— To appear: *Proceedings of ICCAD*, San Jose, CA, Nov. 10-14, 1996 —

Comparing Models of Computation

Edward A. Lee and Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720-1770
{eal, alberto}@EECS.Berkeley.EDU

Abstract

We give a denotational framework (a "meta model") within which certain properties of models of computation can be understood and compared. It describes concurrent processes 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. 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, Petri nets, and discrete-event systems.

1. Introduction

Design automation depends on the high-level modeling and specification of systems. Such modeling and specification often uses innovative and sometimes domain-specific languages. A rich and interesting variety of languages with different semantic properties have been developed. These languages are often more abstract than the established conventional languages into which they are frequently compiled.

An impediment to further progress in such abstract 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. These terms attempt to describe the model of computation underlying a language, but by being imprecise and overused, they often do more to cause confusion than to illuminate. This paper describes one approach to this problem, giving a formalism that will enable description and differentiation of models of computation. To be sufficiently precise, this language is a mathematical one, although the mathematics is not much more sophisticated than basic set theory. It is denotational rather than operational, meaning that it declares relationships rather than describing procedures [18]. It is also incomplete, in that it focuses on certain properties of models of computation, namely their concurrency and communication, and ignores other aspects.

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 [19], 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 number of terms. 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 $e_1 = (t, v_1) \in s$ and $e_2 = (t, v_2) \in s$, 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 $\lambda \in S$ and $\Lambda \in S^N$. For any signal s, $s \cup \lambda = s$, and for any tuple $\mathbf{s}$, $\mathbf{s} \cup \Lambda = \mathbf{s}$, where by the notation $\mathbf{s} \cup \Lambda$ 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 $\perp$ (called "bottom"), which indicates the absence of a value. Notice that while it might seem intuitive that $(t, \perp) \in \lambda$ for any $t \in T$, this would violate $s \cup \lambda = s$ (suppose that s already contains an event at t). Thus, it is important to view $\perp$ 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 [5][10]. 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 $\mathbf{s} \in S^N$ is said to *satisfy* the process if $\mathbf{s} \in P$. 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 define the processes as subsets of S^8.

A *connection* $C \subseteq S^N$ 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

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

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

$$Q = \bigcap_{P_i \in \mathbf{P}} P_i, \quad (2)$$

where $\mathbf{P}$ is the collection of processes $P_i \subseteq S^N$, $1 \le i \le M$. 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 $P \subseteq S^N$, the *projection* along s_j onto S^N, $\Pi_j(P) \subseteq S^{N-1}$, is defined by

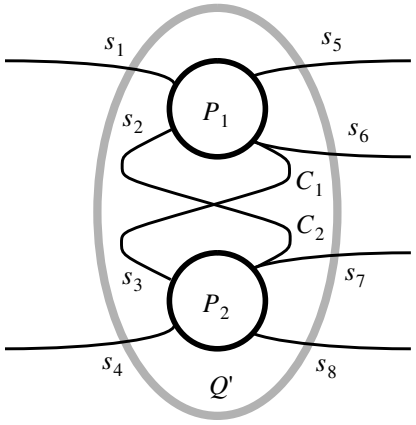

Figure 1. **An interconnection of processes.**

$(s_1, ..., s_{j-1}, s_{j+1}, ..., s_N) \in \Pi_j(P)$ if there exists

$s_j \in S$ such that $(s_1, ..., s_{j-1}, s_j, s_{j+1}, ..., s_N) \in P$. (3)

Thus, in figure 1, we can define the composite process $Q' = \Pi_2(\Pi_3(Q)) \subseteq S^6$. 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 $\pi_j(P) \subseteq S$ is defined by

$s \in \pi_j(P)$ if there exist $s_1, ..., s_{j-1}, s_{j+1}, ..., s_N \in S$

such that $(s_1, ..., s_{j-1}, s, s_{j+1}, ..., s_N) \in P$. (4)

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 $I \subseteq S^N$ such that $I \cap P$ is the total set of acceptable behaviors. The set of all possible inputs $B \subseteq 2^{S^N}$ is a further characterization of a process. For example, $B = \{I ; I \subseteq S^N, \pi_1(I)=s, s \in S\}$ 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 for any input constraint $I \in B$ 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. In such cases, 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 $(\mathbf{s}_1, \mathbf{s}_2) \in F$ and $(\mathbf{s}_1, \mathbf{s}_3) \in F$, 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 $B = \{\{(\boldsymbol{p}, \boldsymbol{q}); \boldsymbol{q} \in S^n\}; \boldsymbol{p} \in S^n\}$. 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 2. 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.

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 $Q \subseteq S^N$ is said to be *closed* if $B = S^N$, a set with only one element, $I = S^N$. Since the set of behaviors is $I \cap P = P$, there are no input constraints. 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 $\mathbf{s}_1 \in S^m$ there is an $\mathbf{s}_2 \in S^n$ such that $(\mathbf{s}_1, \mathbf{s}_2) \in P$. Otherwise it is *partial*. 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.

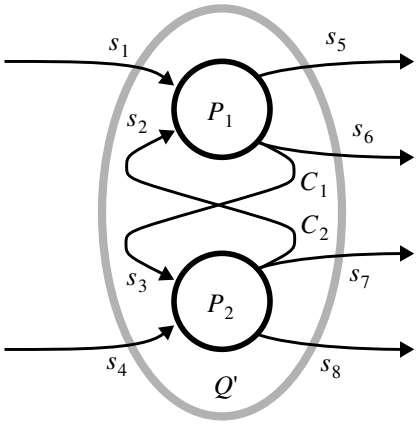

Figure 2. **A partitioning of the signals in figure 1 into inputs and outputs.**

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 [4]. We denote this relation using the template "$<$". Of course, we can define a related relation, denoted "$\leq$", where $t_1 \leq t_2$ 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 using partial orders to model concurrent systems. Pratt motivates doing so, and then generalizes the notion of formal string languages to allow partial ordering rather than just total ordering [16]. Mazurkiewicz uses partial orders in developing an algebra of concurrent "objects" associated with "events" [14]. Partial orders have also been used to analyze Petri nets [17]. Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient [10].

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. 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. 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 $T(s) \subseteq T$ 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.

4.3 Discrete-event systems

Many simulators, including most digital circuit simulators, are based on a discrete-event model (see for example [6]). Given a system Q, and a tuple of signals $\mathbf{s} \in Q$ that satisfies the system, let $T(\mathbf{s})$ denote the set of tags appearing in any signal in the tuple $\mathbf{s}$. Clearly $T(\mathbf{s}) \subseteq T$ 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 $\mathbf{s} \in Q$, $T(\mathbf{s})$ is *order-isomorphic* to a subset of the integers[1]. "Order-isomorphic" means simply that there exists an order-preserving bijection between the events in $T(\mathbf{s})$ and a subset of the integers (or the entire set of integers).

In the control systems community, a discrete-event model also requires that the set of *values* V be countable, or even finite [3][7]. This helps to keep the state space finite, which can be a big help in formal analysis. However, in the simulation community, it is irrelevant whether V is countable [6]. In simulation, the distinction is 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 [11] 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 $\perp \in V$, where $\perp$ (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* logic 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 [20], 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 $P|^t$

1. This elegant definition is due to Wan-Teh Chang.

are tuples and sets of tuples of cut signals, respectively. A functional process F is *causal* if

$$\mathbf{s}_2 = F(\mathbf{s}_1) \Rightarrow \mathbf{s}_2\big|^t = F(\mathbf{s}_1\big|^t)\big|^t \text{ for all } t \in T. \quad (5)$$

Yates [20] considers timed systems with metric time where $\Delta > 0$ is a tag[1]; a functional process F is Δ-*causal* if $F\big|^\Delta = \{\Lambda\}$, the tuple of empty signals, and for all $t \in T, t > \Delta$,

$$\mathbf{s}_2 = F(\mathbf{s}_1) \Rightarrow \mathbf{s}_2\big|^t = F(\mathbf{s}_1\big|^{t-\Delta})\big|^t. \quad (6)$$

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.

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, is easily shown to 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

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, a 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,

$$s_1 \sqsubseteq s_2 \Leftrightarrow s_1 \subseteq s_2$$
$$\text{and for all } e_1 \in s_1 \text{ and } e_2 \in s_2 - s_1,\ e_2 > e_1, \quad (7)$$

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,

$$s_{t_1} \sqsubseteq s_{t_2} \Leftrightarrow t_1 \le t_2. \quad (8)$$

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 $\mathbf{s}_2 \in S^N$, $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2$ if each corresponding element is a prefix, i.e. $s_{1,i} \sqsubseteq s_{2,i}$ for each $1 \le i \le N$. 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 [9]. Our contribution here is only to present these results using our notation. A process F is *monotonic* if it is functional, and

$$\mathbf{s} \sqsubseteq \mathbf{s}' \Rightarrow F(\mathbf{s}) \sqsubseteq F(\mathbf{s}'). \quad (9)$$

A process $F: S^m \to S^n$ 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

$$F(\sqcup\ W) = \sqcup\ F(W). \quad (10)$$

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 [9].

Consider a composition Q of continuous processes $F_1, F_2, ..., F_M$. Assume $Q \subseteq S^N$ for some N. In general, the composition may not be determinate. Consider a trivial case, where $M = 1$ and $F_1: S \to S$ 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 3. Then any signal $s \in S$ satisfies Q. Since there are no inputs to this process and it has many behaviors, it is not determinate.

There is an alternative interpretation of the composi-

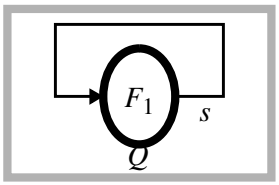

Figure 3. **Example illustrating the need for a least-fixed-point semantics.**

1. The zero element must exist in T for T to be an Abelian group.

tion 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 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 $F_1, F_2, ..., F_M$. It is possible (see [13]) to describe this composition using a continuous function $G : S^q \times S^p \rightarrow S^p$, where p is the total number of output signals for the M functions, and q is the number of remaining signals (which are therefore inputs). We then take the semantics of the composition, given q inputs $\mathbf{q}$ to be the *least fixed point* $\mathbf{p}$ of the equation

$$G(\mathbf{q}, \mathbf{p}) = \mathbf{p} \ . \tag{11}$$

A classic CPO fixed point theorem [4] tells us that this least fixed point exists, and gives us a constructive procedure for finding it.

Under this least-fixed-point semantics, the value of s in figure 3 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 [19].

Notice that the existence of multiple fixed points implies that for a given input constraint $I \subseteq S^N$, 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 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 there is a unique least fixed point).

For the example in figure 3, $Q = S$ (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 $T(s_1) \cap T(s_2) = \varnothing$. 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.

6.2 Sequential processes

A sequential process can be modeled by associating a single signal with the process, as suggested in figure 4(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 Sequential processes with rendezvous

The CSP model of Hoare [8] and the CCS model of Milner [15] involve sequential processes that communicate via rendezvous. Similar models are realized in the languages Occam and Lotos. This idea is depicted in figure 4(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

$$T(e_1) = T(e_2) = T(e_3), \tag{12}$$

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

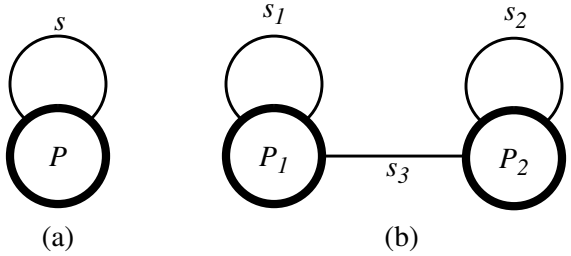

Figure 4. **A sequential process (a) and communicating sequential processes (b).**

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 composition is nondeterminate because of this partial order.

6.4 Dataflow

The dataflow model of computation is a special case of Kahn process networks [12]. 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 5. 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 $s_2 = \{e_{2,i}\}$, 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 [13].

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.

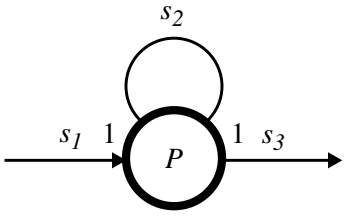

Figure 5. **A simple dataflow process that consumes and produces a single token on each firing.**

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.

6.6 Petri nets

Petri nets can also be modeled in the framework. Petri nets are similar to dataflow, but the events within signals need not be ordered. We associate a signal with each place and each transition in a Petri net. Consider the trivial net in figure 6(a). Viewing the signals s_1 and s_2 as sets of events, there exists a one-to-one function $f: s_2 \to s_1$ such that $f(e) < e$ for all $e \in s_2$. This simply says that every firing (an event in s_2) has a unique corresponding token (an event in s_1) with a smaller tag. In figure 6(b), we simply require that there exist two one-to-one functions $f_1: s_3 \to s_1$ and $f_2: s_3 \to s_2$ such that $f_1(e) < e$ and $f_2(e) < e$ for all $e \in s_3$. In figure 6(c), which represents a nondeterministic choice, we again need two one-to-one functions $f_1: s_2 \to s_1$ and $f_2: s_3 \to s_1$ such that $f_1(e) < e$ for all $e \in s_2$ and $f_2(e) < e$ for all $e \in s_3$, but we impose the additional constraint that $f_1(s_2) \cap f_2(s_3) = \varnothing$, where the notation $f(s)$ refers to the image of the function f when applied to members of the set s. In figure 6(d), we note that if the initial marking of the place is denoted by the set i of events, then it is sufficient to define $s_2 = s_1 \cup i$. Composing these simple primitives then becomes a simple matter of composing the relevant functions. In figure 6(e), $f: s_2 \to s_1$ such that $f(e) < e$ for all $e \in s_2$, $s_2 = s_1$ (the initial marking is empty), therefore

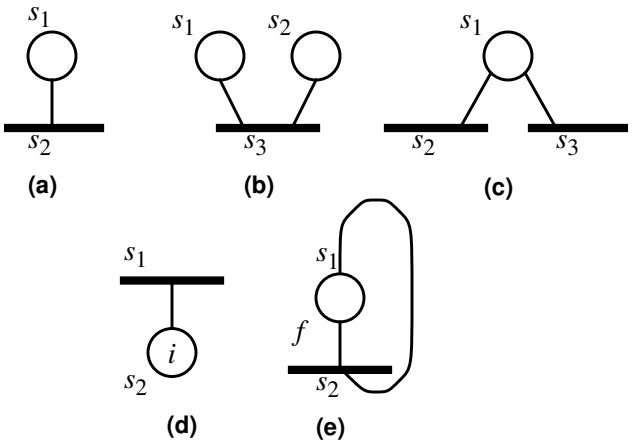

Figure 6. **Some simple Petri nets.**

$s_2 = \emptyset$.

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. 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 its notions of concurrency, communication, and time with those of others. The framework is also not intended to be itself a model of computation, but rather as a "meta model," 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, Alain Girault, Luciano Lavagno, and Praveen Murthy.

This work was partially supported under the Ptolemy project, which is sponsored by DARPA and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the State of California MICRO program, the Semiconductor Research Corporation, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, LG Electronics, Mitsubishi, Motorola, NEC, Philips, and Rockwell. Additional funding came from Magneti Marelli and SGS-Thomson.

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

[4] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 1990.

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

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

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

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

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

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

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

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

[13] E. A. Lee and A. Sangiovanni-Vincentelli, "The Tagged Signal Model — A Preliminary Version of a Denotational Framework for Comparing Models of Computation," Memorandum UCB/ERL M96/33, ERL, University of California, Berkeley, CA 94720, June 4, 1996. (http://ptolemy.eecs.berkeley.edu/papers/96/denotational)

[14] 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.

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

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

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

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

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

[20] 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.