DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720

June 4, 1996

THE TAGGED SIGNAL MODEL

A PRELIMINARY VERSION OF A DENOTATIONAL FRAMEWORK FOR COMPARING MODELS OF COMPUTATION

Edward A. Lee and Alberto Sangiovanni-Vincentelli
EECS, University of California, Berkeley, CA, USA 94720.

Abstract

We give a denotational framework that describes concurrent processes in very general terms as sets of possible behaviors. Compositions of processes are given as intersections of their behaviors. The interaction between processes is through signals, which are collections of events. A system is determinate if given the constraints imposed by the inputs there are exactly one or exactly zero behaviors. Each event is a value-tag pair, where the tags can come from a partially ordered or totally ordered set. Timed models are where the set of tags is totally ordered. Synchronous events share the same tag, and synchronous signals contain events with the same set of tags. Synchronous systems contain synchronous signals. Strict causality (in timed systems) and continuity (in untimed systems) ensure determinacy under certain technical conditions. The framework is used to compare certain essential features of various models of computation, including Kahn process networks, dataflow, sequential processes, concurrent sequential processes with rendezvous, and discrete-event systems.

1. Introduction

A major impediment to further progress in heterogeneous modeling and specification of systems is the confusion that arises from different usage of common terms. Terms like “synchronous”, “discrete

event", "dataflow", and "process" are used in different communities to mean significantly different things. To address this problem, we propose a formalism that will enable description and differentiation of models of computation. To be sufficiently precise, this language is a mathematical one. It is *denotational,* in the sense of Scott and Strachey [23], rather than operational, to avoid associating the semantics of a model of computation with an execution policy. In many denotational semantics, the *denotation* of a program fragment is a partial function or a relation on the state. This approach does not model concurrency well [25], where the notion of a single global state may not be well-defined. In our approach, the denotation of a program fragment (called a process) is a partial function or a relation on signals.

We define precisely a process, signal, and event, and give a framework for identifying the essential properties of discrete-event systems, dataflow, rendezvous-based systems, and process networks. We give unambiguous definitions of timed systems and synchrony. These definitions sometimes conflict with common usage in some communities, and even with our own prior usage in certain cases. We have made every attempt to maintain the spirit of that usage with which we are familiar, but have discovered that terms are used in contradictory ways (sometimes even within a community). Maintaining consistency with all prior usage is impossible without going to the unacceptable extreme of abandoning the use of these terms altogether.

2. The tagged signal model

2.1 SIGNALS

Given a set of *values* V and a set of *tags* T, we define an event e to be a member of $T \times V$. I.e., an event has a tag and a value. We define a *signal* s to be a set of events. A signal can be viewed as a subset of $T \times V$, or as a member of the powerset $2^{(T \times V)}$. A *functional signal* or *proper signal* is a (possibly partial) function from T to V. By "partial function" we mean a function that may be defined

only for a subset of T. By "function" we mean that if $e_1 = (t, v_1) \quad s$ and $e_2 = (t, v_2) \quad s$, then $v_1 = v_2$. Unless otherwise stated, we assume all signals are functional. We call the set of all signals S, where of course $S = 2^{(T \times V)}$. It is often useful to form a collection or *tuple* $\mathbf{s}$ of N signals. The set of all such tuples will be denoted S^N.

The empty signal (one with no events) will be denoted by λ, and the tuple of empty signals by Λ, where the number N of empty signals in the tuple will be understood from the context. These are signals like any other, so $\lambda \quad S$ and $\Lambda \quad S^N$. For any signal s, $s \quad \lambda = s$, and for any tuple $\mathbf{s}$, $\mathbf{s} \quad \Lambda = \mathbf{s}$, where by the notation $\mathbf{s} \quad \Lambda$ we mean the pointwise union of the sets in the tuple.

In some models of computation, the set V of values includes a special value (called "bottom"), which indicates the absence of a value. Notice that while it might seem intuitive that $(t, \quad) \quad \lambda$ for any $t \quad T$, this would violate $s \quad \lambda = s$ (suppose that s already contains an event at t). Thus, it is important to view as an ordinary member of V like any other member.

2.2 TAGS

Frequently, a natural interpretation for the tags is that they mark time in a physical system. Neglecting relativistic effects, time is the same everywhere, so tagging events with the time at which they occur puts them in a certain order (if two events are genuinely simultaneous, then they have the same tag). For *specifying* systems, however, the global ordering of events in a timed system may be overly restrictive. A specification should not be constrained by one particular physical implementation, and therefore need not be based on the semantics of the physical world. Thus, for specification, often the tags *should not* mark time.

In a *model* of a physical system, by contrast, tagging the events with the time at which they occur

may seem natural. They must occur at a particular time, and if we accept that time is uniform, then our model should reflect the ensuing ordering of events. However, when modeling a large concurrent system, the model should probably reflect the inherent difficulty in maintaining a consistent view of time in a distributed system [6][13][18][22]. If an implementation cannot maintain a consistent view of time, then it may be inappropriate for its model to do so (it depends on what questions the model is expected to answer).

Fortunately, there are a rich set of untimed models of computation. In these models, the tags are more abstract objects, often bearing only a partial ordering relationship among themselves.

2.3 PROCESSES

In the most general form, a *process* P is a subset of S^N for some N. A particular $\mathbf{s} \quad S^N$ is said to *satisfy* the process if $\mathbf{s} \quad P$. An $\mathbf{s}$ that satisfies a process is called a *behavior* of the process. Thus a *process* is a set of possible *behaviors* or a *relation* between signals.

Intuitively, N should be the number of signals *associated* with the process, affecting it, being affected by it, or both. However, it is often convenient to make N much larger, perhaps large enough to include all signals in a system. Consider for example the two processes in figure 1. There, we can

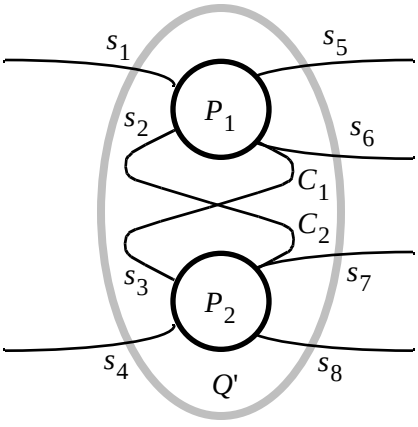

FIGURE 1. An interconnection of processes.

define the processes as subsets of S^8.

A *connection* $C \subset S^N$ is a particularly simple process where two of the signals in the N-tuple are constrained to be identical. For example, in figure 1, $C_1 \subset S^8$ where

$$\mathbf{s} = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8) \in C_1 \text{ iff } s_3 = s_6 . \tag{1}$$

There is nothing special about connections as processes, but they are sufficiently useful that we highlight them.

Figure 1 shows a system. A *system* Q with N signals and M processes (some of which may be connections) is given by

$$Q = \bigcap_{P_i \in \mathbf{P}} P_i , \tag{2}$$

where $\mathbf{P}$ is the collection of processes $P_i \subset S^N$, $1 \le i \le M$. For example, in figure 1, the overall system may be given as $P_1 \cap P_2 \cap C_1 \cap C_2$. That is, any $\mathbf{s} \in S^8$ that satisfies the overall system must satisfy each of P_1, P_2, C_1, and C_2.

Of course, a system is itself a process, making the two terms interchangeable. We will generally use the word “process” to describe a part of a larger system, and “system” to describe an aggregation of all processes and connections under consideration.

As suggested by the gray outline in figure 1, it makes little sense to expose all the signals of a system as signals associated with the system. In figure 1, for example, since signals s_2 and s_3 are identical to s_7 and s_6 respectively, it would make more sense to “hide” two of these signals.

Given a process $P \subset S^N$, the *projection* $\Pi_j(P) \subset S^{N-1}$ is defined by

$$(s_1, ..., s_{j-1}, s_{j+1}, ..., s_N) \quad \Pi_j(P)$$

iff there exists $s_j \quad S$ such that

$$(s_1, ..., s_{j-1}, s_j, s_{j+1}, ..., s_N) \quad P. \tag{3}$$

Thus, in figure 1, we can define the composite process $Q' = \Pi_2(\Pi_3(Q)) \quad S^6$. This projection operator removes one element at a time. It is sometimes useful to use a projection operator that leaves only one element. The projection $\pi_j(P) \quad S$ is defined by

$$s \quad \pi_j(P) \text{ iff there exist } s_1, ..., s_{j-1}, s_{j+1}, ..., s_N \quad S \text{ such that}$$

$$(s_1, ..., s_{j-1}, s, s_{j+1}, ..., s_N) \quad P. \tag{4}$$

If the two signals in a connection are associated with the same process, as shown in figure 2, then the connection is called a *self-loop*. For the example in figure 2, $Q' = \Pi_1(P \cap C)$. For simplicity, we will often denote self-loops with only a single signal.

Many systems have the notion of inputs, which are events or signals that are defined outside the model. Formally, an *input* to a process is an externally imposed constraint $I \quad S^N$ such that $I \cap P$ is the total set of acceptable behaviors. The set of all possible inputs $B \quad 2^{S^N}$ is a further characterization

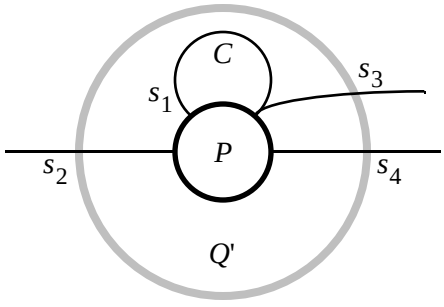

FIGURE 2. A self loop.

of a process. For example, $B = \{I; I \in S^N, \pi_1(I) = s, s \in S\}$ means that the first signal is specified externally and can take on any value in the set of signals. But inputs could also be events within signals, in general.

A system or process is *determinate* if given all inputs it has exactly one behavior or exactly no behaviors; i.e. $|I \cap P| = 1$ or $|I \cap P| = 0$ for each $I \in B$. Otherwise, it is *nondeterminate*. Thus, whether a process is determinate or not depends on our characterization B of the inputs. Fortunately, most interesting cases distinguish input and output *signals*, making the characterization B conceptually simple.

For many (but not all) applications, it is natural to partition the signals associated with a process into *input signals* and *output signals*. Intuitively, the process does not determine the values of the inputs, and does determine the values of the outputs. If $N = m + n$, then (S^m, S^n) is a *partition* of S^N. A *process* P with m *inputs* and n *outputs* is a subset of $S^m \times S^n$. In other words, a process defines a *relation* between input signals and output signals. An $m + n$ tuple $\mathbf{s} \in S^{m+n}$ is said to *satisfy* P if $\mathbf{s} \in P$. It can be written $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2)$, where $\mathbf{s}_1 \in S^m$ is an m-tuple of *input signals for process* P and $\mathbf{s}_2 \in S^n$ is an n-tuple of *output signals for process* P. If the input signals are given by $\mathbf{a} \in S^m$, then the set $I = \{(\mathbf{a}, \mathbf{b}) | \mathbf{b} \in S^n\}$ describes the inputs, and $I \cap P$ is the set of behaviors consistent with the input $\mathbf{a}$.

So far, however, this partition does not capture the notion of a process "determining" the values of the outputs. A process F is *functional* with respect to a partition if it is a single-valued mapping from

S^m or some subset of S^m to S^n. That is, if $(\mathbf{s}_1, \mathbf{s}_2) \quad F$ and $(\mathbf{s}_1, \mathbf{s}_3) \quad F$, then $\mathbf{s}_2 = \mathbf{s}_3$. In this case, we can write $\mathbf{s}_2 = F(\mathbf{s}_1)$, where $F : S^m \rightarrow S^n$ is a (possibly partial) function. Such a process is obviously determinate for an appropriate input characterization B. Given the input signals, the output signals are determined (or there is unambiguously no behavior). Formally, given a partition (S^m, S^n) and a process F that is functional with respect to this partition, the process is determinate for input characterization $B = \{\{(\boldsymbol{p}, \boldsymbol{q}); \boldsymbol{q} \quad S^n\}; \boldsymbol{p} \quad S^n\}$. We will mostly use the symbol F to denote functional processes.

Consider possible partitions for the example in figure 1. Suppose that s_5 and s_6 are outputs of P_1 and s_1 and s_2 are inputs. This is suggested by the arrowheads in figure 3. If P_1 is functional with respect to the partition $((s_1, s_2, s_3, s_4, s_7, s_8), (s_5, s_6))$, then we will denote the process and its function as F_1 rather than P_1. Notice that the irrelevant signals fall in the input partition, since they cannot logically be functions of other signals, as far as P_1 is concerned.

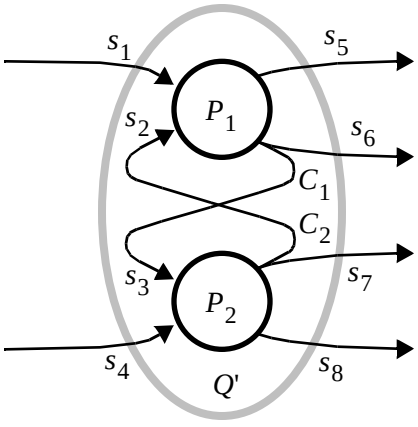

FIGURE 3. A partitioning of the signals in figure 1 into inputs and outputs.

Note that a given process may be functional with respect to more than one partition. A connection, for example, is a process relating two signals, say s_1 and s_2, and it is functional with respect to either $((s_1), (s_2))$ or $((s_2), (s_1))$.

A system $Q \quad S^N$ is said to be *closed* if it is functional with respect to the partition $(\quad , S^N)$. It is *open* if it is not closed.

Given a process P and a partition (S^m, S^n), P is *total with respect to this partition* if for every $\mathbf{s}_1 \quad S^m$ there is an $\mathbf{s}_2 \quad S^n$ such that $(\mathbf{s}_1, \mathbf{s}_2) \quad P$. The signal tuple $\mathbf{s}_1$ is said to be *accepted by process* P. Many (if not most) useful processes are determinate and total. We henceforth assume that all functional processes are total.

3. Partially and totally ordered tags

A *partially ordered tagged system* is a system where the set T of tags is a partially ordered set. *Partially ordered* means that there exists an irreflexive, antisymmetric, transitive relation between members of the set [24]. We denote this relation using the template "<". Of course, we can define a related relation, denoted "≤", where $t_1 \le t_2$ if $t_1 = t_2$ or $t_1 < t_2$.

The ordering of the tags induces an ordering of events as well. Given two events $e_1 = (t_1, v_1)$ and $e_2 = (t_2, v_2)$, $e_1 < e_2$ if and only if $t_1 < t_2$.

We are not alone in choosing to use partial orders to model concurrent systems. Pratt gives an excellent motivation for doing so, and then generalizes the notion of formal string languages to allow partial ordering rather than just total ordering [20]. Mazurkiewicz uses partial orders in developing an algebra of concurrent "objects" associated with "events" [17]. Partial orders have also been used to

analyze Petri nets [21]. Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient [13]. He gives a mechanism in which messages in an asynchronous system carry time stamps and processes manipulate these time stamps. We can then talk about processes having information or knowledge at a *consistent cut*, rather than "simultaneously". Fidge gives a related mechanism in which processes that can fork and join increment a counter on each event [7]. A partial ordering relationship between these lists of times is determined by process creation, destruction, and communication. If the number of processes is fixed ahead of time, then Mattern gives a more efficient implementation by using "vector time" [16]. Unlike the work of Lamport, Fidge, and Mattern, we are not using partial orders in the implementation of systems, but rather are using them as an analytical tool to study models of computation and their interaction semantics. Thus, efficiency of implementation is not an issue.

4. Timed concurrent systems

A *timed system* is a tagged system where T is totally ordered. That is, for any distinct t_1 and t_2 in T, either $t_1 < t_2$ or $t_2 < t_1$. The use of the term "timed" here stems from the observation that in the standard model of the physical world, time is viewed as globally ordering events. Any two events are either simultaneous (have the same tag), or one unambiguously precedes the other.

4.1 METRIC TIME

Some timed models of computation include operations on tags. At a minimum, T is an Abelian group, in addition to being totally ordered. This means that there is an operation $+: T \times T \rightarrow T$, called addition, under which T is closed. Moreover, there is an element, called zero and denoted "0", such that $t + 0 = t$ for all $t \quad T$. Finally, for every element $t \quad T$, there is another element $-t \quad T$ such that $t + (-t) = 0$. A consequence is that $t_2 - t_1$ is itself a tag for any t_1 and t_2 in T.

In a slightly more elaborate model of computation, T has a *metric*, which is a function $f: T \times T \rightarrow$, where is the set of real numbers, that satisfies the following conditions:

$$f(t_1, t_2) = f(t_2, t_1) \text{ for all } t_1, t_2 \quad T, \tag{5}$$

$$f(t_1, t_2) = 0 \Leftrightarrow t_1 = t_2, \tag{6}$$

$$f(t_1, t_2) \geq 0 \text{ for all } t_1, t_2 \quad T, \text{ and} \tag{7}$$

$$f(t_1, t_2) + f(t_2, t_3) \geq f(t_1, t_3) \text{ for all } t_1, t_2, t_3 \quad T. \tag{8}$$

Such systems are said to have *metric time*. In a typical example of metric time, T is the set of real numbers and $f(t_1, t_2) = |t_1 - t_2|$, the absolute value of the difference.

4.2 CONTINUOUS TIME

Let $T(s)$ T denote the set of tags in a signal s. A *continuous-time system* is a metric timed system Q where T is a continuum (a closed connected set) and $T(s) = T$ for each signal s in any tuple $\mathbf{s}$ that satisfies the system. A *connected set* is one where no matter how it is divided into two disjoint sets, at least one of these contains limit points of the other. A *closed set* is one that contains the limit points of any subset. Limit points, of course, are defined in the usual way using the metric.

4.3 DISCRETE-EVENT SYSTEMS

Many simulators are also based on a discrete-event model, including most digital circuit simulators (see for example [8]). Given a system Q, and a tuple of signals $\mathbf{s}$ Q that satisfies the system, let $T(\mathbf{s})$ denote the set of tags appearing in any signal in the tuple $\mathbf{s}$. Clearly $T(\mathbf{s})$ T and the ordering relationship for members of T induces an ordering relationship for members of $T(\mathbf{s})$. A *discrete-event system* Q is a timed system where for all $\mathbf{s}$ Q, the ordering relationship for $T(\mathbf{s})$ is *discrete.* Intuitively, this means that any pair of ordered tags either has a finite number of intervening tags or one of

the tags is maximal (has no tags above it). We explain this now precisely.

Let R denote the set of pairs (t_1, t_2) of tags such that $t_1 < t_2$. In fact, R is an alternative notation to "<" for the ordering itself. Let R^2 denote another relation where (t_1, t_2) R^2 if and only if $t_1 < t_2$ and there exists a τ T such that $t_1 < \tau < t_2$. More generally, let R^n denote a relation where (t_1, t_2) R^n if and only if $t_1 < t_2$ and there exist $\tau_1, \tau_2, ..., \tau_{n-1}$ T such that $t_1 < \tau_1 < \tau_2 < ... < \tau_{n-1} < t_2$. Finally, let R^* denote the transitive closure of the relation R (the relation R applied an arbitrary number of times). The relation R is then said to be *discrete* if $R = (R - R^2)^*$ [17]. This can be understood intuitively if we realize that the set $R - R^2$ is the set of all pairs (t_1, t_2) of tags such that $t_1 < t_2$ where there is no τ T such that $t_1 < \tau < t_2$; i.e., it is the set of ordered tags with no intervening tags. For a non-discrete ordering, this set might be empty. For example, if T is the set of real numbers and R is the usual ordering of real numbers, $R - R^2 =$. It follows similarly when R is the set of rationals. Moreover, this definition of a discrete order precludes Zeno-like situations, where an infinite sequence of events (e.g. events that occur closer and closer together, converging on a particular time) are followed by other events in the order.

Note that this rather technical definition of discrete-event systems appears to be necessary to capture all common cases and to correspond well with intuition. For example, it is common for discrete-event systems to take T to be the set of real numbers, so it would not do to restrict T to being countable. It also is not sufficient to declare that $T(\mathbf{s})$ is countable for all $\mathbf{s}$ Q. If for a particular tuple $\mathbf{s}$ Q of signals satisfying the system the tags $T(\mathbf{s})$ are the set of rational numbers, for example, this would not correspond well with our intuition about what constitutes a discrete-event system.

In some communities, notably the control systems community, a discrete-event model also requires that the set of *values* V be countable, or even finite [4][10]. This helps to keep the state space finite in certain circumstances, which can be a big help in formal analysis. However, in the simulation community, it is largely irrelevant whether V is countable [8]. In simulation, the distinction is technically moot, since all representations of values in a computer simulation are drawn from a finite set. We adopt the broader use of the term, and will refer to a system as a discrete-event system whether V is countable, finite, or neither.

4.4 SYNCHRONOUS SYSTEMS

Two events are *synchronous* if they have the same tag. Two signals are synchronous if all events in one signal are synchronous with an event in the other signal and vice versa. A system is synchronous if every signal in the system is synchronous with every other signal in the system. A *discrete-time system* is a synchronous discrete-event system.

By this definition, the so-called Synchronous Dataflow (SDF) model of computation [14] is not synchronous (we will say more about dataflow models below). The "synchronous languages" [1] (such as Lustre, Esterel, and Argos) are synchronous if we consider V, where (bottom) denotes the absence of an event. Indeed, a key property of synchronous languages is that the absence of an event at a particular "tick" (tag) is well-defined. Another key property is that event tags are totally ordered. Any two events either have the same tag or one unambiguously precedes the other. The language Signal [2] is called a synchronous language, but in general, it is not even timed. It supports nondeterminate operations which require a partially ordered tag model. *Cycle-based* circuit simulators are discrete-time systems.

4.5 CAUSALITY

We begin with a timed notion of causality, momentarily restricting our attention to timed systems.

Borrowing notation from Yates [26], a signal $s' = s|^t$ is defined to be the subset of events in s with tags less than or equal to tag t. This is called a *cut* of s. This generalizes to tuples $\mathbf{s}$ of signals or sets P of tuples of signals, where $\mathbf{s}|^t$ and $P|^t$ are tuples and sets of tuples of cut signals, respectively. A functional process F is *causal* iff

$$\mathbf{s}_2 = F(\mathbf{s}_1) \quad \mathbf{s}_2|^t = F(\mathbf{s}_1|^t)|^t \text{ for all } t \quad T. \tag{9}$$

Yates [26] considers timed systems with metric time where $\Delta > 0$ is a tag[1]; a functional process F is Δ-*causal* if $F|^\Delta = \{\Lambda\}$, the tuple of empty signals, and for all $t \quad T, t > \Delta$,

$$\mathbf{s}_2 = F(\mathbf{s}_1) \quad \mathbf{s}_2|^t = F(\mathbf{s}_1|^{t-\Delta})|^t. \tag{10}$$

Intuitively, Δ-causal means that the process incurs a *time delay* of Δ. Yates proves that every network of Δ-causal functional processes is determinate.

We can define a similar notion for non-metric timed systems if they are also discrete-event systems. We say that tag t_1 *immediately precedes* tag t_2 in signal s if $t_1 < t_2$ and there is no $\tau \quad T(s)$ such that $t_1 < \tau < t_2$; i.e., there are no intervening tags in the signal. A non-metric timed discrete-event functional process F is *strictly causal* iff

$$\mathbf{s}_2 = F(\mathbf{s}_1) \quad \mathbf{s}_2|^{t_2} = F\left(\mathbf{s}_1|^{t_1}\right)|^{t_2} \text{ for all } t_2 \quad T \text{ and } t_1 \text{ immediately preceding } t_2. \tag{11}$$

Strictly causal processes are said to introduce *delay* as well, even though it is hard to interpret this delay as a time delay without metric time.

1. The zero element exists in T because T is an Abelian group.

5. Untimed Concurrent Systems

When tags are partially ordered rather than totally ordered, we say that the system is untimed. Untimed systems cannot have the same notion of causality as timed systems. The equivalent intuition is provided by the monotonicity condition. A slightly stronger condition, continuity, will be sufficient to ensure determinacy. These two conditions depend on a partial ordering of signals called the prefix order.

5.1 THE PREFIX ORDER FOR SIGNALS

A signal is a set of events. Set inclusion, therefore, provides a natural partial order for signals (vs. tags). Instead of the symbol "< " that we used for the ordering of tags, we use the symbol " " for an ordering based on set inclusion. This is an irreflexive antisymmetric transitive binary relation. The reflexive version, of course, is " ". Thus, for two signals s_1 and s_2, $s_1 \quad s_2$ if every event in s_1 is also in s_2.

In many of the models of computation that we will consider, the tags in each signal are totally ordered by "< " even if the complete set T of tags is only partially ordered. In this case, another natural partial ordering for signals emerges; it is called the *prefix order*. For the prefix order, we write $s_1 \sqsubseteq s_2$ if every event in s_1 is also in s_2, and each event in s_2 that is not in s_1 has a tag greater than all tags in s_1. More formally, if both $T(s_1)$ and $T(s_2)$ are totally ordered,

$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1 \quad s_2 \text{ and for all } e_1 \quad s_1 \text{ and } e_2 \quad s_2 - s_1,\ e_2 > e_1, \tag{12}$$

where $s_2 - s_1$ denotes the set of events in s_2 that are not also in s_1. Clearly, in our model, the empty signal λ is a prefix of every other signal, so it too is called *bottom*.

In partially ordered models for signals, it is often useful for mathematical reasons to ensure that the partial order is a *complete partial order* (*cpo*). To explain this fully, we need some more definitions.

An *increasing chain* in S is a set $\{s_t; t \in U\}$, where $U \subseteq T$ is a totally ordered subset of T and for any t_1 and t_2 in U,

$$s_{t_1} \sqsubseteq s_{t_2} \Leftrightarrow t_1 \leq t_2 . \tag{13}$$

An *upper bound* of a subset $W \subseteq S$ is an element $w \in S$ where every element in W is a prefix of w. A *least upper bound* (*lub*) $\sqcup W$ is an upper bound that is a prefix of every other upper bound. A *complete partial order* (*cpo*) is a partial order where every increasing chain has a lub. From a practical perspective, this often implies that our set S of signals must include signals with an infinite number of events.

These definitions are easy to generalize to S^N. For $\mathbf{s}_1 \in S^N$ and $\mathbf{s}_2 \in S^N$, $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2$ iff each corresponding element is a prefix, i.e. $s_{1,i} \sqsubseteq s_{2,i}$ for each $1 \leq i \leq N$. With this definition, if S is a cpo, so is S^N. We will assume henceforth that S^N is a cpo for all N.

We can now introduce the untimed equivalent of causality.

5.2 MONOTONICITY AND CONTINUITY

We now generalize to untimed systems, connecting to well-known results originally due to Kahn [12]. Our contribution here is only to present these results using our notation. A process F is *monotonic* iff it is functional, and

$$\mathbf{s} \sqsubseteq \mathbf{s}' \qquad F(\mathbf{s}) \sqsubseteq F(\mathbf{s}') . \tag{14}$$

A process $F: S^m \rightarrow S^n$ is said to be *continuous* if it is functional and for every increasing chain $W \subseteq S^m$, $F(W)$ has a least upper bound $\sqcup F(W)$, and

$$F(\sqcup W) = \sqcup F(W) . \tag{15}$$

The notation $F(W)$ denotes a set obtained by applying the function F to each element of W. The term

"continuous" is consistent with the usual mathematical definition of continuity. For intuition, it may help some readers to connect the definition to that of continuous functions of real variables. This is easy if $\sqcup$ is interpreted as a *limit* of the increasing chain.

Fact: A continuous process is monotonic [12].

Proof: Suppose $F: S^m \rightarrow S^n$ is continuous and consider two signals $\mathbf{s}_1$ and $\mathbf{s}_2$ in S^m where $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2$. Define the increasing chain $W = \{\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_2, \mathbf{s}_2, \quad\}$. Then $\sqcup W = \mathbf{s}_2$, so from continuity,

$$F(\mathbf{s}_2) = F(\sqcup W) = \sqcup F(W) = \sqcup \{F(\mathbf{s}_1), F(\mathbf{s}_2)\} . \tag{16}$$

Therefore $F(\mathbf{s}_1) \sqsubseteq F(\mathbf{s}_2)$, so the process is monotonic.

Consider a composition Q of continuous processes $F_1, F_2, ..., F_M$. Assume $Q \quad S^N$ for some N. In general, the composition may not be determinate. Consider a trivial case, where $M = 1$ and $F_1: S \rightarrow S$ is the identity function. This function is certainly continuous. Suppose we construct a closed system Q by composing F_1 with a single connection, as shown in figure 4. Then any signal $s \quad S$ satisfies Q. Since there are no inputs to this process and it has many behaviors, it is not determinate.

We will now show that there is an alternative interpretation of the composition Q that is functional, and in fact is also continuous. Under this interpretation, any composition of continuous processes is determinate. Moreover, this interpretation is consistent with execution policies typically used

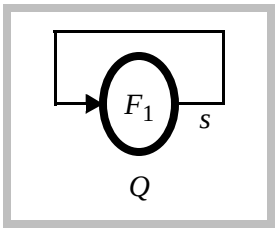

FIGURE 4. A simple composition of continuous processes that is not determinate under a general interpretation, but is determinate under a least-fixed-point semantics. F_1 is the identity function.

for such systems (their operational semantics), and hence is an entirely reasonable denotational semantics for the composition. This interpretation is called the *least-fixed-point* semantics.

Consider again a composition Q of functional and continuous processes $F_1, F_2, ..., F_M$. Each process F_i is a mapping from S^m to S^n for some m and n. Since the m and n are in general different for each process, a notation for the composition of functions can get complicated. Indeed, in standard presentations, it does get complicated, using for example notation from the lambda calculus. Here, we simplify the notation by observing that the function $F_i: S^m \rightarrow S^n$ can be described instead as a function $F_i': S^N \rightarrow S^p$, where N is the total number of signals in Q and $p \geq n$. Any signals in the output of the function F_i' that are not properly part of the output of F_i are simply copied from the input, which includes all signals. Each function F_i' is continuous because F_i is continuous. The process Q can now be illustrated as in figure 5, where q is the total number of signals in Q that are not outputs of one of $F_1, F_2, ..., F_M$, and $p = N - q$.

We can now construct a continuous function $F: S^q \rightarrow S^p$ that describes Q. First, let **q** denote the

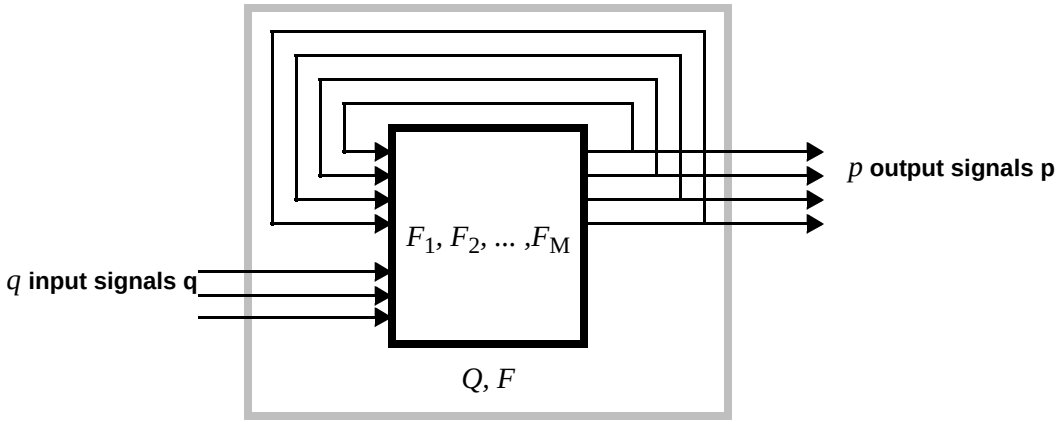

FIGURE 5. A composition of continuous functional processes is functional and continuous.

tuple of q input signals, and let $\mathbf{p}$ denote the remaining p signals. The define $G: S^N \rightarrow S^p$ as

$$G(\mathbf{q}, \mathbf{p}) = F_1'(\mathbf{q}, F_2'(\mathbf{q}, \quad F_M'(\mathbf{q}, \mathbf{p}) \quad)) . \tag{17}$$

It turns out that the order in which we apply the individual function F_i' does not matter, although we do not show that here. We are looking for a *fixed point* $\mathbf{p}$ of this function,

$$G(\mathbf{q}, \mathbf{p}) = \mathbf{p} . \tag{18}$$

To find it, let $\mathbf{p}_i \quad S^p$, $i = 1, 2, \quad$, be defined by

$$\begin{aligned} \mathbf{p}_1 &= G(\mathbf{q}, \Lambda) \\ \mathbf{p}_2 &= G(\mathbf{q}, \mathbf{p}_1) \\ &... \\ \mathbf{p}_n &= G(\mathbf{q}, \mathbf{p}_{n-1}) \\ &... \end{aligned} \tag{19}$$

where Λ is the tuple of empty signals. Notice that since $(\mathbf{q}, \Lambda) \sqsubseteq (\mathbf{q}, \mathbf{p}_1)$ and G is monotonic (because it is continuous), $\mathbf{p}_1 \sqsubseteq \mathbf{p}_2$. By induction, the sequence $\{\mathbf{p}_1, \mathbf{p}_2, ...\}$ is an increasing chain. Since S^p is a cpo, this increasing chain has a least upper bound,

$$\mathbf{p} = \sqcup \{\mathbf{p}_1, \mathbf{p}_2, ...\} . \tag{20}$$

Moreover, since G is continuous,

$$G(\mathbf{q}, \mathbf{p}) = G(\mathbf{q}, \sqcup \{\mathbf{p}_1, \mathbf{p}_2, ...\}) = \sqcup G(\mathbf{q}, \{\mathbf{p}_1, \mathbf{p}_2, ...\}) = \sqcup \{\mathbf{p}_2, \mathbf{p}_3, ...\} = \mathbf{p} . \tag{21}$$

Hence, the least upper bound $\mathbf{p}$ is a fixed point of G, so $\mathbf{s} = (\mathbf{q}, \mathbf{p})$ is a behavior of the composite process Q. Since it is a behavior of the composite process, then it is also a behavior of the individual processes F_i.

Moreover, $\mathbf{p}$ as defined in (20) is the unique *least fixed point* of G, meaning that it is a prefix of

any fixed point. To see this, consider any fixed point $\mathbf{p}'$. Certainly, $\Lambda \sqsubseteq \mathbf{p}'$, so $G(\mathbf{q}, \Lambda) \sqsubseteq G(\mathbf{q}, \mathbf{p}')$ (by monotonicity). Since $\mathbf{p}'$ is a fixed point, $G(\mathbf{q}, \Lambda) \sqsubseteq \mathbf{p}'$. By induction, $G(\mathbf{q}, \mathbf{p}_n) = \mathbf{p}_n \sqsubseteq \mathbf{p}'$ for all $n = 1, 2, \ldots$. Hence, $\mathbf{p} \sqsubseteq \mathbf{p}'$, where $\mathbf{p}$ is given by (20). We have just proven a classic cpo fixed point theorem.

Note that $\mathbf{p}$ as given by (20) is a uniquely defined function of $\mathbf{q}$. Let us call this function F. If we define this function to be the *semantics* (the meaning) of the composition Q, then the composition is functional, and hence determinate. It is easy to show that it is a continuous function of $\mathbf{q}$ as well.

Under this least-fixed-point semantics, the value of s is figure 4 is λ, the empty signal. Under this semantics, this is the only signal that satisfies the composite process, so the composite process is determinate. Intuitively, this solution agrees with a reasonable execution of the process, in which we would not produce any output from F_1 because there are no inputs. This reasonable operational semantics therefore agrees with the denotational semantics. For a complete treatment of this agreement, see Winskel [25].

Notice that the existence of multiple fixed points implies that for a given input constraint $I \quad S^N$, the set $Q \cap I$ of signal tuples that satisfy the system has size greater than one, implying nondeterminism. We are getting around this nondeterminism by defining the single unique signal tuple that satisfies the system to be $\min(Q \cap I)$, the smallest member (in a prefix order sense) of the set $Q \cap I$, as long as there is at least one behavior in $Q \cap I$. This minimum exists and is in fact equal to the least fixed point, as long as the composing processes are continuous (every member of $Q \cap I$ is a fixed point, and we have shown that there is a unique least fixed point).

For the example in figure 4, $Q = S$ (any signal seems to satisfy the process, for Q defined as in

equation (2)), and $I = S$ (there are no inputs, so the inputs impose no constraints). Thus $Q \cap I = S$. The least fixed-point semantics dictates that we take the behavior to be $\min(Q \cap I) = \min(S) = \lambda$, the empty signal.

6. Models of Computation

A variety of models have been proposed for concurrent systems where actions, communications, or both are partially ordered rather than totally ordered.

6.1 KAHN PROCESS NETWORKS

Let $T(s)$ denote the tags in signal s. In a *Kahn process network*, $T(s)$ is totally ordered for each signal s, but the set of all tags T may be partially ordered. In particular, for any two distinct signals s_1 and s_2, it could be that $T(s_1) \cap T(s_2) =$. Processes in Kahn process networks are also constrained to be continuous, and least-fixed-point semantics are used so that compositions of processes are determinate.

For example, consider a simple process that produces one output event for each input event. Denote the input signal $s_1 = \{e_{1,i}\}$, where $e_{1,i} < e_{1,j}$ if the index $i < j$. Let the output be $s_2 = \{e_{2,i}\}$. Then the process imposes the ordering constraint that $e_{1,i} < e_{2,i}$ for all i.

Lamport [13] also considers a model for distributed systems that is similar to this one where events occur inside processes, instead of in signals. The set of events inside a process is totally ordered, thus giving the process a sequential nature. Partial ordering constraints exist between events in different processes, thus modeling communication. This perspective is only slightly different from our model for Kahn processes, where partial ordering constraints between events are imposed by the causal behavior of processes, rather than the communication between them. The notion of sequential events defining a process, however, is useful when specializing Kahn process networks to dataflow.

6.2 SEQUENTIAL PROCESSES

A sequential process can be modeled by associating a single signal with the process, as suggested in figure 6(a), where the events $T(s)$ in the signal s are totally ordered. The sequential actions in the process (such as state changes) are represented by events on the signal.

6.3 RENDEZVOUS OF SEQUENTIAL PROCESSES

The CSP model of Hoare [11] and the CCS model of Milner [19] involve sequential processes that communicate via rendezvous. Similar models are realized in the languages Occam and Lotos. This idea is depicted in figure 6(b). In this case $T(s_i)$ is totally ordered for each $i = 1, 2, 3$. Moreover, representing each rendezvous point there will be events e_1, e_2, and e_3 in signals s_1, s_2, and s_3 respectively, such that

$$T(e_1) = T(e_2) = T(e_3), \tag{22}$$

where $T(e_i)$ is the tag of the event e_i.

Note that although the literature often refers to CSP and CCS as synchronous models of computation, under our definition they are not synchronous. They are not even timed. The events in s_1 and s_2 that are not associated with rendezvous points have only a partial ordering relationship with each other. This partial ordering becomes particularly important when there are more than two processes. Moreover, if a process can reach a state where it will rendezvous with one of several other processes, the

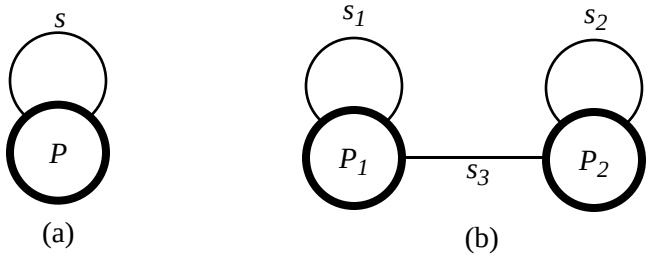

FIGURE 6. A sequential process (a) and communicating sequential processes (b).

composition is nondeterminate because of this partial order.

The Ada language also has rendezvous, although the implementation is bit different. In Ada, remote procedure calls (RPC) are used to communicate. This could be modeled using figure 6(b) as follows. A procedure in P_2 corresponds to a sequence of events $e_{2,1}$, $e_{2,2}$, ..., $e_{2,n}$ in signal s_2. These events would be constrained to lie between a calling event $e_{3,1}$ in signal s_3 and a return event $e_{3,2}$ in s_3. These latter two events would be constrained to lie between two events in s_1. Ada also supports nondeterminism in its rendezvous mechanism, in the form of "select" statements. By issuing a *select* statement, process P_2 can rendezvous with any of several RPC calls. It is easy to see how partial ordering constraints on events can adequately model this style of nondeterminism.

6.4 DATAFLOW

The dataflow model of computation is a special case of Kahn process networks [15]. A *dataflow process* is a Kahn process that is also sequential, where the events on the self-loop signal denote the *firings* of the dataflow actor. The *firing rules* of a dataflow actor are partial ordering constraints between these events and events on the inputs. A *dataflow process network*, is a network of such processes.

For example, consider a dataflow process P with one input signal and one output signal that *consumes* one input event and *produces* one output event on each firing, as shown in figure 7. Denote the input signal by $s_1 = \{e_{1,i}\}$, where $e_{1,i} < e_{1,j}$ if the index $i < j$. The firings are denoted by the signal

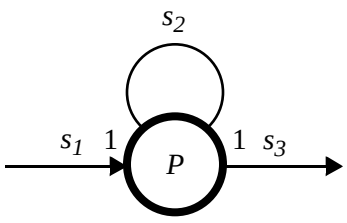

FIGURE 7. A simple dataflow process that consumes and produces a single token on each firing.

$s_2 = \{e_{2,i}\}$, and the output by $s_3 = \{e_{3,i}\}$, which will be similarly ordered. Then the inputs and outputs are related to the firings as $e_{1,i} < e_{2,i} < e_{3,i}$. A network of such processes will establish a partial ordering relationship between the firings of the actors.

More interesting examples of dataflow actors can also be modeled. The so-called *switch* and *select* actors, for example, are shown in figure 8. Each of them takes a Boolean-valued input signal (the bottom signal) and uses the value of the Boolean to determine the routing of *tokens* (events). The switch takes a single token at its left input s_1 and routes it the top right output s_3 if the Boolean in s_2 is true. Otherwise, it routes the token to the bottom right output s_4.

The partial ordering relationships imposed by the switch and select are inherently more complicated than those imposed by the simple dataflow actor in figure 7. But they can be fully characterized nonetheless. Suppose the control signal in the switch is given by $s_2 = \{(t_{2,i}, v_{2,i})\}$, where the index $i = 1$ denotes the first event on s_2, $i = 2$ the second, etc. Suppose moreover that the Booleans are encoded so that $v_{2,i} \in \{0, 1\}$. Let

$$b_k = \sum_{i=1}^{k} v_{2,i} \text{ for } k > 0. \tag{23}$$

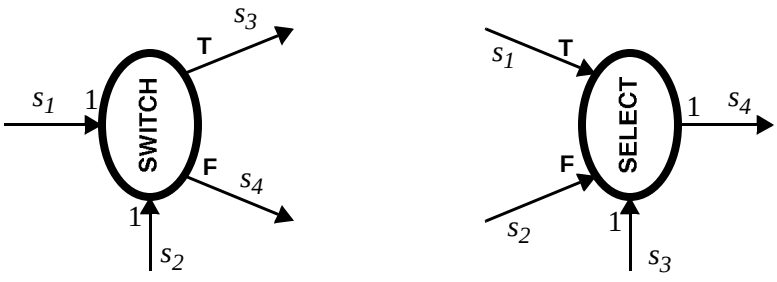

FIGURE 8. More complicated dataflow actors.

Denote the input signal by $s_1 = \{e_{1,i}\}$ and the output signals by $s_3 = \{e_{3,k}\}$ and $s_4 = \{e_{4,m}\}$ for indexes $i > 0$, $k > 0$ and $m > 0$. Then the ordering constraints imposed by the actor are

$$e_{3,k} > e_{1,b_k} \tag{24}$$

$$e_{4,m} > e_{1,(m-b_m)} \ . \tag{25}$$

The firings of the actor (not shown explicitly) would lie between these two input/output events.

6.5 DISCRETE-EVENT SIMULATORS

In a typical discrete-event simulator, sequential processes are interconnected with signals that contain events that explicitly include time stamps. These are the only types of systems we have discussed where the tags are explicit in the implementation. Each sequential process consists of a sequence of firings, as in dataflow, but unlike dataflow, events are globally ordered, so the firings are globally ordered. Indeed, the operational semantics of a discrete-event system is to execute the firings sequentially in time as follows. Find the event on the event queue with the smallest tag. Find the process for which the signal that contains this event is an input. Fire the process, and remove the event from the event queue. When events are produced in a firing, place them in the event queue sorted by tag. This operational semantics is completely consistent with our denotational semantics.

In some discrete-event simulators, such as VHDL simulators, tags contain both a time value and a "delta time." Delta time has the interpretation of zero time in the simulation. But it is used to avoid the ambiguity of having events with exactly the same tag, which could result in nondeterminism. In the denotational and operational semantics, the time value and delta time together determine the ordering of tags.

7. Heterogeneous Systems

It is assumed above that when defining a system, the sets T and V include all possible tags and

values. In some applications, it may be more convenient to partition these sets and to consider the partitions separately. For instance, V might be naturally divided into subsets V_1, V_2, ... according to a standard notion of *data types*. Similarly, T might be divided, for example to separately model parts of a heterogeneous system that includes continuous-time, discrete-event, and dataflow subsystems. This suggests a type system that focuses on signals rather than values. Of course, processes themselves can then also be divided by types, yielding a *process-level type system* that captures the semantic model of the signals that satisfy the process.

8. Conclusions

We have given the beginnings of a framework within which certain properties of models of computation can be understood and compared. Of course, any model of computation will have important properties that are not captured by this framework. The intent is not to be able to completely define a given model of computation, but rather to be able to compare and contrast its notions of concurrency, communication, and time with those of other models of computation. The framework is also not intended to be itself a model of computation, so it should not be interpreted as some “grand unified model” that when implemented will obviate the need for other models. It is too general for any useful implementation and too incomplete to provide for computation. It is meant simply as an analytical tool.

9. Acknowledgments

We wish to acknowledge useful discussions with Gerard Berry, Frédéric Boussinot, Wan-Teh Chang, Stephen Edwards, and Praveen Murthy.

This work was partially supported under the Ptolemy project, which is sponsored by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the National Science Foundation (MIP-9201605), the State of California MICRO program, and

the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, LG Electronics, Mentor Graphics, Mitsubishi, Motorola, NEC, Philips, and Rockwell.

10. References

[1] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1270-1282, 1991.

[2] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.

[3] F. Boussinot, R. De Simone, "The ESTEREL Language," *Proceedings of the IEEE*, Vol. 79, No. 9, September 1991.

[4] C. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis*, Irwin, Homewood IL, 1993.

[5] E. Dijkstra, "Cooperating Sequential Processes", in *Programming Languages*, E F. Genuys, editor, Academic Press, New York, 1968.

[6] C. Ellingson and R. J. Kulpinski, "Dissemination of System-Time," *IEEE Trans. on Communications*, Vol. Com-23, No. 5, pp. 605-624, May, 1973.

[7] C. J. Fidge, "Logical Time in Distributed Systems," *Computer*, Vol. 24, No. 8, pp. 28-33, Aug. 1991.

[8] G. S. Fishman, *Principles of Discrete Event Simulation*, Wiley, New York, 1978.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[10] Y.-C. Ho (Ed.), *Discrete Event Dynamic Systems: Analyzing Complexity and Performance in the Modern World*, IEEE Press, New York, 1992.

[11] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[12] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July, 1978.

[14] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE Proceedings*, September, 1987.

[15] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, May 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[16] F. Mattern, "Virtual Time and Global States of Distributed Systems," in Parallel and Distributed Algorithms, M. Cosnard and P. Quinton, eds., North-Holland, Amsterdam, 1989, pp. 215-226.

[17] A. Mazurkiewicz, “Traces, Histories, Graphs: Instances of a Process Monoid,” in *Proc. Conf. on Mathematical Foundations of Computer Science*, M. P. Chytil and V. Koubek, eds., Springer-Verlag LNCS 176, 1984.

[18] D. G. Messerschmitt, “Synchronization in Digital System Design,” *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 8, pp. 1404-1419, October 1990.

[19] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[20] V. R. Pratt, “Modeling Concurrency with Partial Orders,” Int. J. of Parallel Programming, Vol. 15, No. 1, pp. 33-71, Feb. 1986.

[21] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag (1985).

[22] M. Raynal and M. Singhal, “Logical time: Capturing Causality in Distributed Systems,” *Computer*, Vol. 29, No. 2, February 1996.

[23] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, MA, 1977.

[24] W. T. Trotter, *Combinatorics and Partially Ordered Sets*, Johns Hopkins University Press, Baltimore, Maryland, 1992.

[25] G. Winskel, *The Formal Semantics of Programming Languages*, the MIT Press, Cambridge, MA, USA, 1993.

[26] R. K. Yates, “Networks of Real-Time Processes,” in Concur ‘93, *Proc. of the 4th Int. Conf. on Concurrency Theory*, E. Best, ed., Springer-Verlag LNCS 715, 1993.