteq y$ and b is an output action. Using the algebraicity of $\bar{Y}$ and the continuity of F', we can obtain finite $x' \sqsubseteq x$ and $z' \sqsubseteq z$ such that $y'b \sqsubseteq F'(x')(z')$. Choose a finite prefix γ' of γ such that $x' \sqsubseteq \mathrm{tr}(\gamma')|X$, $y' \sqsubseteq \mathrm{tr}(\gamma')|Y$, and $z' \sqsubseteq \mathrm{tr}(\gamma')|Z$. By the definition of A, and the monotonicity of F', we must have b enabled in state $cod(\gamma')$. If $r = cod(\gamma')$ and t is the transition

$r \xrightarrow{b} rb$, then $\mathrm{tr}(\gamma)$ and $\mathrm{tr}(\gamma' t)$ are consistent, hence γ and $\gamma' t$ are consistent, and they have a $\sqsubseteq$-upper bound $\delta \neq \gamma$. Since this contradicts the assumption that γ is completed, we conclude that $y = F'(x)(z)$.

Finally, suppose $z \notin \mu(G(x))$. Because $z \in C(x) = \mu(G(x)^c)$, we have $z \sqsubset \mu(d)$ for some $d \in G(x)$. Therefore, there must exist some finite $z'' \in C(x)$, such that $z'' \not\sqsubseteq z$, but with z and z'' consistent and $z \sqcup z'' \sqsubseteq \mu(d)$. The argument then proceeds for z'' similarly to the above for y'', contradicting the assumption that γ is completed, and concluding that $z \in \mu(G(x))$. ∎

6 Specializations of the Main Result

Our result specializes to two interesting subclasses of automata: "determinate" automata and "semi-determinate" automata. In the case of determinate automata, we restrict G to be single-valued in Theorem 1. The theorem then becomes the statement that the determinate automata compute exactly the class of continuous functions from input to output. Although this result has perhaps been known in various forms for a long time, the interesting point here is how we obtain it as a specialization of a general result. In the case of semi-determinate automata, we restrict G to be a constant function in Theorem 1. The theorem then states that semi-determinate automata compute exactly those relations R such that $R(x) = \{F_i(x) : i \in G(\epsilon)\}$, where the F_i are continuous and $G(\epsilon)$ may be thought of as an "oracle set."

An interesting consequence of the latter result is that there are relations (*e.g.* **amerge**) that are implementable by monotone automata, but cannot be represented in oracleized form as above, hence cannot be implemented by semi-determinate automata. Since **imerge** can be implemented by a semi-determinate automaton, this gives us a separation in expressive power between **amerge** and **imerge**.

6.1 Determinate Automata

An automaton is *determinate* if it satisfies the following condition:

(Determinacy) Suppose $b : q \to r$ and $b' : q \to r'$, where b and b' are distinct non-input actions. Then $b||b'$.

Intuitively, a determinate automaton exhibits no "internal nondeterminism"—the only possible nondeterministic choices are those that occur between input transitions. The determinacy property may also be seen as a kind of Church-Rosser or "diamond" property for non-input actions.

Call coinitial computations γ and δ *input-consistent* if their input traces $\mathrm{tr}(\gamma)|X$ and $\mathrm{tr}(\delta)|X$ are consistent.

The following lemma gives the characteristic property of determinate automata:

Lemma 8 *Suppose A is a determinate automaton. Then two coinitial computations γ and δ of A are consistent iff they are input-consistent.*

Proof – The proof may be accomplished by an inductive argument in which residuals are used to construct the lub of two coinitial input-consistent computations γ and δ. (See [21].) ∎

Theorem 2 *For determinate automata, Theorem 1 holds if we restrict G to be single-valued.*

Proof – Suppose $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$, where G is single-valued and F and G are as in Theorem 1. It is straightforward to check that application of the construction in the proof of Lemma 7 yields a determinate automaton.

Conversely, suppose A is a determinate automaton with input/output relation R. Lemma 4 and Lemma 8, together with an application of Zorn's Lemma, show that for each $x \in \bar{X}$, there is a unique completed computation γ_x with $\mathrm{tr}(\gamma_x)|X = x$. Hence G is single-valued. ∎

Corollary 9 *Suppose $R : \bar{X} \to 2^{\bar{Y}}$. Then R is the input/output relation of a determinate automaton iff $R = I_Y \circ H$, where $H : \bar{X} \to \bar{Y}$ is continuous.*

Proof – Given H, we may take D to be a one-element domain, and G to be the identically $\perp$ function, which is clearly single-valued. ∎

Since there exist nonfunctional relations (*e.g.* **imerge** and **amerge**) that are computable by monotone automata, it follows from the previous theorem that the class of determinate automata is strictly weaker in expressive power than the class of all monotone automata.

6.2 Semi-Determinate Automata

An automaton A is *semi-determinate* if it satisfies the following condition:

(Semi-Determinacy) There exists a collection $C \subseteq E \setminus (X \cup Y)$ of *choice actions*, such that:

1. Whenever $q \xrightarrow{a} r$ and $r \xrightarrow{c} s$, where $c \in C$ and $a \notin C$, then $a||c$ and c is enabled in state q.
2. The automaton, formed from A by deleting all elements of C from each set $T(q, r)$, is determinate.

It is not difficult to see from this definition that if there exists a set C with these properties, then there is a unique largest such set. When we speak of *the* set C

of choice actions for a particular semi-determinate automaton, we refer to the largest such set. A *pure-choice* computation will be a computation in which only choice actions occur.

Lemma 10 *Suppose A is semi-determinate. Then every finite computation γ of A can be factored uniquely as $\gamma = \gamma'\gamma''$, where γ' contains only choice actions, and γ'' contains no choice actions. Moreover, if $\gamma = \gamma'\gamma''$ and $\delta = \delta'\delta''$ are two such factorizations, where γ and δ are input-consistent, then γ and δ are consistent iff γ' and δ' are consistent.*

Proof – The required factorization is obtained by using property (1) in the definition of semi-determinacy, and the commutativity property in the definition of automata, to "permute all choice actions to the front." The consistency assertion then follows using property (2) in the definition of semi-determinacy. A formal proof may be carried out by induction on the length of a computation sequence. ∎

Theorem 3 *For semi-determinate automata, Theorem 1 holds if we restrict G to be a constant function.*

Proof – Given D, F, and constant function G, let A be constructed as in the proof of Lemma 7. Because the enabling of actions in Z depends only on the $\bar{Z}^{\circ}$ component of the state, it follows that A is semi-determinate with $C = Z$.

Conversely, given A, we use a slight modification of the construction in the proof of Lemma 6. Let D be the domain of all initial pure-choice computations. For each $x \in \bar{X}$, let $G(x)$ be the set of all maximal initial pure-choice computations. Then $G(x)$ is obviously a fringe set, and G is constant, hence continuous. Now, the fact that deleting the choice transitions from A yields a determinate automaton implies that for each $x \in \bar{X}$ and for each initial pure-choice computation δ, there exists a unique completed initial computation $\gamma_{x,\delta}$ of A such that $\mathrm{tr}(\gamma_{x,\delta})|X = x$ and such that δ is the greatest pure-choice computation with $\delta \sqsubseteq \gamma_{x,\delta}$. Let $F(x)$ be the function that takes each $\delta \in D$ to the trace $\mathrm{tr}(\gamma_{x,\delta})|Y$. It is straightforward to see from this definition that $F(x)$ is continuous for each $x \in \bar{X}$, and that F itself is continuous as a function of x. Finally, observe that $R(x) = F(x)^{\dagger}(G(x))$ for all $x \in \bar{X}$. ∎

Corollary 11 *The relation* **imerge** *is the input/output relation of a semi-determinate automaton, but* **amerge** *is not.*

Proof – We may express **imerge** $= F(x)^{\dagger}(G(x))$, where D is the domain of finite and infinite sequences of natural numbers with the prefix order, $G(x)$ is the set of all infinite sequences, and $F(x)$ uses its argument as an oracle to schedule the selection of values from the two input ports. More formally, let X_0 be the subset of X consisting of the actions (p_0, m), and similarly for X_1. Define recursively,

$$\begin{aligned}
F(x)(\epsilon) &= \epsilon \\
F(x)(nd) &= H_0(x|X_0, x|X_1, n+1, d) \\
H_0(x, x', 0, \epsilon) &= \epsilon \\
H_0(x, x', 0, nd) &= H_1(x, x', n+1, d) \\
H_0(\epsilon, x', k+1, d) &= \epsilon \\
H_0((p_0, m)x, x', k+1, d) &= (p_2, m)H_0(x, x', k, d) \\
H_1(x, x', 0, \epsilon) &= \epsilon \\
H_1(x, x', 0, nd) &= H_0(x, x', n+1, d) \\
H_1(x, \epsilon, k+1, d) &= \epsilon \\
H_1(x, (p_1, m)x', k+1, d) &= (p_2, m)H_1(x, x', k, d).
\end{aligned}$$

In contrast, if we could express **amerge** in a similar form, then for any given $d \in D$, the function F_d defined by $F_d(x) = F(x)(d)$ would be a monotone function with the property $F_d(x) \in$ **amerge**(x) for all $x \in \bar{X}$. However, no such function can exist. To see this, observe that we must have $F_d((p_0, 5)) = (p_2, 5)$ and $F_d((p_1, 7)) = (p_2, 7)$, so there is no possible value for $F((p_0, 5)(p_1, 7))$ that will make F_d monotone. Hence, **amerge** is not the input/output relation of a semi-determinate automaton. ∎

7 Networks of Automata

So far, we have avoided entirely the issue of how automata may be composed into networks of communicating, concurrently executing components. What we have established so far is the existence of a hierarchy of monotone input/output automata, consisting of the determinate automata, the semi-determinate automata, and all the monotone automata, and we have established some separation results for this hierarchy. The results so far may be summarized by saying that semi-determinate automata compute a strictly larger class of input/output relations than do determinate automata, and compute a strictly smaller class of input/output relations than do arbitrary monotone input/output automata. However, we would like to say more. We would like our results to imply something about the "implementability" of various relations in terms of networks of "primitive components." To do this, we need to define an operation of parallel composition, by which a collection of automata is combined into a network, and we must show that the various classes of automata are closed under this operation.

7.1 Parallel Composition

Although it would be possible to define parallel composition on collections of unrestricted monotone input/output automata, the definitions are more transparent if we restrict our attention to "monotone port automata," which we defined in Section 3.

Formally, suppose $\mathcal{A} = \{A_i : i \in I\}$ is a collection of port automata, where $A_i = (E_i, X_i, Y_i, Q_i, q_i^{\mathrm{I}}, T_i)$. We say that $\mathcal{A}$ is *compatible* if for all $i, j \in I$, if $i \neq j$ then $E_i \cap (E_j \setminus X_j) \subseteq X_i \cap Y_j$.

If $\mathcal{A}$ is compatible, then its *parallel composition* is the automaton $\prod A_i = (E, X, Y, Q, q^{\mathrm{I}}, T)$, where

- $E = \bigcup E_i$, with $a||b$ iff $a||_i b$ for all $i \in I$ such that both a and b are in E_i.
- $Y = \bigcup Y_i$ and $X = (\bigcup X_i) \setminus Y$.
- $Q = \prod_{i \in I} Q_i$,
- $q^{\mathrm{I}} = (q_i^{\mathrm{I}} : i \in I)$,
- $e \in T((q_i : i \in I), (r_i : i \in I))$ iff for all $i \in I$, either $e \notin E_i$ and $r_i = q_i$, or else $e \in E_i$ and $e \in T_i(q_i, r_i)$.

Intuitively, component automata in a network communicate by transmitting values on shared ports. The outputting of a value v on port p by one component automaton occurs simultaneously with the inputting of value v from port p by all other automata that share port p. The compatibility condition states that only input or output actions may be shared between components, and that each shared action (port) may be an output action (port) for at most one component automaton in a network. It is not our purpose here to further justify this particular definition of parallel composition. The reader may refer to the papers [10, 11, 15] for additional motivation and discussion. Here we wish merely to observe the following:

Theorem 4 *The parallel composition of a compatible collection of port automata is a port automaton. Moreover, the classes of determinate port automata and semi-determinate port automata are closed under parallel composition of compatible collections.*

Proof – The disambiguation, receptivity, and commutativity properties are immediate from the definition and the corresponding properties of the A_i. To show monotonicity, suppose $a \in X$ and $b \in Y$. Then a and b are both port actions, and $port(a) \neq port(b)$. Hence whenever both $a, b \in E_i$, we have $a||_i b$, so $a||b$.

Now, assume the A_i are determinate, and suppose $a, b \in E \setminus X$ are both enabled in state q. If both $a, b \in E_i$, then there are four cases:

1. If both $a, b \in E_i \setminus X_i$, then $a||_i b$ by the determinacy of A_i.
2. If $a \in X_i$ and $b \in E_i \setminus X_i$, then $a||_i b$ by monotonicity of A_i.
3. If $a \in E_i \setminus X_i$ and $b \in X_i$, then $a||_i b$ by monotonicity of A_i.
4. If both $a, b \in X_i$, then $port(a) \neq port(b)$. This is because if $port(a) = port(b)$, then for some $j \in I$ we would have both $a, b \in E_j \setminus X_j$, hence not $a||_j b$, contradicting the determinacy of A_j. Since $port(a) \neq port(b)$, we have $a||_i b$.

Since $a||_i b$ in all four cases, we conclude that $a||b$ and $\prod A_i$ is determinate.

Finally, assume the A_i are semi-determinate, and let C_i be the largest set of choice actions for A_i. We claim that $\prod A_i$ is semi-determinate, with $C = \bigcup C_i$ as a set of choice actions. For each $i \in I$, let A_i' be the determinate automaton obtained from A_i by deleting all elements of C_i from each $T(q, r)$. Similarly, let A' be the automaton from $\prod A_i$ obtained by deleting all elements of C from each $T(q, r)$. Then $A' = \prod A_i'$, so A' is determinate. Now suppose

$$(q_i : i \in I) \xrightarrow{a} (r_i : i \in I) \xrightarrow{c} (s_i : i \in I),$$

where $a \in E \setminus C$ and $c \in C$. Since each C_i is a set of internal actions, by compatibility we have $c \in C_i$ for precisely one $i \in I$. There are two cases: either $a \in E_i \setminus C_i$ or else $a \notin E_i$. If $a \in E_i \setminus C_i$, then since A_i is semi-determinate we know that $a||_i c$ and that c is enabled for A_i in state q_i. Since $c \notin C_j$ for $j \neq i$ it follows that $a||c$ and that c is enabled for A in state $(q_i : i \in I)$. If $a \notin E_i$, then $a||c$ by definition of $||$. Since $q_i = r_i$ we know that c is enabled for A_i in state q_i. Then c is also enabled for A in state $(q_i : i \in I)$. ∎

Now, we can interpret our results as saying something about the implementability of relations in terms of networks of "primitives." For each continuous input/output function, let us choose a "standard" determinate port automaton that computes that function. For example, we may choose the automata that result from the construction of Lemma 7. Similarly, we may choose a particular semi-determinate port automaton that computes **imerge**, and a port automaton that computes **amerge**. Then our results imply:

1. the impossibility of implementing **fmerge** by any network of our standard primitives.
2. the impossibility of implementing **amerge** by any network of our standard functional primitives and our standard **imerge** automaton.
3. the impossibility of implementing **imerge** or **amerge** by any network of our standard functional primitives.

7.2 Finite Indeterminacy

We have not yet answered the question of whether **imerge** can be implemented in terms of our standard functional primitives and our standard **amerge** automaton. Although the answer to this question is "yes,"

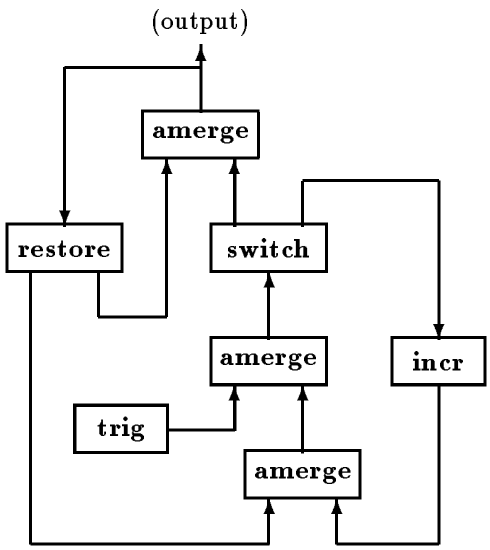

Figure 1: Implementation of Iterated Unbounded Choice by **amerge**.

for a long time it seemed likely to the author that one could prove the opposite by trying to define a class of automata with a property of "finite indeterminacy," and showing that no such automaton could compute **imerge**. This was to be accomplished by proving a version of Theorem 1 for automata with finite indeterminacy in which the sets $G(x)$ were required to have a topological compactness property. After a number of failed attempts, the author realized that it is possible to construct a finite network of standard functional and **amerge** processes that computes **imerge**. One way to interpret this result is that there is can be no definition of "finite indeterminacy" that is satisfied by some standard automaton that computes **amerge**, is not satisfied by any automaton that computes **imerge**, and that is preserved under parallel composition.

Figure 1 shows how **amerge** can be used together with functional processes to construct a network that outputs an arbitrary infinite sequence of nonnegative integers. Once one has such a network, it is straightforward to use it as an "oracle" (as suggested by the proof of Corollary 11) to construct an implementation of **imerge**.

We give only an informal description of the operation of the network, rather than a formal definition and correctness proof. All processes other than the **amerge** processes are functional. All tokens that traverse the channels contain either numeric values or a special "trigger" value, used by the node **switch**. The process **trig** generates an infinite sequence of trigger values. The process **incr** repeatedly reads numeric values from its input, increments them and then issues the incremented values at the output. The process **switch** reads numeric values and outputs them to the right-hand output until a trigger value is read. Once this has happened, the next numeric value is output to the left, and the cycle repeats. Multiple trigger values arriving in succession at the input are treated identically to a single trigger value. The process **restore** initially outputs a zero on its right-hand output and a one on its left-hand output. Once this is done, it repeats the following forever: Read a numeric value from the input; if the value read is zero, output a zero on the right-hand output, otherwise output a one on the left-hand output.

The network operates as follows: Initially, the **restore** process supplies a zero token at one input of the top **amerge** and a one token to the bottom **amerge**, which injects it into the loop. The nonzero token cycles an indeterminate number of times (possibly forever) around the loop, getting its value incremented each time. If a trigger token ever makes it through the middle **amerge** to the **switch**, then the loop will be terminated, however we can't guarantee that this will ever happen. What we do know, though, is that *either* the zero token will make it through the top **amerge**, or a nonzero token will exit the loop and pass through the top **amerge**. Thus, eventually a token will pop out the top, and it is easy to see that any nonnegative integer value is possible.

Once a token has been output, it is the function of the **restore** process to reinitialize the network. If a zero was output, then the **restore** process supplies another zero token to the top **amerge**. If a nonzero value was output, then the **restore** process injects another "one" token into the loop via the bottom **amerge**. Note that it is *not* necessary (and it is in fact impossible) to remove the token still circulating around the loop in the case that a zero was output. Since the circulating token might have any value, simply restoring a zero token to the top **amerge** is sufficient to reinitialize the network.

8 Conclusion

We have defined a class of concurrent automata that can be used to model indeterminate dataflow networks, we have defined two subclasses of automata in terms of their transition structure, and we have obtained a characterization of the input/output relations computed by each class. We have also defined an operation of parallel composition, and observed that each of the three classes of automata is closed under this operation. The three classes (determinate, semi-determinate, and unrestricted) automata form a strict hierarchy with respect to their power to compute relations.

An interesting direction for future research would be to obtain a version of Theorem 1 that characterizes the relations that are computable by automata that are parallel compositions of single-process components. Formally, we may define an automaton to be *single-process* if it satisfies the following condition:

(Single-Process) Suppose $b : q \to r$ and $b' : q \to r'$, where b and b' are distinct non-input actions. Then not $b \| b'$.

Define a *network automaton* to be an automaton of the form $\prod A_i$, where $\{A_i : i \in I\}$ is a compatible collection of single-process automata. The problem is then to find conditions under which we can "decompose" an automaton into a network of single-process automata, and to relate these conditions to the structure of the domain of computations. One condition that does permit us to perform such a decomposition is the to require that the complement $\#$ of the concurrency relation $\|$ on actions be an equivalence relation. We can then recover the "processes" as the equivalence classes of $\#$. At the moment, though, it is not clear how this equivalence relation condition on an automaton is reflected in the structure of its domain of computations.

References

[1] I. J. Aalbersberg and G. Rozenberg. Theory of traces. *Theoretical Computer Science*, 60(1):1–82, 1988.

[2] M. Bednarczyk. *Categories of Asynchronous Systems.* PhD thesis, University of Sussex, October 1987.

[3] J. D. Brock and W. B. Ackerman. Scenarios: a model of non-determinate computation. In *Formalization of Programming Concepts*, pages 252–259, Springer-Verlag. Volume 107 of *Lecture Notes in Computer Science*, 1981.

[4] M. Broy. Nondeterministic data-flow programs: how to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[5] P.-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming. Research Notes in Theoretical Computer Science*, Pitman, London, 1986.

[6] A. A. Faustini. An operational semantics for pure dataflow. In *Automata, Languages, and Programming, 9th Colloquium*, pages 212–224, Springer-Verlag. Volume 140 of *Lecture Notes in Computer Science*, 1982.

[7] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475, North-Holland, 1974.

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

[9] M. Kwiatkowska. *Categories of Asynchronous Systems.* PhD thesis, University of Leicester, May 1989.

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

[11] 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.

[12] A. Mazurkiewicz. Trace theory. In *Advanced Course on Petri Nets*, GMD, Bad Honnef, September 1986.

[13] P. Panangaden. Implementation of amerge with fmerge. June 1988. (Private communication).

[14] P. Panangaden and V. Shanbhogue. *On the Expressive Power of Indeterminate Network Primitives.* Technical Report 87-891, Cornell University Dept. of Computer Science, December 1987.

[15] P. Panangaden and E. W. Stark. Computations, residuals, and the power of indeterminacy. In *Automata, Languages, and Programming*, pages 439–454, Springer-Verlag. Volume 317 of *Lecture Notes in Computer Science*, 1988.

[16] G. D. Plotkin. Domains: lecture notes. 1979. (unpublished manuscript).

[17] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[18] D. S. Scott. *Lectures on a Mathematical Theory of Computation.* Technical Report PRG-19, Oxford University Computing Laboratory, Programming Research Group, May 1981.

[19] M. W. Shields. Deterministic asynchronous automata. In *Formal Methods in Programming*, North-Holland. 1985.

[20] E. W. Stark. Compositional relational semantics for indeterminate dataflow networks. In *Category Theory and Computer Science*, pages 52–74, Springer-Verlag. Volume 389 of *Lecture Notes in Computer Science*, Manchester, U. K., 1989.

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

[22] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64:221–269, 1989.

[23] E. W. Stark. Connections between a concrete and abstract model of concurrent systems. In *Fifth Conference on the Mathematical Foundations of Programming Semantics*, Springer-Verlag. *Lecture Notes in Computer Science*, New Orleans, LA, 1990. (to appear).

[24] G. Winskel. *Events in Computation.* PhD thesis, University of Edinburgh, 1980.