Stability and Sequentiality in Dataflow Networks

Prakash Panangaden* Vasant Shanbhogue
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

November 17, 1989

Abstract

The class of *monotone input/output automata* has been shown in the authors' previous work to be a useful operational model for dataflow-style networks of communicating processes. An interesting class of problems arising from this model are those that concern the relationship between the input/output behavior of automata to the structure of their transition graphs. In this paper, we restrict our attention to the subclass of *determinate* automata, which compute continuous functions, and we characterize classes of determinate automata that compute: (1) the class of functions that are *stable* in the sense of Berry, and (2) the class of functions that are *sequential* in the sense of Kahn and Plotkin.

1 Introduction

The results reported in this paper are part of a general program aimed at relating the input/output behavior of dataflow-like networks of communicating processes, modeled as automata, to the structure of their transition graphs. In previous work [12], we identified the class of *monotone input/output automata* as a useful operational model of an interestingly large class of dataflow networks, and we established a monotonicity property for the input/output relations computed by such networks. These results have subsequently been generalized in [18], where characterizations are given for the relations computed by the full class of monotone input/output automata, and for a subclass of "semi-determinate" automata, which exhibit a more limited form of indeterminacy. This type of result has interesting implications with respect to the power of dataflow networks to perform various kinds of computational tasks; for example, "merging" operations [11].

In this paper, we are concerned with *determinate* automata, which are a subclass of monotone input/output automata whose transition graphs satisfy a kind of Church-Rosser or "diamond" property. It follows from this property that determinate automata compute continuous functions from inputs to outputs. We are interested in finding structural characterizations of classes of automata that correspond to interesting classes of continuous functions. Here, we identify a class of automata that computes exactly the "stable" functions, and a class that computes exactly the "sequential" functions. It should be pointed out that we are not interested in classes of automata

*Research supported in part by NSF Grant CCR-8818979.

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

defined in an arbitrary way. Rather, we are interested only in classes that are defined in terms of *local* properties of their transition graphs. By a local property we mean one that depends on a fixed number of transitions and, hence, is not couched in terms of quantification over arbitrarily long computation sequences. Also, interesting classes of automata should be closed under the operations of "network algebra," by which networks are built from component automata. The classes of stable and sequential automata we define satisfy these conditions.

The "dataflow-like" networks of communicating processes that motivate us are of a type that was first introduced by Kahn [6]. In his paper, Kahn described a class of networks of processes communicating with each other by sending messages containing data values over unidirectional FIFO channels of unbounded capacity. The size and communication topology of a network did not change during execution. Communication between processes was asynchronous and read operations on input channels were assumed to block until data became available. In the denotational semantics given by Kahn, processes were modelled by continuous functions between prefix-ordered domains of sequences, or "streams." He observed that networks with feedback loops computed functions that were related to the functions computed by the component processes according to a natural least-fixed-point principle. This was a very pleasing application of Scott's ideas to a parallel programming situation.

Subsequently Kahn and Plotkin [7] introduced a general class of domains, called *concrete domains*, that generalized the stream domains originally used by Kahn and, more importantly, permitted a general definition of *sequential* function. One motivation for the definition of sequentiality was to try to obtain fully abstract models of the typed lambda calculus. Work by Plotkin [13] showed that the "natural" continuous function models failed to be fully abstract because functions like "parallel OR" existed as values in the semantic function spaces, but could not be implemented in the typed lambda calculus with a sequential interpreter. Unfortunately, the Kahn-Plotkin definition could not be extended to higher types, so it did not yield a fully abstract model. Berry [3] defined a class of domains, the dI-domains, and *stable* functions between them in order to get a definition that both extended to higher types and also ruled out parallel OR. Unfortunately this effort also failed: though stability did succeed in ruling out parallel OR it did include some functions that were intuitively not sequential [5].

Although there has been a reasonably large amount of work on defining various classes of domains and continuous functions between them, there has been comparatively little work on relating these more abstract formulations to a concrete operational semantics. At least in the case of dataflow networks, perhaps the reason has been the lack of a suitable operational framework in which to perform such an investigation. The results of this paper and others [12, 18, 16, 15, 9], strengthen the authors' conviction that monotone input/output automata provide such a framework. The key feature of these automata that permits the operational analysis of networks is the fact that concurrency information is explicitly represented, allowing us to construct a domain of "concurrent computations" whose structure is quite closely linked to the input/output behavior of the automata.

2 Overview of Results

In this section, we give just enough of the formal definitions to permit the statement of our results. The central definition is that of a monotone input/output automaton, which is a kind of nondeterministic transition system that models a system that receives input stimuli from its environment and produces output responses to its environment. The term "monotone" refers to the fact that for these automata, the arrival of an input stimulus can enable an output response, but can never disable one that was previously enabled. A monotone automaton also incorporates some information about concurrency, in the form of a binary "concurrency relation" on the set of actions labeling the transitions of the automaton. The definition of a monotone automaton requires that the concurrency information be reflected in the transition structure in a suitable sense. The particular automata used here have been studied previously by Stark [18, 17]. Related automata have also been studied by Bednarczyk [2], Kwiatkowska [8], and Shields [14].

To define monotone automata, we first need the notion of a "concurrent alphabet," which intuitively consists of a set of actions equipped with a description of which pairs of actions are "concurrent" or "commuting." Formally, a *concurrent alphabet* is a set E, equipped with a symmetric, irreflexive binary relation $\|_E$, called the *concurrency relation.* We usually drop the subscript of $\|_E$ when no confusion is likely. If E and F are concurrent alphabets, then an *isomorphism* from E to F is a bijection $\phi : E \to F$ such that for all $e, e' \in E$ we have $e \|_E e'$ iff $\phi(e) \|_F \phi(e')$. The *direct product* of concurrent alphabets E and F is the concurrent alphabet $E \otimes F$ whose set of elements is the disjoint union of $E + F$ of E and F, and whose concurrency relation $\|_{E\otimes F}$ is defined to be

$$\|_{E\otimes F} \;\;=\;\; \|_E \,\cup\, \|_F \,\cup (E \times F) \cup (F \times E).$$

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

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

where

- $E \simeq X \otimes Y \otimes Z$ is a concurrent alphabet of *actions*, with $X, Y, Z \subseteq E$ the concurrent alphabets of *input actions*, *output actions*, and *internal actions*, respectively. Actions in $E \setminus X$ are called *non-input actions*, and those in $E \setminus Y$ are called *non-output actions.*
- Q is a set of *states*, and $\mathrm{I} \in Q$ is a distinguished *initial state.*
- T is a *transition 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 e, e', if $e \| e'$, $e \in T(q, r)$, and $e' \in T(q, r')$, then there exists a state s such that $e \in T(r', s)$ and $e' \in T(r, s)$.

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

A *transition* of A is a triple $q \xrightarrow{e} r$, where $e \in T(q, r)$. We write $t : q \xrightarrow{e} r$, or just $q \xrightarrow{e} r$, to assert the existence of a transition $t = (q \xrightarrow{e} r)$ of A. Intuitively, a transition $q \xrightarrow{e} r$ represents a potential computation step of A in which action e occurs and the state changes from q to r. We say that action $e \in E$ is *enabled* in state q if there exists a transition $q \xrightarrow{e} r$ in T. By the disambiguation condition, if $q \xrightarrow{e} r$, then r is uniquely determined by q and e, and we sometimes use the notation qe to denote the state r. If $t : q \xrightarrow{e} r$, then q is called the *domain* $\mathrm{dom}(t)$ of t and r is called the *codomain* $\mathrm{cod}(t)$ of t. Transitions t and u are called *coinitial* if $\mathrm{dom}(t) = \mathrm{dom}(u)$.

The term "monotone" refers to the following easily proved property of automata.

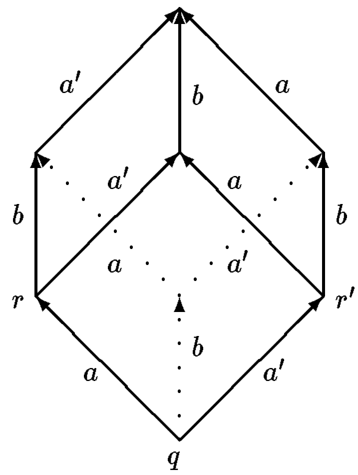


Figure 1: Stability Property for Automata

Lemma 2.1 *Suppose A is an automaton, and non-input action e is enabled in state q. Then for all input actions a, action e is also enabled in state qa.*

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

(Determinacy) $b \| b'$ whenever $b \in T(q, r)$ and $b' \in T(q, r')$, with b and b' distinct non-input actions.

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.

An automaton is *stable* if it is determinate and has the additional property (see Figure 1):

(Stability) Suppose b is a non-input action and a, a' are arbitrary actions, such that $a \| a'$, $a \| b$, and $a' \| b$. Suppose further that $a \in T(q, r)$, $a' \in T(q, r')$, and that b is enabled in states r and r'. Then b is also enabled in state q.

Because of the receptivity condition in the definition of an automaton, a stable automaton automatically also satisfies the stability condition in the case that b is an input action.

An automaton is *sequential* if it is determinate and has the additional properties (see Figure 2):

(Sequentiality)

1. Suppose b is an non-input action and a, a' are arbitrary actions, such that $a \| a'$, $a \| b$, and $a' \| b$. Suppose further that $a \in T(q, r)$, $a' \in T(q, r')$, and b is enabled in state $ra' = r'a$. Then exactly one of the following holds:
   (a) b is enabled in state q.
   (b) b is enabled in state r but not in state r'.
   (c) b is enabled in state r' but not in state r.

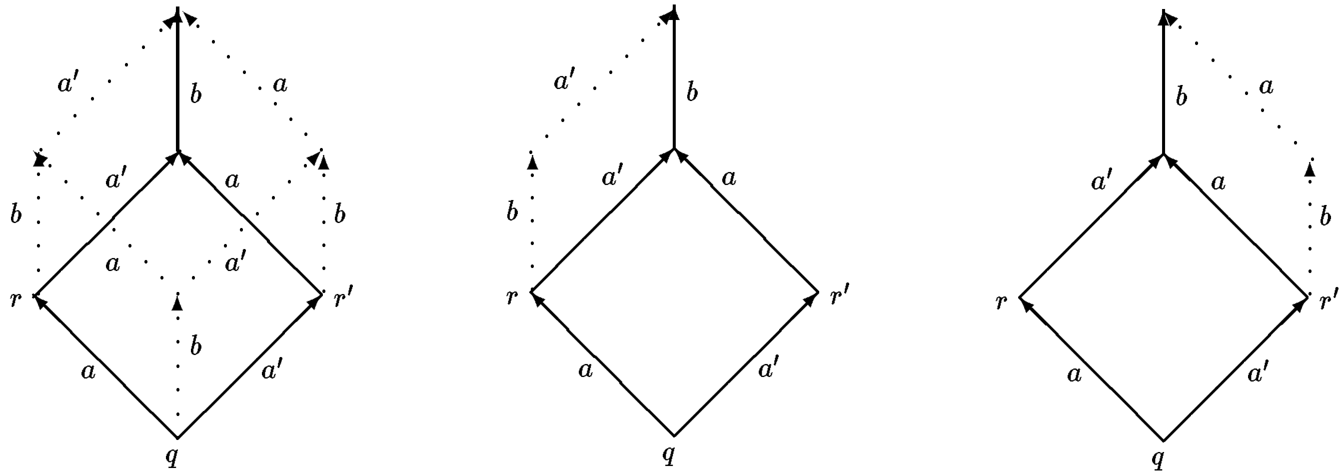


Figure 2: Sequentiality Property for Automata

2. The complement of the concurrency relation $||$ is an equivalence relation with a finite number of equivalence classes.

An automaton is *strictly sequential* if it is sequential and in addition no two non-input actions are concurrent.

The following is obvious from the definitions.

Lemma 2.2 *If an automaton is sequential, then it is stable.*

Our results concern the correspondence between the classes of determinate, stable, and sequential automata defined above, and certain classes of continuous functions between Scott domains. To state these results in a completely precise fashion requires that we describe how to extract the input/output behavior of an automaton. The formal definitions involve some technical details, which we postpone for the moment to allow us to get right to the statement of the results. All that is necessary for now is to assume that there is a natural way to associate with each concurrent alphabet E a Scott domain $\bar{E}$ of "traces over E," and to associate with each automaton A, having input alphabet X and output alphabet Y, a subset of $\bar{X} \times \bar{Y}$, called the "input/output relation" of A. If the input/output relation happens to be the graph of a function $f : \bar{X} \to \bar{Y}$, then we call f the "function computed by A."

The following correspondence between determinate automata and continuous functions was shown in [15], and forms a prototype after which our new results are patterned.

Theorem 1 *Determinate automata compute functions. Moreover, a function $f : \bar{X} \to \bar{Y}$ is the function computed by a determinate automaton iff f is a continuous function.*

Proof – Immediate from Proposition 4.4 and Proposition 4.5 in Section 4. ∎

Our first new result concerns the relationship between the class of stable automata defined above and the class of "stable functions" studied by Berry [3]. Berry originally defined the notion of a stable function in an attempt to provide fully abstract models of the λ-calculus. The definition was intended to rule out functions like "parallel OR" that are not implementable in the lambda calculus with a sequential operational semantics but were nevertheless elements of the semantic domain. Intuitively, for a stable function one can specify the minimum input needed to produce a given output relative to a given input/output pair. Formally, a function $f : \bar{X} \to \bar{Y}$ is *stable* if for all $x \in X$ and $y \sqsubseteq f(x)$, there exists a least $x' \sqsubseteq x$ such that $y \sqsubseteq f(x')$.

Theorem 2 *A function $f : \bar{X} \to \bar{Y}$ is the function computed by a stable automaton iff f is a stable function.*

Proof – Immediate from Proposition 5.2 and Proposition 5.9 in Section 5. ∎

Our second result relates sequential automata to functions that are sequential in the sense of Kahn and Plotkin [7]. The definition of sequential functions used by Kahn and Plotkin applies to continuous functions between "concrete domains." This class of domains has a rather technical definition, which we do not give in this paper, because the method used in proving our result requires that we consider a class of domains that is somewhat smaller than the class of concrete domains. Specifically, our proof applies to functions $f : \bar{X} \to \bar{Y}$, where the concurrent alphabets X and Y are such that the complements $\#_X$ and $\#_Y$ of the concurrency relations $\|_X$ and $\|_Y$ are equivalence relations with a finite number of equivalence classes. We call such concurrent alphabets *port alphabets.* The restriction to port alphabets implies that the associated domains $\bar{X}$ and $\bar{Y}$ are isomorphic to finite cartesian products of prefix-ordered domains of sequences. This class of domains is at least large enough to model dataflow networks, in which each process communicates with its environment by sending and receiving messages containing data values over a fixed, finite number of "input ports" and "output ports." At the moment, we do not know how to extend our result to a larger class of domains.

The formal definition of the class of sequential functions is postponed until Section 7.

Theorem 3 *Suppose X and Y are port alphabets. Then a function $f : \bar{X} \to \bar{Y}$ is the function computed by a sequential automaton iff f is a sequential function.*

Proof – Immediate from Proposition 7.6 and Proposition 7.14 in Section 7. ∎

3 Domains and Traces

To prove our results requires a fairly detailed analysis of the structure of the domains of "concurrent computations" of monotone automata. To express the results of this analysis, it is useful to have at our disposal some terminology from domain theory. In addition, there is a natural way in which each concurrent computation of an automaton determines a "trace," in the sense of Mazurkiewicz [10]. Since much of the structure of the domain of concurrent computations is conveniently described in terms of the mapping from computations to traces, it will therefore also be useful to have available some basic facts about traces. Additional material on trace theory can be found in [1]. A systematic development of the properties of monotone automata and their relationship to trace theory is given in [17].

3.1 Domains

A (Scott) *domain* is an ω-algebraic, consistently complete CPO $D = (D, \sqsubseteq, \perp)$. A subset U of D is *consistent* if it has an upper bound (and hence a least upper bound). If D and E are domains, then a monotone map $f : D \to E$ is *strict* if it preserves $\perp$, and *continuous* if it preserves directed suprema. A *subdomain* of D is a subset U of D, which is a domain under the restriction of the ordering on D, and is such that the inclusion of U in D is continuous. A subdomain U of D is *normal* if for all $d \in D$, the set $\{u \in U : u \sqsubseteq d\}$ is directed.

A domain D is *finitary* if for all finite (=isolated=compact) elements $d \in D$ the set $\{d' \in D : d' \sqsubseteq d\}$ is finite. Domain D is *distributive* if it satisfies the identity $d \sqcap (d' \sqcup d'') = (d \sqcap d') \sqcup (d \sqcap d'')$, whenever d' and d'' are consistent. A finitary, distributive domain has been called a *dI-domain* [3]. A continuous function $f : D \to E$ is called *conditionally multiplicative* (c.m.) if $f(d \sqcap d') = f(d) \sqcap f(d')$, whenever d, d' are consistent elements of D. Obviously, the class of c.m. functions is closed under composition. Berry [3] has shown that for dI-domains, the class of c.m. functions coincides with the class of stable functions.

If D is a domain, then a *complete prime* of D is an element $p \in D$ such that whenever $p \sqsubseteq \bigsqcup U$ for some consistent $U \subseteq D$, then in fact $p \sqsubseteq u$ for some $u \in U$. Define $\mathrm{primes}(d)$ to be the set of all complete primes p such that $p \sqsubseteq d$. The domain D is called *prime algebraic* if $d = \bigsqcup \mathrm{primes}(d)$ for all $d \in D$. It follows easily from these definitions that the set $P(D) = \{\mathrm{primes}(d) : d \in D\}$, equipped with inclusion order, is a domain, and the map $\mathrm{primes} : D \to P(D)$ is an isomorphism. Moreover, if d, d' are consistent elements of D, then $\mathrm{primes}(d \sqcup d') = \mathrm{primes}(d) \cup \mathrm{primes}(d')$ and $\mathrm{primes}(d \sqcap d') = \mathrm{primes}(d) \cap \mathrm{primes}(d')$. It follows from these observations that if a finitary domain is prime algebraic, then it is distributive. Winskel [20, 19] has shown the converse: every finitary, distributive domain is also prime algebraic. Many of the properties of dI-domains become immediately obvious when one uses this representation theorem to translate abstract statements about dI-domains to concrete statements in terms of sets of primes.

3.2 Traces

We now review some basic facts about trace theory. Our presentation here is a bit different than the usual ones, because rather than emphasizing the monoid of finite traces over a concurrent alphabet E, we stress instead the idea of a domain $\bar{E}$ of finite and infinite traces, partially ordered by prefix. We shall see that the domain of concurrent computations of an automaton with alphabet E has a natural embedding as a subdomain of $\bar{E}$. We omit most of the proofs in this section because the results are known, and the proofs are fairly straightforward once one chooses the right concrete representation for traces.

Suppose E is a concurrent alphabet. Let E^* denote the free monoid generated by E, then *permutation equivalence* is the least congruence $\sim_E$ on E^* such that $a \|_E b$ implies $ab \sim_E ba$ for all $a, b \in E$. The quotient $E^*/\sim_E$ is the *free partially commutative monoid* generated by E, and its elements are called *finite traces*. We use ϵ to denote the monoid identity, and if $x \in E^*$, then we use $[x]$ to denote the corresponding element of $E^*/\sim_E$. If $t = [x]$ is a finite trace, then it is clear that all representatives of t have the same length, which we define to be its *length* $|t|$.

Define the relation $\sqsubseteq$ on the monoid $E^*/\sim_E$ by: $t \sqsubseteq u$ iff $\exists v (tv = u)$. The relation $\sqsubseteq$ is called the *prefix* relation on finite traces.

Proposition 3.1 *The monoid $E^*/\sim_E$ has the following properties:*

1. *The relation $\sqsubseteq$ is a partial order, with ϵ as the least element.*

2. *For all t, u, v, if $tu = tv$, then $u = v$.*

3. *For all t, u, if there exists v with $t \sqsubseteq v$ and $u \sqsubseteq v$, then there exists a least such v, which we denote by $t \sqcup u$.*

These properties permit us to define, for consistent traces t, u, the *residual* $t\backslash u$ to be the unique trace v such that $t \sqcup u = uv$. This partial binary operation has many algebraic properties, which are easily derived from the definition in terms of least upper bounds, and which we use here without further comment. For a systematic development, the reader is referred to [16]. However, many properties of the residual operation are suggested by the following multiset representation of traces: If t is a trace, then define $M(t)$ to be the multiset over E that contains $e \in E$ with multiplicity k if and only if k is the number of occurrences of e in t. If t and u are consistent traces, then it can be shown that $t\backslash u$ is the unique suffix of $t \sqcup u$ such that $M(t\backslash u) = M(t \sqcup u) - M(u) = M(t) - M(u)$ (multiset difference), and that the greatest common prefix $t \sqcap u$ of t and u is the unique prefix of t (or u) such that $M(t \sqcap u) = M(t) \cap M(u)$.

As an example of a fact we need that is immediately evident from the multiset representation, we obtain a useful alternative characterization of $t \sqcap u$ in terms of the notion of "orthogonal" or "independent" traces. Formally, define traces t and u to be *orthogonal*, and write $t \perp u$, if every action occuring in t commutes with every action occurring in u. Equivalently, traces t and u are orthogonal if they are consistent and in addition $|t\backslash u| = |t|$ and $|u\backslash t| = |u|$. Then the following is easily shown:

Lemma 3.2 *Suppose t, u are consistent traces. Then $t \sqcap u$ is the unique trace v such that $v \sqsubseteq t$, $v \sqsubseteq u$, and $t\backslash v \perp u\backslash v$.*

Having obtained from a concurrent alphabet E the monoid of finite traces $E^* /\!\sim$, and having observed that $E^* /\!\sim$ is partially ordered by the prefix relation $\sqsubseteq$, we may now apply the standard construction of completion by ideals to obtain a domain $\bar{E}$, whose finite elements are the principal ideals generated by the elements of $E^* /\!\sim$. We may think of the infinite elements of $\bar{E}$ as *infinite traces*; that is, as equivalence classes of infinite strings. We call $\bar{E}$ the *domain of traces generated by* the concurrent alphabet E. Since the finite elements of $\bar{E}$ are in bijective correspondence with the elements of $E^* /\!\sim_E$, they inherit the monoid operation of $E^* /\!\sim_E$, with the least element of $\bar{E}$ as the monoid identity. In the sequel, we identify elements of $E^* /\!\sim$ with the corresponding finite elements of $\bar{E}$.

Proposition 3.3 *Suppose E is a concurrent alphabet. Then the domain of traces $\bar{E}$ is a dI-domain. Moreover, a trace $t \in \bar{E}$ is a complete prime iff it is finite, and for some $e \in E$ there exists no $u \sqsubset t$ such that u and t contain the same number of occurrences of e.*

It is not difficult to see that if X and Y are concurrent alphabets, then $\overline{X \otimes Y} \simeq \bar{X} \times \bar{Y}$. We will make extensive use of this fact. In Section 5 we need to know that the projections $\pi_X : \bar{X} \times \bar{Y} \to \bar{X}$ and $\pi_Y : \bar{X} \times \bar{Y} \to \bar{Y}$ are c.m. This is easily verified using the representation of dI-domains in terms of sets of complete primes.

Lemma 3.4 *Suppose E is a concurrent alphabet of the form $X \otimes Y$. Then the projection $\pi_Y : \bar{E} \to \bar{Y}$ is c.m.*

Proof – We observe that if p is a complete prime in $\bar{X} \otimes \bar{Y}$, then p cannot simultaneously contain occurrences of elements of both X and Y. It is then easy to see that p is a complete prime of $\bar{X} \otimes \bar{Y}$ iff it is either a complete prime of $\bar{X}$ or a complete prime of $\bar{Y}$. Thus, for an arbitrary $t \in \bar{X} \times \bar{Y}$, we have $\mathrm{primes}(\pi_Y(t)) = \pi_Y(\mathrm{primes}(t))$. Since $\mathrm{primes}(\pi_Y(t \sqcap u)) = \pi_Y(\mathrm{primes}(t \sqcap u)) = \pi_Y(\mathrm{primes}(t)) \cap \pi_Y(\mathrm{primes}(u)) = \mathrm{primes}(\pi_Y(t)) \cap \mathrm{primes}(\pi_Y(u))$, the result follows by the isomorphic representation of a dI-domain as an inclusion-ordered domain of sets of complete primes. ∎

4 Computational Behavior of Monotone Automata

In this section, we show how to associate with each monotone automaton a corresponding input/output relation. We first give a standard definition of "computation sequence." Then, just as the concurrency relation $\|_E$ of a concurrent alphabet E induces a permutation equivalence on finite strings over E, the relation $\|_E$ also induces a permutation equivalence on the finite computation sequences of an automaton A having E as its alphabet of actions. This equivalence extends to infinite computation sequences in a natural way, and we define a "computation" to be an equivalence class of computation sequences. The set of all computations of an automaton starting from the initial state is partially ordered by "prefix," and in fact forms a Scott domain $\mathrm{Comp}_{\mathrm{I}}(A)$ with a number of important properties. Many of these properties are consequences of the fact that there is a natural embedding $\mathrm{tr} : \mathrm{Comp}_{\mathrm{I}}(A) \to \bar{E}$ of $\mathrm{Comp}_{\mathrm{I}}(A)$ as a normal subdomain of $\bar{E}$.

Having defined the notion of computation of an automaton A, we then proceed to define the input/output relation of A. Roughly, we want the input/output relation of A to be the set of all $\langle x, y \rangle \in \bar{X} \times \bar{Y}$ such that x is the "input trace" and y is the "output trace," of some computation $\gamma \in \mathrm{Comp}_{\mathrm{I}}(A)$. However, we are not interested in all computations in $\mathrm{Comp}_{\mathrm{I}}(A)$, but rather only those that are "completed" in the sense that every non-input action that becomes enabled and conflicts only finitely often with other enabled non-input actions, eventually occurs. A major advantage of working with monotone automata is that the notion of a completed computation can be formulated as a maximality condition, in terms of the "prefix" ordering on computations.

Finally, having shown how automata determine input/output relations, we observe that determinate automata have functional input/output relations, and that a function is computed by a determinate automaton if and only if that function is continuous.

4.1 Computations

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. An *infinite computation sequence* is an infinite sequence of transitions:

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

We extend notation and terminology for transitions to computation sequences, so that if γ is a computation sequence, then the *domain* $\mathrm{dom}(\gamma)$ of γ is the state q_0, and if γ is finite, then the *codomain* $\mathrm{cod}(\gamma)$ 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 $\mathrm{dom}(\gamma)$ 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 $\mathrm{cod}(\gamma)$ with $\mathrm{dom}(\delta)$. 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 $\preceq \cup \sim$. 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 $\gamma\delta' = \gamma'\delta$.

An extremely useful property of computations is that they, like traces, admit a "residual" operation. Proposition 4.1 below characterizes this operation uniquely. See [17] for an explicit inductive definition and a systematic development of its properties.

Proposition 4.1 *Suppose γ and γ' are consistent finite computations. Then there exists a unique pair of finite computations $\gamma\backslash\gamma'$ and $\gamma'\backslash\gamma$, 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 $(\gamma'\backslash\gamma)\xi = \delta'$.*

Since we shall make extensive use of the algebra of residuals in some of our proofs, we summarize here its most important laws.

Proposition 4.2 *The operation $\backslash$ has the following properties, where γ, δ, and ξ denote computations.*

1. *If $\gamma\backslash\delta$ is defined, then so is $\delta\backslash\gamma$, and we have $\mathrm{dom}(\gamma\backslash\delta) = \mathrm{cod}(\delta)$, $\mathrm{dom}(\delta\backslash\gamma) = \mathrm{cod}(\gamma)$, and $\mathrm{cod}(\gamma\backslash\delta) = \mathrm{cod}(\delta\backslash\gamma)$.*

2. *For all coinitial $\gamma : q \to r$ and $\delta : q \to s$, $\gamma \sqsubseteq \delta$ iff $\gamma\backslash\delta = \mathrm{id}_s$.*

3. *For all $\gamma : q \to r$, we have $\mathrm{id}_q\backslash\gamma = \mathrm{id}_r$, $\gamma\backslash\mathrm{id}_q = \gamma$, and $\gamma\backslash\gamma = \mathrm{id}_r$.*

The following identities hold whenever either side is defined:

3. $(\xi\backslash\gamma)\backslash(\delta\backslash\gamma) = (\xi\backslash\delta)\backslash(\gamma\backslash\delta)$.

4. $$\begin{aligned} \gamma\backslash\delta\xi &= (\gamma\backslash\delta)\backslash\xi \\ \delta\xi\backslash\gamma &= (\delta\backslash\gamma)(\xi\backslash(\gamma\backslash\delta)). \end{aligned}$$

5. $$\begin{aligned} \gamma\backslash(\delta \sqcup \xi) &= (\gamma\backslash\delta)\backslash(\xi\backslash\delta) \\ (\delta \sqcup \xi)\backslash\gamma &= (\delta\backslash\gamma) \sqcup (\xi\backslash\gamma). \end{aligned}$$

We use the notation $\mathrm{Comp}_q(A)$ to denote the set of all computations of automaton A from state q. There is an obvious mapping that takes each computation sequence to the sequence of actions that occur in it, and this mapping induces a function

$$\mathrm{tr} : \mathrm{Comp}_q(A) \to \bar{E}$$

which assigns a trace to each computation.

The following result, proved in [17], gives a great deal of information about the structure of the domain of computations of an automaton.

Proposition 4.3 *For each state q, the set $\mathrm{Comp}_q(A)$, partially ordered by $\sqsubseteq$, is a domain. The map $\mathrm{tr} : \mathrm{Comp}_q(A) \to \bar{E}$ is an embedding of $\mathrm{Comp}_q(E)$ as a normal subdomain of $\bar{E}$. Moreover, the map tr is length-preserving, and has the property that if γ and δ are consistent finite computations, then $\mathrm{tr}(\gamma)$ and $\mathrm{tr}(\delta)$ are consistent traces, and $\mathrm{tr}(\gamma\backslash\delta) = \mathrm{tr}(\gamma)\backslash\mathrm{tr}(\delta)$.*

Computations γ and δ are called *orthogonal*, and we write $\gamma \perp \delta$, if they are consistent, $|\gamma\backslash\delta| = |\gamma|$, and $|\delta\backslash\gamma| = |\delta|$. Since the map tr preserves length and residual, it also preserves orthogonality.

4.2 Input/Output Relations

Suppose A is an automaton. If γ is a computation of A, then the *input trace* of γ is the trace $\mathrm{tr}^{\mathrm{in}}(\gamma) = \pi_X \circ \mathrm{tr}(\gamma)$, where $\pi_X : \bar{E} \to \bar{X}$ is the obvious projection map. Similarly, the *output trace* of γ is the trace $\mathrm{tr}^{\mathrm{out}}(\gamma) = \pi_Y \circ \mathrm{tr}(\gamma)$. A computation γ of A is called *completed* if it is $\sqsubseteq$-maximal among all computations γ' such that $\mathrm{dom}(\gamma') = \mathrm{dom}(\gamma)$ and $\mathrm{tr}^{\mathrm{in}}(\gamma') = \mathrm{tr}^{\mathrm{in}}(\gamma)$. It is shown in [12] that the condition of completedness can in fact be viewed as a "fairness" property, which is satisfied when every non-input action that becomes enabled and conflicts only finitely often with other enabled non-input actions, eventually occurs. In defining the input/output behavior of an automaton, we are only interested in computations having such a fairness property. Thus, we define the *input/output relation* of A to be the set of all $\langle \mathrm{tr}^{\mathrm{in}}(\gamma), \mathrm{tr}^{\mathrm{out}}(\gamma)\rangle \subseteq \bar{X} \times \bar{Y}$, such that γ is a completed initial computation of A.

The following result is shown in [15]:

Proposition 4.4 *Suppose A is a determinate automaton. Then for each $x \in \bar{X}$, there exists a unique completed initial computation $\Gamma_A(x)$ having input trace x. Moreover, the map $\Gamma_A : \bar{X} \to \mathrm{Comp}_{\mathrm{I}}(A)$ is continuous, and the input/output relation of A is the graph*

$$\{\langle x, (\mathrm{tr}^{\mathrm{out}} \circ \Gamma_A)(x)\rangle : x \in \bar{X}\}$$

of the continuous function

$$\mathrm{tr}^{\mathrm{out}} \circ \Gamma_A : \bar{X} \to \bar{Y}.$$

We call the function $\mathrm{tr}^{\mathrm{out}} \circ \Gamma_A$ the *function computed by* the determinate automaton A.

There is a standard way to construct a determinate automaton A from a continuous function f. Simply let the states of A be pairs $\langle x, y\rangle$ of finite elements of $\bar{X} \times \bar{Y}$, where the x component records the input that has arrived so far, and the y component records the output that has been emitted so far. The transitions of the automaton are defined to preserve the invariant relation $y \sqsubseteq f(x)$. We formalize this idea in the following:

Proposition 4.5 *Suppose $f : \bar{X} \to \bar{Y}$ is continuous. Then f is the function computed by a determinate automaton.*

Proof – Define the automaton $A = (E, X, Y, \emptyset, Q, \mathrm{I}, T)$ as follows:

- $Q = (X^* /\sim) \times (Y^* /\sim)$, with $\mathrm{I} = \langle \epsilon, \epsilon\rangle$.
- T is defined as follows: If $a \in X$, then $a \in T(\langle x, y\rangle, \langle x', y'\rangle)$ iff $x' = xa$ and $y' = y$. If $b \in Y$, then $b \in T(\langle x, y\rangle, \langle x', y'\rangle)$ iff $x' = x$, $y' = yb$, and $yb \sqsubseteq f(x)$.

It is straightforward to verify that A satisfies the requirements for a determinate automaton, and that the function computed by A is f. ∎

5 Stability

In this section we show that stable automata compute exactly the stable functions. Recall that a function $f : \bar{X} \to \bar{Y}$ is *stable* if for all $x \in X$ and $y \sqsubseteq f(x)$, there exists a least $x' \sqsubseteq x$ such that $y \sqsubseteq f(x')$. Now, we are interested here only in functions of the form $f : \bar{X} \to \bar{Y}$ where X and Y are concurrent alphabets. For functions between such domains, stability has a simpler characterization.

Lemma 5.1 (Berry) *Suppose D and E are dI-domains. Then a function $f : D \to E$ is stable iff it is c.m.*

By this result, to show that stable automata compute exactly the stable functions, it suffices to show that they compute exactly the c.m. functions.

It is not difficult to show that every c.m. function is the function computed by some stable automaton.

Proposition 5.2 *Suppose $f : \bar{X} \to \bar{Y}$ is a c.m. function. Then f is the function computed by a stable automaton.*

Proof – We verify that the automaton constructed in the proof of Proposition 4.5 is stable. Suppose $a \| a'$, $a \| b$, and $a' \| b$. Let $q = (x, y)$, and suppose b is enabled in states qa and qa'. There are three cases: (1) both a and a' are input actions, (2) a is a non-input action, or (3) a' is an input action.

(1) Suppose both a and a' are input actions. Then $yb \sqsubseteq f(xa)$ and $yb \sqsubseteq f(xa')$, so $yb \sqsubseteq f(xa) \sqcap f(xa') = f(xa \sqcap xa') = f(x)$. Hence b is enabled in state (x, y).

(2) Suppose a is a non-input action. Then $yb \sqsubseteq yba = yab \sqsubseteq f(x)$, hence b is enabled in state (x, y).

(3) Similar to case (2). ∎

We now wish to show that stable automata always compute stable functions. We have already observed that the function f computed by a determinate automaton A factors as $f = \mathrm{tr}^{\mathrm{out}} \circ \Gamma_A$, where $\Gamma_A : \bar{X} \to \mathrm{Comp}_{\mathrm{I}}(A)$ takes each input to the unique completed initial computation for that input, and the function $\mathrm{tr}^{\mathrm{out}} : \mathrm{Comp}_{\mathrm{I}}(A) \to \bar{Y}$ takes computations to their output traces. It is therefore sufficient to show that the functions Γ_A and $\mathrm{tr}^{\mathrm{out}}$ are c.m. if A is stable.

The fact that Γ_A is c.m. for any determinate A is an easy observation:

Lemma 5.3 *Suppose A is a determinate automaton, and let $\Gamma_A : \bar{X} \to \mathrm{Comp}_{\mathrm{I}}(A)$ be the map that takes each trace $x \in \bar{X}$ to the unique completed computation of A having x as its input trace. Then Γ_A is c.m.*

Proof – Suppose $x, x' \in \bar{X}$ are consistent. Then $\Gamma_A(x \sqcap x') \sqsubseteq \Gamma_A(x)$ and also $\Gamma_A(x \sqcap x') \sqsubseteq \Gamma_A(x')$, hence $\Gamma_A(x \sqcap x') \sqsubseteq \Gamma_A(x) \sqcap \Gamma_A(x')$. Conversely, if $\gamma \sqsubseteq \Gamma_A(x)$ and $\gamma \sqsubseteq \Gamma_A(x')$, then $\mathrm{tr}^{\mathrm{in}}(\gamma) \sqsubseteq x \sqcap x'$, hence $\gamma \sqsubseteq \Gamma_A(x \sqcap x')$. Thus, $\Gamma_A(x) \sqcap \Gamma_A(x') \sqsubseteq \Gamma_A(x \sqcap x')$. ∎

To prove that the map $\mathrm{tr}^{\mathrm{out}} : \mathrm{Comp}_{\mathrm{I}}(A) \to \bar{Y}$ is c.m. if A is stable is more difficult, and requires a certain amount of technical analysis of the structure of the domains of computations of stable automata. Our goal is to characterize the meets of computations in terms of the orthogonality relation. Since we know from Section 3 that the map $\mathrm{tr}^{\mathrm{out}}$ preserves length and residual, hence orthogonality, the meet-preserving property of $\mathrm{tr}^{\mathrm{out}}$ will then follow using Lemma 3.2.

We obtain the desired characterization in the next four lemmas, of which the first two apply to arbitrary determinate automata, and the second two apply only to stable automata. The first lemma (Lemma 5.4) says that any prefix of the join of two orthogonal computations itself has an orthogonal join-decomposition. This lemma is used to prove the second (Lemma 5.5), which intuitively says that if cancelling a common prefix from two given computations yields orthogonal results, then the prefix that was cancelled was in fact the greatest common prefix. The third lemma (Lemma 5.6) extends the stability property to "large diamonds" (see Figure 1), whose sides can be arbitrary orthogonal computations rather than just single transitions. The fourth lemma (Lemma 5.7) is a converse to Lemma 5.5, which says that if we cancel the greatest common prefix from two given computations, then the results are orthogonal. In the proofs of these lemmas, we make extensive use of the algebraic properties of residuals (Proposition 4.2).

Lemma 5.4 *Suppose A is an arbitrary automaton, and γ, γ' are finite computations of A with $\gamma \perp \gamma'$. If $\delta \sqsubseteq \gamma \sqcup \gamma'$, then $\delta = \xi \sqcup \xi'$ where $\xi \sqsubseteq \gamma$ and $\xi' \sqsubseteq \gamma'$.*

Proof – We proceed by induction on $|\delta|$. If $|\delta| = 0$, then $\delta = \mathrm{id}$, and we may take $\xi = \xi' = \mathrm{id}$. If $|\delta| > 0$, then $\delta = t\zeta$, where t is a single transition. Then either $t \sqsubseteq \gamma$ and $t \perp \gamma'$, or else $t \sqsubseteq \gamma'$ and $t \perp \gamma$. We consider the first case; the other is symmetric. Since $\gamma \perp \gamma'$, we have also $\gamma \backslash t \perp \gamma' \backslash t = \gamma'$. Since $\delta \sqsubseteq \gamma$ and $\delta \sqsubseteq \gamma'$, we have also $\zeta \sqsubseteq \gamma \backslash t$ and $\zeta \sqsubseteq \gamma'$. We may then apply the induction hypothesis to ζ, $\gamma \backslash t$, and γ' to show that $\zeta = \nu \sqcup \nu'$, where $\nu \sqsubseteq \gamma \backslash t$ and $\nu' \sqsubseteq \gamma'$. But then $\delta = t\nu \sqcup \nu'$, where $t\nu \sqsubseteq \gamma$ and $\nu' \sqsubseteq \gamma'$. ∎

Lemma 5.5 *Suppose A is an arbitrary automaton, and γ and γ' are consistent finite computations of A. If $\delta \sqsubseteq \gamma$, $\delta \sqsubseteq \gamma'$, and $\gamma \backslash \delta \perp \gamma' \backslash \delta$, then $\delta = \gamma \sqcap \gamma'$.*

Proof – (See Figure 3.) If δ satisfies these conditions, then since $\delta \sqsubseteq \gamma$ and $\delta \sqsubseteq \gamma'$, clearly

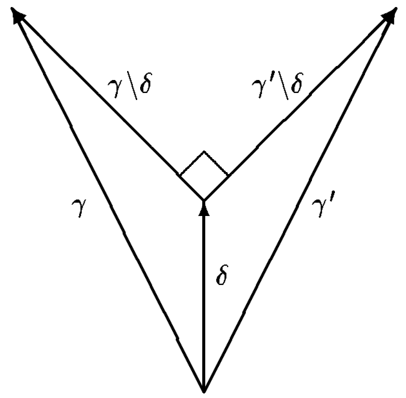


Figure 3: Diagram for Lemma 5.5

$\delta \sqsubseteq \gamma \sqcap \gamma'$. It remains to be shown that $\gamma \sqcap \gamma' \sqsubseteq \delta$, or equivalently, that $(\gamma \sqcap \gamma')\backslash\delta = \mathrm{id}$.

Now, $(\gamma \sqcap \gamma')\backslash\delta \sqsubseteq (\gamma \sqcup \gamma')\backslash\delta = (\gamma\backslash\delta) \sqcup (\gamma'\backslash\delta)$. Since $\gamma\backslash\delta \perp \gamma'\backslash\delta$, by Lemma 5.4 we deduce the existence of $\xi \sqsubseteq \gamma\backslash\delta$ and $\xi' \sqsubseteq \gamma'\backslash\delta$ such that $(\gamma \sqcap \gamma')\backslash\delta = \xi \sqcup \xi'$. From this we get $\gamma \sqcap \gamma' = \delta \sqcup (\gamma \sqcap \gamma') = \delta((\gamma \sqcap \gamma')\backslash\delta) = \delta(\xi \sqcup \xi')$. Thus, $\mathrm{id} = (\gamma \sqcap \gamma')\backslash\gamma = (\delta(\xi \sqcup \xi'))\backslash\gamma$. Using properties of residuals we have

$$\begin{aligned}(\delta(\xi \sqcup \xi'))\backslash\gamma &= (\delta\backslash\gamma)((\xi \sqcup \xi')\backslash(\gamma\backslash\delta)) \\ &= (\xi \sqcup \xi')\backslash(\gamma\backslash\delta) \\ &= (\xi\backslash(\gamma\backslash\delta)) \sqcup (\xi'\backslash(\gamma\backslash\delta)) \\ &= \xi'\backslash(\gamma\backslash\delta).\end{aligned}$$

But we know that $\xi' \perp (\gamma\backslash\delta)$, hence the only way that $\xi'\backslash(\gamma\backslash\delta)$ can be id is if ξ' is id. *Mutatis mutandis* we can show that $\xi = \mathrm{id}$ as well. Thus we see that $(\gamma \sqcap \gamma')\backslash\delta = \mathrm{id}$, which is what we needed to prove. ∎

The next two lemmas apply only to stable automata.

Lemma 5.6 *Suppose A is a stable automaton. Suppose coinitial finite computations γ, γ' and transitions t, t' are such that $\gamma \perp \gamma'$ and $t\backslash(\gamma'\backslash\gamma) = t'\backslash(\gamma\backslash\gamma')$. Then there exists a transition u such that $u\backslash\gamma = t$ and $u\backslash\gamma' = t'$.*

Proof – (See Figure 4.) We first prove the lemma for the special case in which γ' is a single transition v'. This is done by induction on $|\gamma|$. If $|\gamma| = 0$, then $\gamma = \mathrm{id}$ and we may take $u = t$. If $|\gamma| > 0$, then $\gamma = \delta v$, where v is a single transition. Let $v'' = v'\backslash\delta$. Since

$$t\backslash(v''\backslash v) = t'\backslash(\gamma\backslash v') = (t'\backslash(\delta\backslash v'))\backslash(v\backslash v''),$$

we may apply the stability property of A to obtain a transition w such that $w\backslash v = t$ and $w\backslash v'' = t'\backslash(\delta\backslash v')$. Then, since $w\backslash v'' = t'\backslash(\delta\backslash v')$, we may apply the induction hypothesis to obtain a transition u with $u\backslash\delta = w$ and $u\backslash v' = t'$. But then also $u\backslash\gamma = t$, as required.

Now, we use the special case to prove the general case. This is done by induction on $|\gamma'|$. If $|\gamma'| = 0$, then $\gamma' = \mathrm{id}$ and we may take $u = t'$. If $|\gamma'| > 0$, then $\gamma' = \delta'v'$, where v' is a single transition. Let $\gamma'' = \gamma\backslash\delta'$. Then since

$$(t\backslash(\delta'\backslash\gamma))\backslash(v'\backslash\gamma'') = t'\backslash(\gamma''\backslash v'),$$

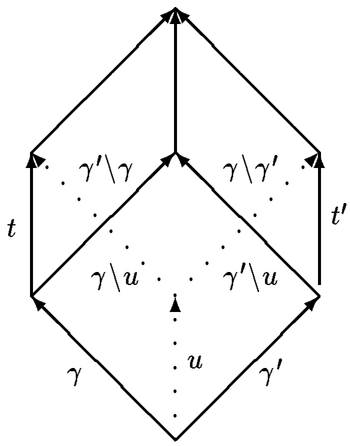


Figure 4: Diagram for Lemma 5.6

we may apply the special case of the lemma to obtain a transition w such that $w\backslash\gamma'' = t\backslash(\delta'\backslash\gamma)$ and $w\backslash v' = t'$. Since $t\backslash(\delta'\backslash\gamma) = w\backslash\gamma''$, we may apply the induction hypothesis to obtain a transition u with $u\backslash\gamma = t$ and $u\backslash\delta' = w$. But then also $u\backslash\gamma' = t'$, as required. ▮

Lemma 5.7 *Suppose A is a stable automaton. Then for all consistent pairs of finite computations γ, γ' of A, we have $\gamma\backslash(\gamma \sqcap \gamma') \perp \gamma'\backslash(\gamma \sqcap \gamma')$.*

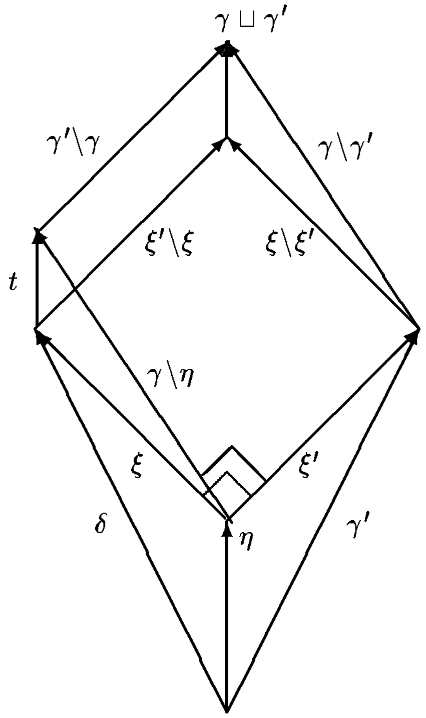
Proof – (See Figure 5.) Proof by induction on pairs (γ, γ'), ordered componentwise by $\sqsubseteq$. For the basis case, if $(\gamma, \gamma') = (\mathrm{id}, \mathrm{id})$, then the result is obvious. For the induction step, suppose we have shown the result for all pairs strictly less than (γ, γ'), where at least one of γ, γ' is not an identity. Suppose $\gamma \neq \mathrm{id}$, the other case is symmetric. Then $\gamma = \delta t$, where t is a single transition. Let $\xi = \delta\backslash(\delta \sqcap \gamma')$ and $\xi' = \gamma'\backslash(\delta \sqcap \gamma')$. By inductive hypothesis, $\xi \perp \xi'$. Observe also that $\xi\backslash\xi' = \delta\backslash\gamma'$ and $\xi'\backslash\xi = \gamma'\backslash\delta$. There are now two cases.

Case $t\backslash(\xi'\backslash\xi) \neq \mathrm{id}$: Then $t \perp (\xi'\backslash\xi)$, so $\xi t \perp \xi'$. But then $\gamma = (\delta \sqcap \gamma')\xi t$ and $\gamma' = (\delta \sqcap \gamma')\xi'$, where $\xi t \perp \xi'$, so by Lemma 5.5 it follows that $\gamma \sqcap \gamma' = \delta \sqcap \gamma'$, $\gamma\backslash(\gamma \sqcap \gamma') = \xi t$, and $\gamma'\backslash(\gamma \sqcap \gamma') = \xi'$. Hence $\gamma\backslash(\gamma \sqcap \gamma') = \xi t \perp \xi' = \gamma'\backslash(\gamma \sqcap \gamma')$, as required.

Case $t\backslash(\xi'\backslash\xi) = \mathrm{id}$: Let ζ' be the greatest computation $\sqsubseteq \xi'$ such that $t\backslash(\zeta'\backslash\xi) \neq \mathrm{id}$. Since $t\backslash(\xi'\backslash\xi) = \mathrm{id}$, there must exist a transition u' such that $t\backslash(\zeta' u'\backslash\xi) = \mathrm{id}$. Then $t\backslash(\zeta'\backslash\xi) = u'\backslash(\xi\backslash\zeta')$ and $\xi \perp \zeta'$, so by Lemma 5.6 there exists a transition v such that $v\backslash\xi = t$ and $v\backslash\zeta' = u'$. Since $\gamma = (\delta \sqcap \gamma')\xi t = (\delta \sqcap \gamma')v(\xi\backslash v)$, $\gamma' = (\delta \sqcap \gamma')v(\xi'\backslash v)$, and $\xi\backslash v \perp \xi'\backslash v$, it follows by Lemma 5.5 that $\gamma \sqcap \gamma' = (\delta \sqcap \gamma')v$, $\gamma\backslash(\gamma \sqcap \gamma') = \xi\backslash v$, and $\gamma'\backslash(\gamma \sqcap \gamma') = \xi'\backslash v$. ▮

Lemma 5.8 *Suppose A is a stable automaton. Then the map* $\mathrm{tr} : \mathrm{Comp}_{\mathrm{I}}(A) \to \bar{E}$ *is c.m.*

Proof – Suppose $\gamma, \gamma' \in \mathrm{Comp}_{\mathrm{I}}(A)$ are consistent and finite. Then by Lemma 5.7 we have $\xi = \gamma\backslash(\gamma \sqcap \gamma') \perp \gamma'\backslash(\gamma \sqcap \gamma') = \xi'$. Since tr preserves orthogonality, $\mathrm{tr}(\xi) \perp \mathrm{tr}(\xi')$. Since also $\mathrm{tr}(\gamma \sqcap \gamma') \sqsubseteq \mathrm{tr}(\gamma)$ and $\mathrm{tr}(\gamma \sqcap \gamma') \sqsubseteq \mathrm{tr}(\gamma')$, it follows by Lemma 3.2 that $\mathrm{tr}(\gamma \sqcap \gamma') = \mathrm{tr}(\gamma) \sqcap \mathrm{tr}(\gamma')$. We have shown that tr is c.m. on finite computations. It follows immediately that it is c.m. on all computations, by the fact that the domain of computations is algebraic. ▮



$t\backslash(\xi'\backslash\xi) \neq id$

$\gamma = \delta t$

$\eta = \delta \sqcap \gamma' = \gamma \sqcap \gamma'$

$t\backslash(\xi'\backslash\xi) = id$

$\gamma = \delta t$

$(\delta \sqcap \gamma')v = \gamma \sqcap \gamma'$

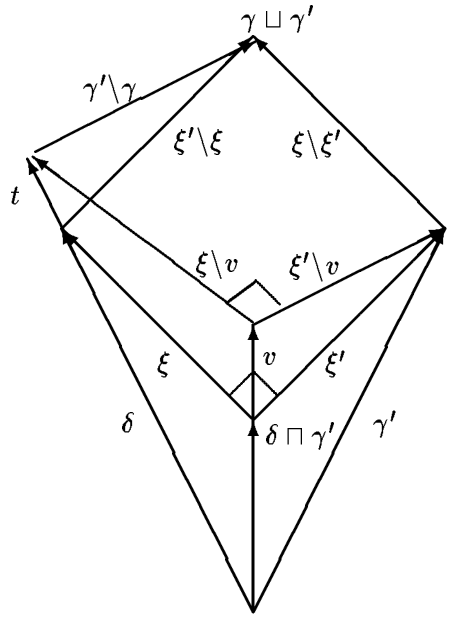


Figure 5: Diagram for Lemma 5.7

We observe in passing that a simple corollary of Lemma 5.8 and Proposition 4.3 is that if A is a stable automaton, then $\mathrm{Comp_I}(A)$ is distributive, hence is a dI-domain. For more general automata, this need not be the case.

Proposition 5.9 *Suppose A is a stable automaton. Then A computes a stable function.*

Proof – If $f : \bar{X} \to \bar{Y}$ is the function computed by A, then $f = \pi_Y \circ \mathrm{tr} \circ \Gamma_A$, where $\Gamma_A : \bar{X} \to \mathrm{Comp_I}(A)$ takes each input to the unique completed initial computation for that input, $\mathrm{tr} : \mathrm{Comp_I}(A) \to \bar{E}$ takes each computation to its trace, and $\pi_Y : \bar{E} \to \bar{Y}$ projects each trace to the corresponding output trace. The function Γ_A is c.m. by Lemma 5.3, the map tr is c.m. by Lemma 5.8, and the map π_Y is c.m. by Lemma 3.4, so f is c.m. as well. Since f is a c.m. function between dI-domains, it is stable by Lemma 5.1. ∎

6 Networks of Automata

In contrast to the "analytic" nature of our proof that stable automata compute stable functions, our proof that sequential automata compute sequential functions has more of a "synthetic" flavor. The reason for this is that we do not know how to do the proof simply by analyzing the structure of the domain of computations of a sequential automaton. Instead, we proceed as follows: (1) We show that strictly sequential automata compute sequential functions. (2) We show that a "network" of sequential automata is sequential, and if each of the component automata in a network computes a sequential function, then the entire network does as well. (3) We show that every sequential automaton is simulated by a network of strictly sequential automata. To prepare the way for such a proof, we need to define a suitable notion of a "network of automata." That is the subject of this section.

6.1 Network Algebra

We define here four algebraic operations on automata: "internalization," which takes some output actions and makes them into internal actions, "tupling," which takes a finite collection of automata, each having the same alphabet of input actions, and places them in parallel with shared input, "sequential composition," which connects two automata in tandem, and "feedback," which takes some outputs of an automaton and feeds them back to some of the inputs. These operations can be used to construct complex networks of automata. It should be noted that we have made no attempt at defining a "complete" and "independent" set of operations; we have merely defined a set of operations that is convenient for the proofs in Section 7. Notably absent are operations for renaming input and output actions, for restricting an automaton to a smaller alphabet of input actions, and for expanding the input and output alphabets of an automaton.

6.1.1 Internalization

Suppose $A = (E, X, Y \otimes V, Z, Q, \mathrm{I}, T)$ is an automaton. Then the *internalization* of V in A is the automaton

$$A \setminus V = (E, X, Y, Z \otimes V, Q, \mathrm{I}, T).$$

6.1.2 Tupling

Suppose $A_1, A_2, \ldots, A_n$ is a finite collection of automata, where $A_i = (E_i, X, Y_i, Z_i, Q_i, \mathrm{I}_i, T_i)$. Then the *tupling* of $A_1, A_2, \ldots A_n$ is the automaton

$$\langle A_1, A_2, \ldots, A_n \rangle = (E, X, Y, Z, Q, \mathrm{I}, T),$$

where $E = X \otimes Y \otimes Z$, $Y = Y_1 \otimes \ldots \otimes Y_n$, $Z = Z_1 \otimes \ldots \otimes Z_n$, $\mathrm{I} = \langle \mathrm{I}_1, \ldots, \mathrm{I}_n \rangle$, and $e \in T(\langle q_1, \ldots, q_n \rangle, \langle r_1, \ldots, r_n \rangle)$ iff one of the following holds:

1. $e \in X$ and $e \in T_i(q_i, r_i)$ for all i.
2. $e \in Y_i \otimes Z_i$, $e \in T_i(q_i, r_i)$, and for all $j \neq i$ we have $r_j = q_j$.

6.1.3 Sequential Composition

Suppose $A = (E \otimes V, X, V, Z, Q, \mathrm{I}, T)$ and $B = (E' \otimes V, V, Y, Z', Q', \mathrm{I}', T')$ are automata. Then the *sequential composition* of A and B is the automaton

$$A; B = (E \otimes E' \otimes V, X, Y, Z \otimes Z' \otimes V, Q \times Q', \langle \mathrm{I}, \mathrm{I}' \rangle, T''),$$

where $e \in T''(\langle q, q' \rangle, \langle r, r' \rangle)$ iff one of the following holds:

1. $e \in E$, $e \in T(q, r)$ and $r' = q'$.
2. $e \in E'$, $e \in T'(q', r')$ and $r = q$.
3. $e \in V$, $e \in T(q, r)$ and $e \in T'(q', r')$.

6.1.4 Feedback

Suppose $A = (E \otimes W, X \otimes W, Y \otimes V, Z, Q, \mathrm{I}, T)$ is an automaton, and $\phi : V \to W$ is an isomorphism of concurrent alphabets. Then the *feedback of A by ϕ* is the automaton

$$A[W \leftarrow \phi(V)] = (E, X, Y \otimes V, Z, Q, \mathrm{I}, T'),$$

where $e \in T'(q, r)$ iff one of the following conditions holds:

1. $e \notin V$ and $e \in T(q, r)$.
2. $e \in V$, $\phi(e) = e'$, and there exists a computation sequence $q \xrightarrow{e} s \xrightarrow{e'} r$ for A.

6.2 Networks of Determinate Automata

It is an important fact that all the classes of automata we have defined are closed under the operations of network algebra.

Lemma 6.1 *The classes, of determinate automata, stable automata, and sequential automata, are all closed under the operations of internalization, tupling, sequential composition, and feedback.*

Proof – The proof is a tedious, but straightforward, use of the definitions. ∎

The following results show how operations on determinate automata correspond to operations on the functions they compute. These results are proved by a systematic analysis of the way in which the operations of network algebra affect the structure of the domain of computations. The only case that is somewhat difficult to prove is the case for feedback, and we refer the reader to [9, 15] for the method.

Proposition 6.2 *For determinate automata:*

1. *If A computes function $f : \bar{X} \to \bar{Y} \times \bar{V}$, then $A \setminus V$ computes function $\pi_Y \circ f : \bar{X} \to \bar{Y}$.*
2. *If automata $A_1, \ldots, A_n$ compute functions $f_1, \ldots, f_n$, respectively, where $f_i : \bar{X} \to \bar{Y}_i$, then $\langle A_1, \ldots, A_n \rangle$ computes function $\langle f_1, \ldots, f_n \rangle : \bar{X} \to \bar{Y}_1 \otimes \ldots \otimes \bar{Y}_n$.*
3. *If A computes function $f : \bar{X} \to \bar{V}$, and B computes function $g : \bar{V} \to \bar{Y}$, then $A; B$ computes function $g \circ f : \bar{X} \to \bar{Y}$.*
4. *Suppose A computes function $f : \bar{X} \times \bar{W} \to \bar{Y} \times \bar{V}$ Let $\phi : V \to W$ be an isomorphism of concurrent alphabets, and let $\bar{\phi} : \bar{V} \to \bar{W}$ be the induced map on traces. Let the functional*

$$\Phi : [\bar{X} \to \bar{Y} \times \bar{V}] \to [\bar{X} \to \bar{Y} \times \bar{V}]$$

be defined by: $\Phi(g)(x) = f(x, (\bar{\phi} \circ \pi_V \circ g)(x))$. Then Φ is continuous, and $A[W \leftarrow \phi(V)]$ computes its least fixed point $\mu\Phi$.

Define a *network of automata* $A_1, \ldots, A_n$ to be an automaton $N(A_1, \ldots, A_n)$, equipped with a specific way to construct it from the automata $A_1, \ldots, A_n$ by a finite number of applications of the operations of network algebra. In a more formal presentation, we would give a formal syntax for network algebra, and then define a network to be a pair consisting of an automaton and a term of network algebra. Such a high degree of formality is unnecessary for the purposes of this paper.

7 Sequentiality

In this section, we establish the connection between sequential automata and functions that are sequential in the sense of Kahn and Plotkin [7]. As mentioned in Section 2, whereas Kahn and Plotkin's original definition of sequentiality applies to continuous functions between arbitrary concrete domains, we work here only with domains $\bar{X}$ and $\bar{Y}$ where X and Y are port alphabets. We therefore state the definition of sequentiality only as it applies to this special case.

Recall that a concurrent alphabet E is a *port alphabet* if the complement $\#_E$ of the concurrency relation $\|_E$ is an equivalence relation with a finite number of equivalence classes. We call the equivalence classes of $\#_E$ the *ports* of E. If E is a port alphabet, then it is easy to see that E is isomorphic to a finite direct product $E_1 \otimes \ldots \otimes E_n$, where each E_i has an empty concurrency relation. Also, the domain $\bar{E}$ of traces is isomorphic to a finite cartesian product $\bar{E}_1 \times \ldots \times \bar{E}_n$, where each $\bar{E}_i$ is a prefix-ordered domain of sequences of actions. If p is a port of E, and $x, x' \in \bar{E}$, then we write $x' \sqsupset_p x$ if $x' \sqsupseteq x$ and $\pi_p(x') \sqsupset \pi_p(x)$, where $\pi_p : \bar{E} \to \bar{E}_p$ is the evident projection.

If X and Y are port alphabets, where X has m ports and Y has n ports, then a function $f : \bar{X} \to \bar{Y}$ can be thought of as having m *inputs* or arguments and producing n *outputs* or results. The function f is *sequential* if it is continuous and the following condition holds:

- Suppose $f(x) = y$, and o is a port of Y. If there exists $x' \sqsupset x$ such that $f(x') \sqsupset_o y$, then there exists a port i of X such that whenever $x' \sqsupset x$ is such that $f(x') \sqsupset_o y$, then also $x' \sqsupset_i x$.

We say then that input on port i is *needed at x* for output on port o. In other words, f is sequential if for all inputs x and all output ports o, if there is some way in which additional input beyond x can cause additional output to be produced on port o, then there exists an input port i such that input on i is needed at x for output on o.

It is easily verified from the above definition that constant functions are sequential, as are all order-isomorphisms. Moreover, the composition of sequential functions is sequential, projections $\pi_X : \bar{X} \times \bar{Y} \to \bar{X}$ and $\pi_Y : \bar{X} \times \bar{Y} \to \bar{Y}$ are sequential, and if $f : \bar{X} \to \bar{Y}$ and $g : \bar{X} \to \bar{Z}$ are sequential, then so is $\langle f, g \rangle : \bar{X} \to \bar{Y} \times \bar{Z}$. We observe also that every one-input, one-output continuous function is sequential.

If a continuous function has more than one input, then it need not be sequential even if it has just one output. One way such a function can fail to be sequential is if it is not stable. As an example, consider the following two-input, one-output function:

$$f : \bar{X}_1 \times \bar{X}_2 \to \bar{Y},$$

where $X_1 = X_2 = Y = \{*\}$. The function f is defined by: $f(x_1, x_2) = *$ if either $(*, \epsilon) \sqsubseteq (x_1, x_2)$ or else $(\epsilon, *) \sqsubseteq (x_1, x_2)$, otherwise $f(x_1, x_2) = \epsilon$.

However, even if a one-output function is stable, it need not be sequential, as is shown by the following instructive three-input, one-output example (due to Berry [4]):

$$f : \bar{X}_1 \times \bar{X}_2 \times \bar{X}_3 \to \bar{Y},$$

where $X_1 = X_2 = X_3 = \{a, b\}$ with no commuting actions, and $\bar{Y} = \{*\}$, the one-action concurrent alphabet. The function f is defined by: $f(x, y, z) = *$, if either $(a, b, \epsilon) \sqsubseteq (x, y, z)$, $(b, \epsilon, a) \sqsubseteq (x, y, z)$, or else $(\epsilon, a, b) \sqsubseteq (x, y, z)$, and $f(x, y, z) = \epsilon$ otherwise. This function is easily seen to be stable, but it is not sequential, because there is no single input port that is needed for the production of the output $*$. Rather, depending on the values that arrive, various combinations of inputs on pairs of ports may cause the output. For this function, the construction in the proof of Proposition 4.5 produces an automaton that is stable, but not sequential, since for example the output $*$ is enabled after a arrives on input 1 and b arrives on input 2, but $*$ is not enabled in the absence of one of these inputs.

We can show that the class of sequential functions is closed under the operations of network algebra.

Proposition 7.1

1. *If function $f : \bar{X} \to \bar{Y} \times \bar{V}$ is sequential, then so is $\pi_Y \circ f : \bar{X} \to \bar{Y}$.*

2. *If $\{f_1, \ldots, f_n\}$ is a finite collection of sequential functions, where $f_i : \bar{X} \to \bar{Y}_i$, then $\langle f_1, \ldots, f_n \rangle : \bar{X} \to \bar{Y}_1 \times \ldots \otimes \bar{Y}_n$ is sequential.*

3. *If $f : \bar{X} \to \bar{V}$ and $g : \bar{V} \to \bar{Y}$ are sequential, then so is $g \circ f : \bar{X} \to \bar{Y}$.*

4. *Suppose function $f : \bar{X} \times \bar{W} \to \bar{Y} \times \bar{V}$ is sequential. Let $\phi : V \to W$ be an isomorphism of concurrent alphabets, and let $\bar{\phi} : \bar{V} \to \bar{W}$ be the induced function on traces. Let*

$$\Phi : [\bar{X} \to \bar{Y} \times \bar{V}] \to [\bar{X} \to \bar{Y} \times \bar{V}]$$

be the continuous functional defined by: $\Phi(g)(x) = f(x, (\bar{\phi} \circ \pi_Y \circ g)(x))$. *Then the least fixed point* $\mu\Phi$ *of* Φ *is sequential.*

Proof – Assertions (1)-(3) are fairly straightforward. We sketch the proof of (4), which is not trivial but also not particularly profound. The basic idea is to prove the following assertion: a chain of sequential functions has a sequential function as its lub. Statement (4) then follows immediately from this in view of the standard characterization of the least fixed point $\mu\Phi$ as the lub of a chain of iterates of Φ.

To prove the assertion, it is convenient to work in terms of concrete data structures as in [5]. If $f : D \to D'$ is a continuous function between the domains of states D and D' of two concrete data structures, then for each $x \in D$ and cell c of D' "accessible" from $f(x)$, we use $N(f, c, x)$ to denote the set of cells of D that are "needed" to fill cell c. The set $N(f, c, x)$ is always finite by the continuity of f. Now, suppose $f_0 \sqsubseteq f_1 \sqsubseteq \ldots$ is a chain of sequential functions, and $f = \bigsqcup_{i \geq 0} f_i$. Then the essential observation is that for some $m \geq 0$, we have $N(f_m, c, x) \supseteq N(f_{m+1}, c, x) \supseteq \ldots$, and $N(f, c, x) = \bigcap_{i \geq m} N(f_i, c, x)$. From this, and the fact that the sequentiality of the f_i implies that each $N(f_i, c, x)$ is nonempty, it follows that $N(f, c, x)$ is nonempty as well, as required to show that f is sequential. ∎

7.1 Automata From Sequential Functions

In this section, we prove that every sequential function is the function computed by some sequential automaton. We do this by showing first that the construction, given in the proof of Proposition 4.5, of an automaton from a function, produces a strictly sequential automaton in the case that the function has one input port and one output port. Next, we show that this construction also produces a strictly sequential automaton for a certain class of multiple-input, one-output functions, called "serializers." Finally, we show that every m-input, n-output sequential function f can be expressed in the form $f = \langle h_1 \circ g_1, \ldots, h_n \circ g_n \rangle$, where $h_1, \ldots, h_n$ are one-input, one-output continuous functions, and $g_1, \ldots, g_n$ are m-input, one-output serializers that merge their inputs onto one output using a strategy that depends on f. It follows from these facts that the function f is computed by a network of strictly sequential automata, which is a sequential automaton by Lemma 6.1.

Lemma 7.2 *If* f *is a constant function or a one-input, one-output continuous function, then* f *is the function computed by a strictly sequential automaton.*

Proof – The construction in the proof of Proposition 4.5 is easily seen to produce strictly sequential automata for such f. ∎

We have already defined in Section 2 the notion of the direct product of concurrent alphabets. Here we need the notion of the *direct sum* of concurrent alphabets E and F. Formally, this is the concurrent alphabet $E \oplus F$ which, like the direct product $E \otimes F$, has the disjoint union $E + F$ as its set of elements, but whose concurrency relation $\|_{E \oplus F}$ is defined to be

$$\|_{E \oplus F} \;=\; \|_E \cup \|_F.$$

Suppose X is a port alphabet with $m \geq 1$ ports, so that $\bar{X} \simeq \bar{X}_1 \times \ldots \times \bar{X}_m$ where $X_1, \ldots, X_m$ are one-port concurrent alphabets. Define $X_\sigma = X_1 \oplus \ldots \oplus X_m$, and if $a \in X_i$, then let $(a)_i$ denote the corresponding element of X_σ. Then the domain $\bar{X}_\sigma$ is just the domain of finite and infinite strings over the disjoint union of the X_i, and there is an obvious quotient map $\eta_X : \bar{X}_\sigma \to \bar{X}$ which is the unique continuous extension of the map that takes a finite string $(a_1)_{i_1} \dots (a_n)_{i_n} \in \bar{X}_\sigma$ to the corresponding trace $a_1 \dots a_n$ in $\bar{X}$.

Define a *serializer of* X to be an m-input, one-output continuous function

$$f : \bar{X}_1 \times \ldots \times \bar{X}_m \to \bar{X}_\sigma,$$

such that:

1. $\eta_X \circ f \sqsubseteq \mathrm{id}_{\bar{X}}$.
2. For all finite $x \in \bar{X}$, if there exists $x' \sqsupset x$ such that $f(x') \sqsupset f(x)$, then there exists a unique port i of $\bar{X}$ such that for all $x' \sqsupset x$ we have $f(x') \sqsupset f(x)$ iff $x' \sqsupset_i x$.

Intuitively, a serializer uses a deterministic strategy to repeatedly select one of its input ports, read one input action from that port, and output that action onto its single output port. Clearly, serializers are sequential functions.

Lemma 7.3 *If f is a serializer, then f is the function computed by a strictly sequential automaton.*

Proof – The construction in the proof of Proposition 4.5 is easily seen to produce strictly sequential automata for these functions. ∎

Lemma 7.4 *Suppose f is a one-output sequential function with at least one input. Then $f = h \circ g$, where g is a serializer and h is a one-input, one-output continuous function.*

Proof – Suppose $f : \bar{X} \to \bar{Y}$, where $X = X_1 \otimes \ldots \otimes X_m$ is such that each X_i has just one port. Define $h = f \circ \eta_X : X_\sigma \to Y$, where X_σ and η_X are defined as above. Clearly, h is a one-input, one-output continuous function.

Call an input $x \in \bar{X}$ *extensible* if there exists $x' \sqsupset x$ such that $f(x') \sqsupset f(x)$. Since f is a one-output sequential function, for each extensible input $x \in \bar{X}$ there exists an input port i that is needed by f at x, in the sense that whenever $x' \sqsupset x$ is such that $f(x') \sqsupset f(x)$, then $x' \sqsupset_i x$. Choose arbitrarily a partial function $s : \bar{X} \to \{1, \ldots, m\}$ that assigns such a needed port to each extensible input $x \in \bar{X}$, and is undefined if input x is not extensible.

Define function $g : \bar{X} \to \bar{X}_\sigma$ and auxiliary function $g' : \bar{X} \times \bar{X}_\sigma \to \bar{X}_\sigma$ to be the least solution to the following recursion equations:

$$\begin{aligned}
g(x_1, \ldots, x_m) &= g'(x_1, \ldots, x_m, \epsilon) \\
g'(x_1, \ldots, x_m, z) &= \begin{cases} \epsilon, & \text{if } s(\eta(z)) \text{ undefined} \\ & \quad \text{or } s(\eta(z)) = i \text{ and } x_i = \epsilon \\ (a)_i g'(x_1, \ldots, x_i', \ldots, x_m, z(a)_i), & \text{if } s(\eta(z)) = i \text{ and } x_i = a x_i'. \end{cases}
\end{aligned}$$

That is, the function g uses s as a "strategy" for selecting inputs one-at-a-time and transferring them to the single output. Clearly, g is a serializer.

To complete the proof, we claim that $f = h \circ g$. Since g is a serializer, $\eta_X \circ g \sqsubseteq \mathrm{id}_X$, so $h \circ g = f \circ \eta_X \circ g \sqsubseteq f$. It remains to be shown that we do not have $h \circ g \sqsubset f$. Suppose we do, then for some finite $x \in \bar{X}$ we have $h(g(x)) \sqsubset f(x)$. Let $z = g(x)$, then we must have $\eta_X(z) \sqsubset x$ by definition of h.

Now, since $\eta_X(z) \sqsubset x$ and $f(\eta_X(z)) = h(z) \sqsubset f(x)$, the input $\eta_X(z)$ is extensible, and it follows by the sequentiality of f that there exists some input port that is needed by f at input $\eta_X(z)$. Thus, we have $s(\eta_X(z)) = i$ for some input port i such that $x \sqsupset_i \eta_X(z)$ does not hold. But then we have $f(\eta_X(z)) \sqsubset f(x)$ and $\eta_X(z) \sqsubset x$, but not $x \sqsupset_i \eta_X(z)$. This is a contradiction with the sequentiality of f. ∎

Lemma 7.5 *Suppose f is an m-input, n-output sequential function, with $m, n \geq 1$. Then $f = \langle h_1 \circ g_1, \ldots, h_n \circ g_n \rangle$, where $g_1, \ldots, g_n$ are serializers, and $h_1, \ldots, h_n$ are one-input, one-output continuous functions.*

Proof – Suppose $f : \bar{X} \to \bar{Y}$, and express Y as the direct product $Y_1 \otimes \ldots \otimes Y_n$ of one-port concurrent alphabets. Then, apply the previous lemma to the functions $\pi_i \circ f$, where $\pi_i : \bar{Y} \to \bar{Y}_i$ is the projection on the ith output. ∎

Proposition 7.6 *Suppose $f : \bar{X} \to \bar{Y}$ is an m-input, n-output sequential function. Then f is the function computed by a sequential automaton.*

Proof – If either $m = 0$ or $n = 0$, then f is constant, hence is computed by a strictly sequential automaton by Lemma 7.2. Suppose $m, n \geq 1$. Then by Lemma 7.5, $f = \langle h_1 \circ g_1, \ldots, h_n \circ g_n \rangle$, where $g_1, \ldots, g_n$ are m-input serializers, and $h_1, \ldots, h_n$ are one-input, one-output continuous functions. By Lemmas 7.2 and 7.3, the g_i and h_i are computed by strictly sequential automata, say A_i and B_i, respectively. Let N be the network $\langle A_1; B_1, \ldots, A_n; B_n \rangle$. Since N is a network of strictly sequential automata, by Lemma 6.1 it is sequential. By Proposition 6.2, N computes f. ∎

7.2 Strictly Sequential Automata

We now turn our attention to the problem of showing that strictly sequential automata compute sequential functions. For this section, let a strictly sequential automaton A be fixed. Let X and Y be the input and output alphabets of A, respectively.

If $x \in \bar{X}$ is a finite trace, and q is a state of A, then we call x an *enabling input* from state q if there are no non-input transitions enabled in state q, but there is a non-input transition enabled in state qx.

Lemma 7.7 *Suppose $x, x' \in \bar{X}$ are finite traces, such that $x \perp x'$ and $x \sqcup x'$ is an enabling input from state q. Then either x is an enabling input from state q, or else x' is an enabling input from state q.*

Proof – Let $x_0 = x$ and $x'_0 = x'$. For $i \geq 0$, we show that if x_i and x'_i are such that both x_i and x'_i are nonempty, $x_i \perp x'_i$, and $x_i \sqcup x'_i$ is an enabling input from state q, then there exist $x_{i+1} \sqsubseteq x_i$ and $x'_{i+1} \sqsubseteq x'_i$ such that $x_{i+1} \sqcup x'_{i+1}$ is an enabling input from state q, but such that $|x_{i+1}| + |x'_{i+1}| < |x_i| + |x'_i|$. Since $|x_0| + |x'_0|$ is finite, we eventually obtain x_n and x'_n such that $x_n \sqcup x'_n$ is an enabling input from state q, and one of x_n and x'_n is empty. If x'_n is empty, then x_n is an enabling input from state q, hence x is an enabling input from state q by monotonicity (Lemma 2.1). Similarly, if x_n is empty, then x' is an enabling input from state q.

So, suppose x_i and x'_i are both nonempty. Then $x_i = za$ and $x'_i = z'a'$, where $a, a' \in X$. Since $x_i \perp x'_i$, we must have $a \| a'$, and $x_i \sqcup x'_i = (z \sqcup z')aa' = (z \sqcup z')a'a$. Now, since $x_i \sqcup x'_i$ is an enabling input from state q, there is some non-input action b that is enabled in state $q(x_i \sqcup x'_i)$. Applying the sequentiality property, there are three possibilities:

1. b is enabled in state $q(z \sqcup z')$. In this case we may take $x_{i+1} = z$ and $x'_{i+1} = z'$.
2. b is enabled in state $q(x_i \sqcup z')$ but not in state $q(z \sqcup x'_i)$. Then $x_i \sqcup z'$ is an enabling input from state q, so we may take $x_{i+1} = x_i$ and $x'_{i+1} = z'$.
3. b is enabled in state $q(z \sqcup x'_i)$ but not in state $q(x_i \sqcup z')$. Then $z \sqcup x'_i$ is an enabling input from state q, so we may take $x_{i+1} = z$ and $x'_{i+1} = x'_i$.

∎

Call a finite trace $x \in \bar{X}$ *single-port* if it consists entirely of actions for one port.

Lemma 7.8 *Suppose a finite trace $x \in \bar{X}$ is an enabling input from state q. Then there exists a single-port $x' \sqsubseteq x$ such that x' is also an enabling input from state q.*

Proof – We show that if x is not single-port, then there exists a prefix x' of x, such that x' is an enabling input from state q, and such that the number of distinct ports on which input occurs in x' is strictly less than the number on which input occurs in x. For, if x is not single port, then it can be decomposed $x = x_i \sqcup x'$, where x_i consists entirely of inputs on port i, and x' is a nonempty trace that contains no inputs on port i, so that $x_i \perp x'$. By the previous lemma, either x_i is an enabling input from state q, or else x' is an enabling input from state q. In the first case, we are done, since x_i is single-port. In the second case, we have reduced by one the number of ports on which input occurs.

Since every finite trace contains input on at most a finite number of ports, repeated application of this argument eventually yields the required single-port enabling input. ∎

Lemma 7.9 *If finite traces $x, x' \in \bar{X}$ are both single-port enabling inputs from state q, then x, x' are inputs for the same port.*

Proof – Suppose x and x' were two single-port enabling inputs from state q, such that x consists of inputs for port i and x' consists of inputs for port $i' \neq i$. Then $x \perp x'$. Suppose non-input action b is enabled by x and non-input action b' is enabled by x'. Then by monotonicity, both b and b' are enabled by $x \sqcup x'$. Since A is strictly sequential, we cannot have $b \| b'$. Since A is determinate, we must have $b = b'$. But then since b is not enabled in state q we have a contradiction with stability by Lemma 5.6. ∎

Say that input port i is *needed at state* q if every enabling input from state q contains some action for port i. It follows from the results above that if there exists an enabling input x from state q, such that x consists entirely of inputs for port i, then in fact port i is needed at state q, and every single-port enabling input from state q consists entirely of inputs for port i.

Lemma 7.10 *Let γ be a finite completed computation. If there exists δ such that $\gamma \sqsubset \delta$ and* $\mathrm{tr}^{\mathrm{out}}(\gamma) \sqsubset \mathrm{tr}^{\mathrm{out}}(\delta)$*, then there exists an input port i that is needed at state* $\mathrm{cod}(\gamma)$*. Moreover, for every such δ we have that* $\mathrm{tr}^{\mathrm{in}}(\delta) \sqsupset_i \mathrm{tr}^{\mathrm{in}}(\gamma)$.

Proof – Suppose γ is a finite completed computation, such that there exists δ as stated. Then there exists some finite input trace x that is an enabling input from state $\mathrm{cod}(\gamma)$. By Lemma 7.8, we may assume without loss of generality that x consists entirely of input for a single port i. Moreover, by Lemma 7.9, any two single-port enabling inputs from $\mathrm{cod}(\gamma)$ consist of inputs for the same port i. Hence port i is needed in state $\mathrm{cod}(\gamma)$, and every δ with the stated properties must be such that $\mathrm{tr}^{\mathrm{in}}(\delta) \sqsupset_i \mathrm{tr}^{\mathrm{in}}(\gamma)$. ∎

Lemma 7.11 *Suppose A is a strictly sequential automaton. Then A computes a sequential function.*

Proof – If A is strictly sequential, then no two non-input actions of A are concurrent, hence A either has one internal port and no output ports, or else A has one output port and no internal ports. If A has no output ports, then the output domain $\bar{Y}$ of A is the trivial one-point domain, and A computes a constant function, hence a sequential function. For the rest of the proof, suppose A has one output port and no internal ports. Let $f : \bar{X} \to \bar{Y}$ be the function computed by A.

By results of Berry and Curien [4], to show that f is sequential, it suffices to show that it is sequential at all finite $x \in \bar{X}$; that is, if $x, x' \in \bar{X}$ are finite traces such that $x' \sqsupset x$ and $f(x') \sqsupset f(x)$, then there exists some input port i that is needed at x. Let $\Gamma_A(x)$ be the unique completed computation for input x, so that $\mathrm{tr}^{\mathrm{out}}(\Gamma_A(x)) = f(x)$. Now $f(x)$ cannot be infinite, since in that case (because there is just one output port) it would be impossible to have $f(x') \sqsupset f(x)$. Hence $f(x)$ is finite. Therefore, there must exist a finite $\gamma \sqsubseteq \Gamma_A(x)$, such that γ is completed, and $\mathrm{tr}^{\mathrm{out}}(\gamma) = f(x)$. Since $f(x') \sqsupset f(x)$, the unique completed computation $\Gamma_A(x')$ on input x' is such that $\Gamma_A(x') \sqsupset \gamma$ and $\mathrm{tr}^{\mathrm{out}}(\Gamma_A(x')) \sqsupset \mathrm{tr}^{\mathrm{out}}(\gamma)$. Thus, by Lemma 7.10 there exists some input port i that is needed at state $\mathrm{cod}(\gamma)$.

Now, if $x'' \in \bar{X}$ is any trace such that $x'' \sqsupset x$ and $f(x'') \sqsupset f(x)$, then we must have $\gamma \sqsubset \Gamma_A(x'')$. Since input i is needed in state $\mathrm{cod}(\gamma)$, and since $\mathrm{tr}^{\mathrm{out}}(\Gamma_A(x'')) = f(x'') \sqsupset f(x) = \mathrm{tr}^{\mathrm{out}}(\Gamma_A(x))$, it follows that $x'' \sqsupset_i x$. We have therefore shown that for any $x'' \sqsupset x$, if $f(x'') \sqsupset f(x)$, then $x'' \sqsupset_i x$. In other words, port i is needed at x. ∎

7.3 Decomposition of Sequential Automata

In this section, we show that every sequential automaton is simulated, in a very strong sense, by a network of strictly sequential automata. It follows from this and the results of the previous section, that every sequential automaton computes a sequential function.

We first define the notion of simulation we use. Suppose A and A' are automata with the same alphabets of input, output, and internal actions. A *simulation of A by A'* is a partial function $h : Q' \to Q$, such that the following conditions hold:

1. $h(\mathrm{I}') = \mathrm{I}$.
2. Suppose $h(q')$ is defined. Then the set of actions enabled for A' in state q' is identical to the set of actions enabled for A in state $h(q')$. Moreover, if $e \in T'(q', r')$, then $h(r')$ is defined and $e \in T(h(q'), h(r'))$.

Lemma 7.12 *Suppose h is a simulation of A by A'. Then the domains of computations of A and A' are isomorphic, hence A and A' have identical input/output relations.*

Proof – Obvious from the definition of simulation. ∎

Lemma 7.13 *If A is a sequential automaton, then A is simulated by some network*

$$N(A_1, \ldots, A_n, B_1, \ldots, B_p),$$

whose component automata $A_1, \ldots, A_n, B_1, \ldots, B_p$ are all strictly sequential.

Proof – Suppose $A = (E, X, Y, Z, Q, \text{I}, T)$. Since A is sequential, the complement $\#_E$ of the concurrency relation $\|_E$ is an equivalence relation with a finite number of equivalence classes. Let n, p be the number of equivalence classes of $\#_E$ that contain, respectively, only output actions, or only internal actions. If $n + p \leq 1$ then A is already strictly sequential and there is nothing to prove, so assume that $n + p > 1$. Suppose further that

$$Y = Y_1 \otimes \ldots \otimes Y_n \qquad Z = Z_1 \otimes \ldots \otimes Z_p,$$

where each of the Y_i and Z_j has exactly one port. Define

$$Y' = Y_1' \otimes \ldots \otimes Y_n' \qquad Z' = Z_1' \otimes \ldots \otimes Z_p' \qquad E' = X \otimes Y' \otimes Z',$$

where $Y_1', \ldots, Y_n'$ and $Z_1', \ldots, Z_p'$ are isomorphic copies of $Y_1, \ldots, Y_n$ and $Z_1, \ldots, Z_p$, respectively. In what follows, if e denotes an action in an unprimed alphabet, say Y_i, then e' will denote the corresponding action in Y_i'. We use the same convention for traces.

The idea of the proof is to construct strictly sequential automata $\{A_i : 1 \leq i \leq n\}$ and $\{B_j : 1 \leq j \leq p\}$, where A_i has input alphabet E' and output alphabet Y_i, and B_j has input alphabet E' and output alphabet Z_j, such that A_i is responsible for ensuring that actions in Y_i occur only in situations where they would be enabled for A, and B_j has a similar responsibility for actions in Z_j. The network $N(A_1, \ldots, A_n, B_1, \ldots, B_p)$ is obtained by tupling the A_i and B_j, feeding back the output of each of the A_i and B_j to the corresponding inputs of all the others, and finally internalizing all actions in Z_j. The reason for feeding back the output of each automaton to the inputs of all the others is to ensure that each automaton has enough information to carry out its responsibilities.

Formally, for each $i \in \{1, \ldots, n\}$, define

$$A_i = (E' \otimes Y_i, E', Y_i, \emptyset, (E')^* /\!\sim, \epsilon, T_i),$$

where $e \in T_i(z', w')$ iff one of the following holds:

1. $e \in Y_i'$ and $w' = z'$ (automaton A_i ignores inputs in Y_i').
2. $e \in E' \setminus Y_i'$ and $w' = z'e$ (automaton A_i records the occurrence of all inputs in $E' \setminus Y_i'$).
3. $e \in Y_i$, $w' = z'e'$, and the trace $w \in \bar{E}$ is enabled for A in state I (automaton A_i permits actions in Y_i to occur only if they would be enabled for A in the same situation).

For each $j \in \{1, \ldots, p\}$, define

$$B_j = (E' \otimes Z_j, E', Z_j, \emptyset, (E')^* /\!\sim, \epsilon, U_j),$$

similarly to A_i, except with Z_j replacing Y_i. The fact that the A_i and B_j are strictly sequential is immediate from the above definitions and the assumption that A is sequential.

We now define (with slight abuse of our network algebra notation):

$$\begin{aligned} N(A_1, \ldots, A_n, B_1, \ldots, B_p) &= \langle A_1, \ldots, A_n, B_1, \ldots, B_p \rangle \\ &\qquad [(Y_1', \ldots, Y_n', Z_1', \ldots, Z_p') \leftarrow (Y_1, \ldots, Y_n, Z_1, \ldots, Z_p)'] \\ &\qquad\qquad \backslash (Z_1, \ldots, Z_p). \end{aligned}$$

That is, the network N is obtained by tupling all the A_i and B_j, then feeding back each of the Y_i and Z_j outputs to the corresponding primed inputs, and finally internalizing all the Z_j.

By following through the definitions of the network algebra operations involved, it can be shown that the network $N(A_1, \ldots, A_n, B_1, \ldots, B_p)$ is isomorphic to the automaton

$$N = (E, X, Y, Z, Q', \mathrm{I}', T'),$$

where $I = \{1, 2, \ldots, n+p\}$, the state set $Q' = \prod_{i \in I}(E^* /\!\sim)$ is the product of $n + p$ copies of the monoid of finite traces $E^* /\!\sim$, the initial state $\mathrm{I}' = \langle \epsilon : i \in I \rangle$ consists of $n + p$ copies of the empty trace ϵ, and the transition relation T' is defined as follows:

- $e \in T'(\langle z_i : i \in I \rangle, \langle w_i : i \in I \rangle)$ iff for all i, the trace w_i is enabled for A in state q_{I}, and $w_i = z_i e$.

Define a state $\langle z_i : i \in I \rangle$ of N to be *reachable* if there exists some finite computation γ from the initial state such that $\langle z_i : i \in I \rangle = \mathrm{cod}(\gamma)$. It is easily proved, by induction on the length of a finite computation γ, that if $\langle z_i : i \in I \rangle$ is reachable, then $z_i = z_j$ for all $i, j \in I$. Define $h : (\prod_{i \in I}(E')^* /\!\sim) \to Q$ to be the partial function whose domain of definition is the set of reachable states, and is such that if $\langle z : i \in I \rangle$ is reachable, then $h(\langle z : i \in I \rangle) = \mathrm{I}z$, the unique state reached by automaton A by applying trace z to the initial state I. It is now obvious that h is a simulation of A by N, hence because $N \simeq N(A_1, \ldots, A_n, B_1, \ldots, B_p)$, we conclude that there exists a simulation of A by $N(A_1, \ldots, A_n, B_1, \ldots, B_p)$. ∎

Proposition 7.14 *Suppose A is a sequential automaton. Then A computes a sequential function.*

Proof – If A is a sequential automaton, then by Lemma 7.13 A is simulated by a network $N(A_1, \ldots, A_n, B_1, \ldots, B_p)$ of strictly sequential automata. By Lemma 7.11, each A_i computes a sequential function, so by Proposition 7.1, the network, hence A, computes a sequential function as well. ∎

8 Conclusions

In this paper we have examined the relationship between operational and mathematical characterizations of sequentiality and stability. We have defined easily identifiable properties of concurrent automata that correspond to sequentiality and to stability. The proof proceeds by an examination of the mathematical properties of the domains of computations that result. Though it was a well-known "folk theorem" that Kahn networks compute sequential functions the characterization of such networks that people had in mind was expressed informally in terms of an underlying language for Kahn processes. Roughly speaking, if communication with input chanels is restricted to be via "blocking reads" then the resulting process computes a sequential function. Our characterization is in terms of abstract automata and, more importantly, shows that all sequential functions are computed by the automata that we define. Our characterization of stable automata is the first such characterization that we have seen.

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] G. Berry. Stable models of typed lambda-calculi. In *Proceedings of the Fifth International Colloquium on Automata Languages And Programming*, pages 72–89, 1978. Springer-Verlag Lecture Notes In Computer Science 62.

[4] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.

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

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

[7] G. Kahn and G. Plotkin. *Structures de données concrètes.* Technical Report 336, IRIA-LABORIA, 1978.

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

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

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

[11] P. Panangaden and V. Shanbhogue. *On The Expressive Power of Indeterminate Primitives.* Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[12] 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.

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

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

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

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

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

[18] E. W. Stark. *On the Relations Computed by a Class of Concurrent Automata.* Technical Report 88-09, SUNY at Stony Brook Computer Science Dept., 1988.

[19] G. Winskel. Event structures. In *Advanced Course on Petri Nets*, GMD, Bad Honnef, September 1986.

[20] G. Winskel. *A Representation of Completely Distributive Algebraic Lattices.* Technical Report CMU-CS-83-154, Carnegie-Mellon University Dept. of Computer Science, 1983.