A DENOTATIONAL FRAMEWORK FOR COMPARING MODELS OF COMPUTATION

by

Edward A. Lee and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M97/11

30 January 1997
A DENOTATIONAL FRAMEWORK FOR COMPARING MODELS OF COMPUTATION

by

Edward A. Lee and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M97/11

30 January 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
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 (a "meta model") within which certain properties of models of computation can be understood and compared. It describes concurrent processes in general terms as sets of possible behaviors. A process is determinate if given the constraints imposed by the inputs there are exactly one or exactly zero behaviors. Compositions of processes are processes with behaviors in the intersection of the behaviors of the component processes. 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 processes have only synchronous signals as behaviors. Strict causality (in timed tag systems) and continuity (in untimed tag 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

A major impediment to further progress in modeling and specification of concurrent systems is the
confusion that arises from different usage of common terms. Terms like “synchronous”, “discrete event”, “dataflow”, “signal”, 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. It is not intended as a “grand unifying model of computation” but rather as a “meta model” within which certain properties can be studied. To be sufficiently precise, this language is a mathematical one. It is denotational, in the sense of Scott and Strachey [27], 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 [29], where the notion of a single global state may not be well-defined. In our approach, the denotation of a process is a partial function or a relation on signals, and hence we can model concurrency well.

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, Petri nets, and process networks. Our definitions of these terms 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.

Our objectives overlap somewhat with prior efforts to provide mathematical models for concurrent systems, such as CSP [15], CCS [23], event structures [30], and interaction categories [1]. We do not have a good answer for the question “do we really need yet another meta model for concurrent systems?” except perhaps that our objectives are somewhat different, and result in a model that has some elements in common with other models, but overall appears to be somewhat simpler. It is more
descriptive of concurrency models (more "meta") than some process calculi, which might for example assume a single interaction mechanism, such as rendezvous, and show how other interaction mechanisms can be described in terms of it. We assume no particular interaction mechanism, and show how to use the framework to describe and compare a number of interaction mechanisms (including rendezvous). We devote most of our attention, however, to interaction mechanisms in practical use for designing electronic systems, such as discrete-event models and dataflow.

The prior frameworks closest to ours, Abramsky’s interaction categories [1] and Winskell’s event structures [30], have been presented as categorical concepts. We avoid category theory here because it does not appear to be necessary for our more limited objectives, and because we wish to make the concepts more accessible to a wider audience. But it would be wrong to not acknowledge the influence.

We limit the mathematics to sets, posets, relations, and functions.

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 will use tags to model time, precedence relationships, synchronization points, and other key properties of a model of computation. The values represent the operands and results of computation.

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 $\mathcal{P}(T \times V)$ (the set of all subsets of $T \times V$). 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 sig-
nals $S$, where of course $S = \emptyset (T \times V)$. It is often useful to form a collection or tuple $s$ of $N$ signals. The set of all such tuples will be denoted $S^N$. Position in the tuple serves the same purposes as naming of signals in other process calculi. Reordering of the tuple serves the same purposes as renaming. A similar use of tuples is found in the interaction categories of Abramsky [1].

The empty signal (one with no events) will be denoted by $\lambda$, and the tuple of empty signals by $\Lambda$, 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 $s$, $s \cup \lambda = s$ (ordinary set union). For any tuple $s$, $s \cup \Lambda = s$, where by the notation $s \cup \Lambda$ 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 $\bot$ (called "bottom"), which indicates the absence of a value. Notice that while it might seem intuitive that $(t, \bot) \in \lambda$ 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 $\bot$ as an ordinary member of the set $V$ like any other member.

2.2 PROCESSES

In the most general form, a process $P$ is a subset of $S^N$ for some $N$. A particular $s \in S^N$ is said to satisfy the process if $s \in P$. An $s$ that satisfies a process is called a behavior of the process. Thus a process is a set of possible behaviors. A process may also be viewed as a relation between signals.¹

2.2.1 Composing processes

Since a process is a set of behaviors, a composition of processes should be simply the intersection of the behaviors of each of the processes. A behavior of the composition process should be a behavior of each of the component processes. However, we have to use some care in forming this intersection.

¹. A relation between sets $A$ and $B$ is simply a subset of $A \times B$. 
Consider for example the two processes $P_1$ and $P_2$ in figure 1. These are each subsets of $S^d$. There, we can define a composite process as a subset of $S^8$ simply by forming the cross product of the sets of behaviors $Q = P_1 \times P_2$. Since there is no interaction between the processes, a behavior of the composite process consists of any behavior of $P_1$ together with any behavior of $P_2$. A behavior of $Q$ is an 8-tuple, where the first 4 elements are a behavior of $P_1$ and the remaining 4 elements are a behavior of $P_2$.

More interesting systems have processes that interact. Consider figure 2. A connection $C \subset S^N$ is a particularly simple process where two (or more) of the signals in the $N$-tuple are constrained to be identical. For example, in figure 2, $C_{4,5} \subset S^8$ where
\[
s = (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8) \in C_{4,5} \text{ if } s_4 = s_5. \tag{1}
\]

$C_{2,7}$ can be given similarly as $s_2 = s_7$. There is nothing special about connections as processes, but they are useful to couple the behaviors of other processes. For example, in figure 2, the composite pro-

![Diagram of composition of independent processes](image.png)

FIGURE 1. Composition of independent processes.

1. This serves a similar purpose as the tensor product in the interaction categories of Abramsky [1].
cess may be given as \((P_1 \times P_2) \cap C_{4,5} \cap C_{2,7}\). That is, any \(s \in S^8\) that satisfies the composite process must be a member of each of \(P_1 \times P_2, C_{4,5},\) and \(C_{2,7}\).

Given \(M\) processes in \(S^N\) (some of which may be connections), a process \(Q\) composed of these processes is given by

\[
Q = \bigcap_{P_i \in P} P_i ,
\]

where \(P\) is the collection of processes \(P_i \subseteq S^N, 1 \leq i \leq M\).

As suggested by the gray outline in figure 2, it makes little sense to expose all the signals of a composite process. In figure 2, for example, since signals \(s_2\) and \(s_3\) are identical to \(s_7\) and \(s_4\) respectively, it would make more sense to "hide" two of these signals and to model the composition as a subset of \(S^6\) rather than \(S^8\).

Let \(I = (i_1, \ldots, i_m)\) be an ordered set of indexes in the range \(1 \leq i \leq N\), and define the projection \(\pi_I(s)\) of \(s = (s_1, \ldots, s_N) \subseteq S^N\) onto \(S^m\) by \(\pi_I(s) = (s_{i_1}, \ldots, s_{i_m})\). Thus, the ordered set of indexes

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure2}
\caption{An interconnection of processes.}
\end{figure}
defines the signals that are part of the projection and the order in which they appear in the resulting tuple. The projection can be generalized to processes. Given a process $P \subseteq S^N$, define the projection $\pi_j(P)$ to be the set $\{s' \text{ such that there exists } s \in P \text{ where } \pi_j(s) = s' \}$. Thus, in figure 2, we can define the composite process $Q' = \pi_1((P_1 \times P_2) \cap C_{4,5} \cap C_{2,7}) \subseteq S^6$, where $I = \{1, 3, 4, 6, 7, 8\}$.

If the two signals in a connection are associated with the same process, as shown in figure 3, then the connection is called a self-loop. For the example in figure 3, $Q = \pi_1(P \cap C_{1,3})$, where $I = \{2, 3, 4\}$. For simplicity, we will often denote self-loops with only a single signal, obviating the need for the projection or the connection.

Note that this projection operator is really quite versatile. There are several other ways we could have used it to define the composition in figure 2, even avoiding connection processes altogether. The operator can also be used to construct arbitrary permutations of signals, accomplishing the same end as renaming in other process calculi. Some basic examples are shown in figure 4. Note that the numbering of signals (cf. names) affects the expression for the composition. Note further that figure 4d shows that the connection processes are easily replaced by more carefully constructed intersections.

2.2.2 Inputs and outputs

Many processes (but by no means all) have the notion of inputs, which are events or signals that

![Figure 3. A self loop.](image-url)
are defined outside the process. Formally, an input to a process in \( S^N \) is an externally imposed constraint \( A \subseteq S^N \) such that \( A \cap P \) is the total set of acceptable behaviors. The set of all possible inputs \( B \subseteq \wp(S^N) \) is a further characterization of a process. Within this definition, there is a very rich set of ways to model inputs. Inputs could be individual events, for example, or entire signals. Fortunately, the latter case is more useful for most models of computation, and can easily be defined more precisely.

Given a process \( P \subseteq S^N \) with \( m \) input signals having indexes in the set \( I \), each element \( A \in B \) is a set of tuples of signals \( \{ s : \pi_i(s) = s' \} \) for some \( s' \in S^m \). In other words, the input completely defines \( s' \), a tuple of \( m \) input signals. By saying that \( A \cap P \) is the set of acceptable behaviors, we simply say that

\[
Q = P_1 \times P_2
\]

(a)

\[
Q = \pi_{\{1\}}(P_1) \times P_2 \times \pi_{\{2\}}(P_1)
\]

(b)

\[
Q = P_1 \times S
\]

(c)

\[
Q = \pi_{\{1,4\}}((P_1 \times P_2) \cap C_{2,3}) = \pi_{\{1,3\}}((P_1 \times S) \cap (S \times P_2))
\]

(d)

\[
Q = \pi_{\{1,3,4,6\}}((P_1 \times P_2) \cap C_{2,5})
\]

(e)

**FIGURE 4.** Examples of composition of processes.
the \( m \) input signals must appear within any behavior tuple.

A process \( Q \subseteq S^N \) is said to be closed if \( B = \{ S^N \} \), a set with only one element, \( A = S^N \). Since the set of behaviors is \( A \cap P = P \), there are no input constraints in a closed process. A process is open if it is not closed.

So far, however, we have not captured the notion of a process "determining" the values of the outputs depending on the inputs. To do this, consider an index set \( I \) for \( m \) input signals and an index set \( O \) for \( n \) output signals. A process \( P \) is functional\(^1\) with respect to \((I, O)\) if for every \( s \in P \) and \( s' \in P \) where \( \pi_I(s) = \pi_I(s') \), it follows that \( \pi_O(s) = \pi_O(s') \). For such a process, there is a single-valued mapping \( F : S^m \rightarrow S^n \) such that for all \( s \in P \), \( \pi_O(s) = F(\pi_I(s)) \). A process is total if \( \pi_I(P) = S^m \). In this case, \( F \) is defined over all \( S^m \). It is partial otherwise, i.e. \( \pi_I(P) \subseteq S^m \).

Note that a given process may be functional with respect to more than one pair of index sets \((I, O)\). A connection, for example \( s_1 = s_2 \), is functional with respect to either \((\{1\}, \{2\})\) or \((\{2\}, \{1\})\). In both cases, \( F \) is the identity function.

In figures 2, 3, and 4, there is no indication of which signals might be inputs and which might be outputs. Figure 5 modifies figure 2 by adding arrowheads to indicate inputs and outputs. In this case, \( P_1 \) might be functional with respect to \((I, O) = (\{1, 2\}, \{3, 4\})\).

2.2.3 Determinacy

A process is determinate if for any input \( A \in B \) it has exactly one behavior or exactly no behaviors; i.e. \( |A \cap P| = 1 \) or \( |A \cap P| = 0 \), where \( |X| \) is the size of the set \( X \). Otherwise, it is nondeterminant.

---

1. A relation \( R \subseteq A \times B \) is a function if for every \((a, b) \in R\) and \((a, c) \in R\), \( b = c \).
note. Thus, whether a process is determinate or not depends on our characterization \( B \) of the set of possible inputs.

A process in \( S^N \) that is functional with respect to \((I, O)\) is obviously determinate if \( I \) and \( O \) together contain all the indexes in \( 1 \leq i \leq N \). Given the input signals, the output signals are determined (or there is unambiguously no behavior, if the function is partial).

In figure 4, if all processes are functional with inputs on the left and outputs on the right, then obviously the composition processes are also functional. Thus, the compositions in figure 4 preserve determinacy. A slightly more subtle situation involves source processes (processes with outputs but no inputs), like the example in figure 6. This composition will be functional (and hence determinate) if \( P_1 \) is functional and \( P_2 \) has exactly one behavior.

A much more complicated situation involves feedback, as illustrated by the example in figure 7. Whether determinacy is preserved depends on the tag system and more details about the process.

3. Tag Systems

So far, tags have had no explicit role in the description of processes. But we have also said nothing

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure5.png}
\caption{A partitioning of the signals in figure 1 into inputs and outputs.}
\end{figure}
about the operational interaction of processes. Do they synchronize? Are they causal? Under what conditions exactly are they determinate? To answer these questions, we need structure in the system of tags. This structure turns out to be the major distinguishing feature between various concurrent models of computation.

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). Such a simple model of time is certainly intuitively appealing.

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, but should instead reflect ordering induced by causality (this will be

\[ Q = \pi_{\{1, 3, 5\}}((P_1 \times P_2) \cap C_{2, 4}) \]

FIGURE 6. Composition of a functional process with a source process.

\[ \]

\[ Q \]

FIGURE 7. Feedback (a directed self-loop).
explained below).

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 (i.e. again neglecting relativistic effects), 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 [10][17][22][26]. This difficulty appears even in relatively small systems, such as VLSI chips, where clock distribution is challenging. If an implementation cannot maintain a consistent view of time across its subsystems, then it may be inappropriate for its model to do so (it depends on what questions the model is expected to answer).

The central role of a tag system is to establish ordering among events. An *ordering relation* on the set $T$ is a reflexive, transitive, antisymmetric relation on members of the set. We denote this relation using the template "$\leq\$. Reflexive means that $t \leq t$, transitive means that $t \leq t'$ and $t' \leq t''$ imply that $t \leq t''$, and antisymmetric means that $t \leq t'$ and $t' \leq t$ imply $t = t'$, for all $t, t', t''$ in $T$. Of course, we can define a related irreflexive relation, denoted "$<\$, where $t < t'$ if $t \leq t'$ and $t \not= t'$. The ordering of the tags induces an ordering of events as well. Given two events $e = (t, v)$ and $e' = (t', v')$, $e < e'$ if and only if $t < t'$. A set $T$ with an ordering relationship is called an *ordered set*. If the ordering relationship is partial (there exist $t, t' \in T$ such that neither $t < t'$ nor $t' < t$, then $T$ is called a *partially-ordered set* or *poset* [7][28].

3.1 TIMED MODELS OF COMPUTATION

A *timed model of computation* has a tag system where $T$ is a *totally ordered set*. That is, for any distinct $t$ and $t'$ in $T$, either $t < t'$ or $t' < t$. In timed systems, a tag is also called a *time stamp*. There
are several distinct flavors of timed models.

3.1.1 Metric time

Some timed models of computation include operations on tags. At a minimum, \( T \) may be 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 \( t \in T \). Finally, for every element \( t \in T \), there is another element \( -t \in T \) 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 tag system, \( T \) has a metric, which is a function \( d:T \times T \rightarrow \mathbb{R} \), where \( \mathbb{R} \) is the set of real numbers, that satisfies the following conditions:

\[
\begin{align*}
    d(t, t') & = d(t', t) \quad (3) \\
    d(t, t') & = 0 \iff t = t' \quad (4) \\
    d(t, t') & \geq 0 \quad \text{and} \quad (5) \\
    d(t, t') + d(t', t'') & \geq d(t, t'') \quad (6)
\end{align*}
\]

for all \( t, t', t'' \in T \). Such systems are said to have metric time. In a typical example of metric time, \( T \) is the set of real numbers and \( d(t - t') = |t - t'| \), the absolute value of the difference. Metric time is frequently used when directly modeling physical systems (without relativistic effects).

3.1.2 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 \( s \) that satisfies the system. A connected set is one where no matter how it is partitioned into two dis-
joint sets, at least one of these contains limit points of sequences in 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 (more general topological definitions are also possible).

3.1.3 Discrete-event

Many simulators, including most digital circuit simulators, are based on a discrete-event model (see for example [12]). Given a process $P$, and a tuple of signals $s \in P$ that satisfies the process, let $T(s)$ denote the set of tags appearing in any signal in the tuple $s$. Clearly $T(s) \subseteq T$ and the ordering relationship for members of $T$ induces an ordering relationship for members of $T(s)$. A discrete-event model of computation has a timed tag system, and for all processes $P$ and all $s \in P$, $T(s)$ is order-isomorphic to a subset of the integers$^1$. We explain this now in more detail.

A map $f:A \to B$ from one ordered set $A$ to another $B$ is order-preserving or monotonic if $a < a'$ implies that $f(a) < f(a')$, where the ordering relations are the ones for the appropriate set. A map $f:A \to B$ is a bijection if $f(A) = B$ (the image of the domain is the range) and $a \neq a'$ implies that $f(a) \neq f(a')$. An order isomorphism is an order-preserving bijection. Two sets are order-isomorphic if there exists an order isomorphism from one to the other.

This definition of discrete-event systems corresponds well with intuition. It says that the time stamps that appear in any behavior can be enumerated in chronological order. Note that it is not sufficient to just be able to enumerate the time stamps (the ordering is important). The rational numbers, for example, are enumerable, but would not be a suitable set of time stamps for a discrete-event system. This is because between any two rational numbers, there are an infinite number of other rational numbers. Thus it is also not sufficient for $T(s)$ to be merely isomorphic to a set of integers, since the ratio-

$^1$This elegant definition is due to Wan-Teh Chang.
nals are isomorphic to the set of integers. But they are not order-isomorphic. "Order-isomorphism" captures the notion of "discrete" (indeed, Mazurkiewicz gives a considerably more complicated but equivalent notion of discreteness in terms of relations [21]). It captures the intuitively appealing concept that between any two finite time stamps there will be a finite number of time stamps.

Note further that while we insist that $T(s)$ be discrete (which is stronger than enumerable), we do not even constrain $T$ to be enumerable. For example, it is common for discrete-event systems to take $T$ to be the set of real numbers. We then insist that processes (and inputs) be defined in such a way that $T(s)$ is always a discrete subset of $T$.

If $T(s)$ always has a least tag, then we say that the model is a one-sided discrete-event model of computation. This simply captures the notion of starting the processes at some point in time. In this case, $T(s)$ will be order-isomorphic to a subset of $\omega$, the set of non-negative integers with the usual numerical order. Note in particular that $T(s)$ might be finite, thus capturing the notion of stopping the processes, or it might be infinite.

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 [6][14]. 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 [12]. 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.

3.1.4 Discrete-event simulators

The discrete-event model of computation is frequently used in simulators for such applications as
circuit design, communication network modeling, transportation systems, etc. In a typical discrete-event simulator, events explicitly include time stamps. These are the only types of systems we discuss where the tags are explicit in the implementation. The discrete-event simulator operates by keeping a list of events sorted by time stamp. The event with the smallest time stamp is processed and removed from the list. In the course of processing the event, new events may be generated. These are usually constrained to have time stamps larger than (or sometimes equal to) the event being processed. We will return to this causality constraint later, where we will see that under appropriate circumstances, it ensures determinacy.

In some discrete-event simulators, such as VHDL simulators, tags conceptually contain both a time value and a “delta time.” Delta time has the interpretation of zero time in the simulation, but is an important part of the tag. It is not usually explicit in the simulation, but it affects the semantics. It is used to ensure strict causality (to be defined precisely below), and thus to ensure determinism. A suitable tag system for such a discrete-event simulator would have $T = \omega \times \omega$, where $\omega$ is the set of non-negative integers with the usual numerical order. The first component will typically be called the “time stamp”, while the second component will be called the “delta time offset.” The ordering relation between two tags $t = (t_1, t_2)$ and $t' = (t'_1, t'_2)$ is given by $t < t'$ if and only if $t_1 < t'_1$ or $t_1 = t'_1$ and $t_2 < t'_2$.

Note, however, that $T = \omega \times \omega$ is not order isomorphic with $\omega$ or any subset. In principle, between tags $t = (t_1, t_2)$ and $t' = (t'_1, t'_2)$ where the time stamps $t_1$ and $t'_1$ are finite, there could be an infinite number of tags. This can occur in practice in a discrete-event simulation when a zero-delay feedback loop is modeled and there is no fixed point (or the fixed point is not found). Events circulate forever around the loop, incrementing the delta time component of the tag, but failing to increment the time stamp component. The simulation gets stuck, and time fails to advance. We will see later
3.1.5 Synchronous and discrete-time 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 process is synchronous if every signal in any behavior of the process is synchronous with every other signal in the behavior. A discrete-time system is a synchronous discrete-event system.

By this definition, the so-called Synchronous Dataflow (SDF) model of computation [18] is not synchronous (we will say more about dataflow models below). The “synchronous languages” [2] (such as Lustre, Esterel, and Argos) are synchronous if we consider $\bot \in V$, where $\bot$ (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 [3] is called a synchronous language, but in general, it is not even timed. It supports nondeterminate operations that require a partially ordered tag model. Cycle-based logic simulators are discrete-time systems.

Note that many authors will dispute this definition of the term “synchronous.” For example, the process algebra community (based on CSP [15] and CCS [23], for instance), refers to processes that “synchronize” (rendezvous) as “synchronous.” However, by our definition, they are not even timed (we will have more to say about rendezvous below). We believe that our definition captures the essential and original meaning of the word, latinized from the Greek “sun” (together) and “khronos” (time).

3.1.6 Sequential systems

A degenerate form of timed tag systems is a sequential system. The tagged signal model for a
sequential process has a single signal \( s \), and the tags \( T(s) \) in the signal are totally ordered. For example, under the Von Neumann model of computation, the values \( v \in V \) denote states of the system and the signal denotes the sequence of states corresponding to the execution of a program. Below we will show several ways to construct untimed concurrent systems by composing sequential systems.

### 3.2 Untimed Models of Computation

When tags are partially ordered rather than totally ordered, we say that the tag system is *untimed*. A variety of untimed models of computation have been proposed. In general, the ordering of tags denotes causality or synchronization. Processes can be defined in terms of constraints on the tags in signals.

We are not alone in using 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 [24]. Mazurkiewicz uses partial orders in developing an algebra of concurrent "objects" associated with "events" [21]. Partial orders have also been used to analyze Petri nets [25]. Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient [17]. 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 [11]. 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" [20]. 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 seman-
tics. Thus, efficiency of implementation is not an issue.

3.2.1 Rendezvous of sequential processes

The communicating sequential processes (CSP) model of Hoare [15] and the calculus of communicating systems (CCS) model of Milner [23] are key representatives of a family of models of computation that involve sequential processes that communicate with rendezvous. Similar models are realized, for example, in the languages Occam and Lotos. Intuitively, rendezvous means that sequential processes reach a particular point at which they must verify that another process has reached a corresponding point before proceeding. This can be captured in the tagged signal model as depicted in figure 8. In this case $T(s_i)$ is totally ordered for each $i = 1, 2, 3$. Thus, each $(P_i, s_i)$ for $i = 1, 2, 3$ denotes a sequential process. 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

$$T(e_1) = T(e_2) = T(e_3),$$

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

Note that although the literature sometimes refers to CSP and CCS as synchronous models of computation, under our definition they are not synchronous. They are not even timed. Events directly modeling a rendezvous are synchronous, but events that are not associated with rendezvous have only a partial ordering relationship with each other. Indeed, this partial ordering is one of the most interesting
properties of these models of computation, particularly when there are more than two processes.

In some such models of computation, a process can reach a state where it will rendezvous with one of several other processes (this sort of behavior is supported, for example, by the “select” statement in Ada). In this case, a composition of such processes is often nondeterminate.

3.2.2 Kahn process networks

In a Kahn process network [16], processes communicate via channels, which are one-way unbounded FIFO queues with a single reader and a single writer. Let $T(s)$ again denote the tags in signal $s$. The first-in, first-out property of the channels implies that $T(s)$ is totally ordered for each signal $s$. But the set of all tags $T$ is in general partially ordered. Moreover, signals are discrete, or more technically, $T(s)$ is order-isomorphic with a set of integers for each signal $s$.

For example, consider a simple process that produces one output event for each input event. Suppose the input signal is $s = \{ e_i; i \in \omega \}$, where $\omega$ is the set of non-negative integers with the usual numerical order, and $i < j \Rightarrow e_i < e_j$. Let the output be $s' = \{ e'_i; i \in \omega \}$, similarly ordered. Then the process imposes the ordering constraint that $e_i < e'_i$ for all $i \in \omega$.

The importance of the tags in a particular signal $s$ is limited to the ordering that it imposes on events. Let $\Sigma(s)$ denote the sequence of values in $s$ (an ordered set, ordered according to the tags). That is, the tags are discarded. Then two signals $s$ and $s'$ are sequence equivalent if $\Sigma(s) = \Sigma(s')$. Thus $\Sigma$ induces a set $E_\Sigma$ of equivalence classes in $S$, the set of signals, where each member of $E_\Sigma$ is a set of signals $s$ all with the same sequence $\Sigma(s)$. This notion of sequence equivalence generalizes trivially to tuples of signals.

A process is sequence determinate if all of its behaviors are sequence equivalent. A process is
sequence functional if given a set of equivalent tuples of input signals, all possible outputs are sequence equivalent. Thus, a sequence functional process with $m$ inputs and $n$ outputs has a mapping $F' : (E_2)^m \rightarrow (E_2)^n$ rather than $F : S^m \rightarrow S^n$. Later in the paper we will study constraints on these functions that ensure sequence determinacy.

Whether a sequence determinate process is also determinate depends on the tag system. Sometimes it is useful to have a tag system that represents more information than just the ordering of values in sequences. For example, it might model the timing of the execution of a process network, in which case the timing nondeterminism of a concurrent system is represented in the model even if the process itself is sequence determinate.

3.2.3 Dataflow

The dataflow model of computation is a special case of Kahn process networks [19]. 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 self-loop signal is called the firing signal. 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.

The firing signal is ordered like all signals in the model. Consider two successive events in the firing signal $e_i < e_{i+1}$ (successive means there are no intervening events). An input event $e'$ where $e_i < e' < e_{i+1}$ is said to be consumed by firing $e_{i+1}$. An input event that is less than all firing events is consumed by the first firing. An output event $e''$ where $e_i < e'' < e_{i+1}$ is said to be produced by firing $e_i$. An output event that is greater than all firing events is produced by the last firing (if there is one).

1. The term “dataflow” is sometimes applied to Kahn process networks in general, but this fails to reflect the heritage that dataflow has in computer architecture. The dataflow model originally proposed by Dennis [8] had the notion of a “firing” as an integral part. Our use of the term is consistent with that of Dennis.
For example, consider a dataflow process $P$ with one input signal and one output signal, where each firing consumes one input event and produces one output event, as shown in figure 9. Denote the input signal by $s' = \{e'_i; i \in \mathbb{N}\}$, where $i < j \Rightarrow e'_i < e'_j$. The firings are denoted by the signal $s = \{e_i; i \in \mathbb{N}\}$, and the output by $s'' = \{e''_i; i \in \mathbb{N}\}$, which will be similarly ordered. Then the inputs and outputs are related to the firings as $e_i < e'_{i+1} < e_{i+1}$ (the $i + 1$-th firing consumes the $i + 1$-th input) and $e_i < e''_i < e_{i+1}$ (the $i$-th firing produces the $i$-th output) for all $i$. Because of the transitivity of the ordering relation, this implies that $e'_i < e''_i$ for all $i$, an intuitive sort of causality constraint. A network of such processes will establish a partial ordering relationship between the firings of the actors.

Consider modifying figure 9 with a connection as shown in figure 10. This establishes the identity $s' = s''$, but since $e'_i < e''_i$, $s'$ and $s''$ must be empty. This is the only behavior for this process, and it corresponds to deadlock.

![FIGURE 9. A simple dataflow process that consumes and produces a single token on each firing.](image1)

![FIGURE 10. A deadlocked dataflow graph.](image2)
More interesting examples of dataflow actors can also be modeled. The so-called *switch* and *select* actors, for example, are shown in figure 11. 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 9. But they can be fully characterized nonetheless. Suppose the control signal in the switch is given by $s_2 = \{(t_2, \nu_{2,i})\}$, 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 $\nu_{2,i} \in \{0, 1\}$. Let

$$b_k = \sum_{i=1}^{k} \nu_{2,i} \text{ for } k > 0. \quad (8)$$

Denote the input signal by $s_1 = \{e_{1,i}; i \in N\}$ and the output signals by $s_3 = \{e_{3,i}; i \in N\}$ and $s_4 = \{e_{4,i}; i \in N\}$. Then the ordering constraints imposed by the actor are

$$e_{3,k} > e_{1,b_k} \quad (9)$$

FIGURE 11. More complicated dataflow actors.
3.2.4 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 12(a). Viewing the signals $s_1$ and $s_2$ as sets of events, there exists a one-to-one function $f : s_2 \rightarrow s_1$ 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 12(b), we simply require that there exist two one-to-one functions $f_1 : s_3 \rightarrow s_1$ and $f_2 : s_3 \rightarrow s_2$ such that $f_1(e) < e$ and $f_2(e) < e$ for all $e \in s_3$. In figure 12(c), which represents a nondeterministic choice, we again need two one-to-one functions $f_1 : s_2 \rightarrow s_1$ and $f_2 : s_3 \rightarrow s_1$ 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 $f_1(s_2) \cap f_2(s_3) = \emptyset$, where the notation $f(s)$ refers to the image of the function $f$ when applied to members of the set $s$. In figure 12(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. For example, in figure 12(e),

\[ e_{4,m} > e_{1, (m-b_m)} \]
\[ f_2 : s_2 \rightarrow s_1 \cup i_1, \quad f_3 : s_3 \rightarrow s_2 \cup i_2, \quad f_2(e) < e \text{ for all } e \in s_2, \quad \text{and} \quad f_3(e) < e \text{ for all } e \in s_3, \quad \text{so} \quad f_2(f_3(e)) < e \text{ for all } e \in s_3. \]  

In figure 12(f), \( f : s_2 \rightarrow s_1 \) is such that \( f(e) < e \) for all \( e \in s_2 \), and \( s_2 = s_1 \) (the initial marking is empty), therefore \( s_2 = \emptyset \). The Petri net is not live (it is deadlocked).

3.3 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, something like the interaction categories of Abramsky [1].

4. The Role of Tags in Composition of Processes

In Section 2.2.1, where we composed processes according to equation (2), tags played no evident role. Composition was treated there as combining constraints. Without considering tags, we were able to give some simple conditions in Section 2.2.3 under which compositions of functional processes are determinate. We can often do much more by taking the tags into account. We find that in doing so, we can connect our tagged signal model to well-known results in semantics. We will do this now for two special cases, discrete-event systems and Kahn process networks.

4.1 CAUSALITY IN DISCRETE-EVENT SYSTEMS

Causality is a key concept in discrete-event systems. Intuitively, it means that output events do not
have time stamps less than the inputs that caused them. By studying causality rigorously, we can address a family of problems that arise in the design of discrete-event simulators. These problems center around how to deal with synchronous events (those with identical tags) and how to deal with feedback loops. But causality comes in subtly different forms that have important consequences.

Consider a discrete-event tag system where \( T = \mathbb{R} \), the reals. We can define a metric on the set \( S^n \) of \( n \)-tuples of signals as follows:

\[
d(s, s') = \frac{1}{2^\tau},
\]

where \( \tau \) is the smallest tag where \( s \) and \( s' \) differ. If \( s \) and \( s' \) are identical, then we define \( d(s, s') = 0 \), a sensible extrapolation from (11). Two \( n \)-tuples of signals differ at a tag if any of the component signals differ (pairwise). Two signals differ at a tag if one has an event with that tag and the other does not, or if both have events with that tag but the values of these events differ. It is easy to verify that (11) is a metric by checking that it satisfies (3) through (6). In fact, it is an ultrametric, meaning that in addition to satisfying (6), it satisfies the stronger condition

\[
\max(d(t, t'), d(t', t'')) \geq d(t, t'').
\]

This metric is sometimes called the Cantor metric\(^1\).

The Cantor metric converts our set of \( n \)-tuples of signals into a metric space. In this metric space, two signals are "close" (the distance is small) if they are identical up to a large tag. The metric therefore induces an intuitive notion of an open neighborhood. An open-neighborhood of radius \( r \) is the set of all signals that are identical at least up to and including the tag \( \log_2(r^{-1}) \). We can use this metric to classify three different forms of causality.

\(^1\) The applicability of this metric in this context was pointed out to us by Gerard Berry.
A function $F : S^m \rightarrow S^n$ is causal if for all $s, s' \in S^m$,
\[
d(F(s), F(s')) \leq d(s, s').
\] (13)

In other words, two possible outputs differ no earlier than the inputs that produced them.

A function $F : S^m \rightarrow S^n$ is strictly causal if for all $s, s' \in S^m$,
\[
d(F(s), F(s')) < d(s, s').
\] (14)

In other words, two possible outputs differ later than the inputs that produced them (or not at all).

A function $F : S^m \rightarrow S^n$ is delta causal if there exists a real number $\delta < 1$ such that for all $s, s' \in S^m$,
\[
d(F(s), F(s')) \leq \delta d(s, s').
\] (15)

Intuitively, this means that there is a delay of at least $\Delta = \log_2(\delta^{-1})$, a strictly positive number, before any output of a process can be produced in reaction to an input event. Equation (15) is recognizable as the condition satisfied by a contraction mapping.

A metric space is complete if every Cauchy sequence of points in the metric space that converges, converges to a limit that is also in the metric space. It can be verified that the set of signals $S$ in a discrete-event system is complete. The Banach fixed point theorem (see for example [5]) states that if $F : X \rightarrow X$ is a contraction mapping and $X$ is a complete metric space, then there is exactly one $x \in X$ such that $F(x) = x$. This is called a fixed point. Moreover, the Banach fixed point theorem gives a constructive way (sometimes called the fixed point algorithm) to find the fixed point. Given any $x_0 \in X$, $x$ is the limit of the sequence
\[
x_1 = F(x_0), x_2 = F(x_1), x_3 = F(x_2) \ldots
\] (16)
Consider a feedback loop like that in figure 7 in a discrete-event tag system. The Banach fixed point theorem tells us that if the process $P$ is functional and delta causal, then the feedback loop has exactly one behavior (i.e. it is determinate). This determinacy result was also proved by Yates [31], although he used somewhat different methods. Moreover, Banach fixed point theorem gives us a constructive way to find that behavior. Start with any guess about the signals (most simulators start with an empty signal), and iteratively apply the function corresponding to the process. This is exactly what VHDL, Verilog, and other discrete event simulators do. It is their operational semantics, and the Banach fixed point theorem tells us that if every process in any feedback loop is a delta-causal functional process, then the operational semantics match the denotational semantics\(^1\). I.e., the simulator delivers the right answer.

The constraint that processes be delta causal is fairly severe. In particular, it is not automatically satisfied by processes in VHDL, despite the fact that VHDL processes always exhibit “delta” delay. The common term “delta” is misleading. The contraction mapping condition prevents so-called Zeno conditions where between two finite tags there can be an infinite number of other tags. Such Zeno conditions are not automatically prevented in VHDL.

It is possible to reformulate things so that VHDL processes are correctly modeled as strictly causal (not delta causal). Fortunately, a closely related theorem (see [5], chapter 4) states that if $F : X \rightarrow X$ is a strictly causal function and $X$ is a complete metric space, then there is at most one fixed point $x \in X$, $F(x) = x$. Thus, the “delta” delays in VHDL are sufficient to ensure determinacy, but not enough to ensure that a feedback system has a behavior, nor enough to ensure that the constructive procedure in (16) will work.

If the metric space is compact rather than just complete, then strict causality is enough to ensure

\(^1\) This is sometimes called the full abstraction property.
the existence of a fixed point and the validity of the constructive procedure (16) [5]. In general, the
metric space of discrete-event signals is not compact, although it is beyond the scope of this paper to
show this. Thus, to be sure that a simulation will yield the correct behavior, without further constraints,
we must ensure that the function within any feedback loop is delta causal.

4.2 MONOTONICITY AND CONTINUITY IN KAHN PROCESS NETWORKS

Untimed systems cannot have the same notion of causality as timed systems. The equivalent intu-
ition is provided by the monotonicity condition. Monotonicity is enough to ensure determinacy of
feedback compositions. A slightly stronger condition, continuity, is sufficient to provide a constructive
procedure for finding the one unique behavior. These two conditions depend on a partial ordering of
signals called the prefix order.

A partially ordered tag system is a system where the set $T$ of tags is a partially ordered set or
poset, as defined in Section 3. We can also define an order such that the set of signals becomes a poset.
A signal is a set of events. Set inclusion, therefore, provides a natural partial order for signals. Instead
of the symbol "$<\$" that we used for the ordering of tags, we use the symbol "$\subseteq\$" for an ordering based
on set inclusion. This is a reflexive antisymmetric transitive binary relation. Thus, for two signals $s$
and $s'$, $s \subseteq s'$ if every event in $s$ is also in $s'$.

Recall that for Kahn process networks, we let $\Sigma(s)$ denote the sequence of values in the signal $s$,
which is itself always a totally ordered set of events. In this case, another natural partial ordering for
signals emerges; it is called the prefix order. For the prefix order, we write $\Sigma(s) \subseteq \Sigma(s')$ if $\Sigma(s)$ is a
prefix of $\Sigma(s')$ (i.e., if the first values of $\Sigma(s')$ are exactly those in $\Sigma(s)$). Let $\Sigma(S)$ denote the set of
signals partially ordered by this ordering. Clearly, in $\Sigma(S)$, the empty signal $\Sigma(\lambda)$ is a prefix of every
other signal, at the bottom of the partial order, so it is sometimes called bottom.
In partially ordered models for signals, it is often useful for mathematical reasons to ensure that the poset is a complete partial order (CPO). To explain this fully, we need some more definitions. A chain in $\Sigma(S)$ is a set $\{\sigma_i; \sigma_i \in \Sigma(S) \text{ and } i \in U\}$, where $U$ is a totally ordered set (ordered by "≤") and for any $i$ and $i'$ in $U$,

$$\sigma_i \subseteq \sigma_{i'} \iff i \leq i'.$$

(17)

An upper bound of a subset $W \subseteq \Sigma(S)$ is an element $w \in \Sigma(S)$ where every element in $W$ is a prefix of $w$. A least upper bound (LUB), written $\sqcup W$, is an upper bound that is a prefix of every other upper bound. A lower bound and greatest lower bound are defined similarly. A complete partial order (CPO) is a partial order with a bottom element where every chain has a LUB. From a practical perspective, this usually implies that our set $\Sigma(S)$ of sequences must include sequences with an infinite number of values.

These definitions are easy to generalize to $\Sigma(S)^N$, the set of $N$-tuples of sequences. For $\bar{\sigma} \in \Sigma(S)^N$ and $\bar{\sigma}' \in \Sigma(S)^N$, $\bar{\sigma} \subseteq \bar{\sigma}'$ if each corresponding element is a prefix, i.e. $\sigma_i \subseteq \sigma'_i$ for each $1 \leq i \leq N$, where $\bar{\sigma} = (\sigma_1, ..., \sigma_N)$. With this definition, if $\Sigma(S)$ is a CPO, so is $\Sigma(S)^N$. We will assume henceforth that $\Sigma(S)^N$ is a CPO for all $N$.

4.2.1 Monotonicity and continuity

We can now define the untimed equivalents of causality, connecting to well-known results originally due to Kahn [16]. Our contribution here is only to present these results using our notation. A process $P$ is monotonic if it is sequence functional with function $F$, and

$$\bar{\sigma} \subseteq \bar{\sigma}' \Rightarrow F(\bar{\sigma}) \subseteq F(\bar{\sigma'}).$$

(18)
Intuitively, this says that if an input sequence \( \sigma \) is extended with additional events appended to the end to get \( \sigma' \), then the output \( F(\sigma) \) can only be changed by extending it with additional events to get \( F(\sigma') \). I.e., giving additional inputs can only result in additional outputs. This is intuitively the untimed equivalent of causality.

A process \( P \) is continuous if it is sequence functional with function \( F: \Sigma(S)^m \rightarrow \Sigma(S)^n \) and for every chain \( W \subset \Sigma(S)^m \), \( F(W) \) has a least upper bound \( \cup F(W) \), and

\[
F(\cup W) = \cup F(W). \tag{19}
\]

The notation \( F(W) \) denotes a set obtained by applying the function \( F \) to each element of \( W \). Intuitively, this says that the response of the function to an infinite input sequence is the limit of its response to the finite approximations of this input. “Continuous” here is exactly the topological notion of continuity in a particular topology called the Scott topology. In this topology, the set of all signals with a particular finite prefix is an open set. The union of any number of such open sets is also an open set, and the intersection of a finite number of such open sets is also an open set.

A continuous process is monotonic [16]. To see this, suppose \( F: \Sigma(S)^m \rightarrow \Sigma(S)^n \) is continuous, and consider two signals \( \sigma \) and \( \sigma' \) in \( \Sigma(S)^m \) where \( \sigma \subseteq \sigma' \). Define the increasing chain \( W = \{ \sigma, \sigma', \sigma', \sigma', \ldots \} \). Then \( \cup W = \sigma' \), so from continuity,

\[
F(\sigma') = F(\cup W) = \cup F(W) = \cup \{ F(\sigma), F(\sigma') \}. \tag{20}
\]

Therefore \( F(\sigma) \subseteq F(\sigma') \), so the process is monotonic.

Not all monotonic functions are continuous. Consider for example a system where the set of values is binary, \( V = \{ 0, 1 \} \), and
It is easy to show that this is monotonic but not continuous.

Compositions of continuous (or monotonic) functions without feedback, like those in figures 4 and 6, obviously yield continuous (or monotonic) functions. As before, it is only the feedback case that is subtle.

Consider the feedback system of figure 7. In general, it may not be sequence determinate, even if the process is sequence functional and continuous. Consider a trivial case, where the process P is sequence functional with its function $F: \Sigma(S) \rightarrow \Sigma(S)$ being the identity function. This function is certainly continuous. Then any $\sigma \in \Sigma(S)$ satisfies the composite process $Q$ because for any $\sigma \in \Sigma(S)$, $F(\sigma) = \sigma$. Since the process has many behaviors, it is not sequence determinate.

We will now show that there is an alternative interpretation of the composition $Q$ that is sequence determinate. Under this interpretation, any composition of continuous processes is sequence 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.

A well-known fixed point theorem states that a continuous function $F:X \rightarrow X$ in a CPO X has a least fixed point $x$, $F(x) = x$ (see [7], page 89). By “least fixed point” we mean that for any $y$ such that $F(y) = y$, $x \leq y$. Moreover, the theorem gives us a constructive way to find the least fixed point. Putting it into our context, suppose we have a continuous function $F: \Sigma(S)^n \rightarrow \Sigma(S)^n$. Then define the sequence of sequences
\[
\bar{\sigma}_0 = \Sigma(\Lambda), \quad \bar{\sigma}_1 = F(\bar{\sigma}_0), \quad \bar{\sigma}_2 = F(\bar{\sigma}_1), \ldots \quad (22)
\]

Since \( F \) is monotonic and the tuple of empty sequences \( \Sigma(\Lambda) \) is a prefix of all other tuples of sequences, this sequence is a chain. Since \( \Sigma(S)^n \) is a CPO, this chain has a LUB. The fixed-point theorem tells us that this LUB is the least fixed point of \( F \).

This theorem is very similar to the so-called \textit{Knaster-Tarski fixed point theorem}, which applies to complete lattices rather than CPOs [7]. For this reason, this approach to semantics is sometimes called \textit{Tarskian}.

Note that the constructive technique given by (22) is exactly what one would expect in an implementation of Kahn process networks. Begin with all sequences empty, and start iteratively applying functions. This theorem tells us that this operational semantics is consistent with the denotational semantics (the least fixed point semantics), so again we have full abstraction.

Under this least-fixed-point semantics, the value of \( s_2 \) in figure 7 is \( \lambda \), 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 \( P \) because there are no inputs. This reasonable operational semantics therefore agrees with the denotational semantics. For a complete treatment of this agreement, see Winskel [29].

In terms of the tagged signal model, if \( \Sigma(Q) \) is the set of sequence tuples that satisfy the process \( Q \), we are declaring the behavior of the process to be \( \min(\Sigma(Q)) \), the smallest member (in a prefix order sense) of the set \( \Sigma(Q) \). This minimum exists and is in fact equal to the least fixed point, as long as the composing processes are continuous.

Yet another fixed-point theorem deals with monotonic processes that are not continuous. This the-
orem states that a monotonic function on a CPO has a unique least fixed point, but gives no constructive way to find the least fixed point (see [7], page 96). Fortunately, this lack of constructive solution is not a problem in practice since practical monotonic processes are invariably continuous. Of course, non-monotonic processes create many problems.

5. 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, 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. Of course, a great deal of work remains to be done to determine whether it is useful as an analytical tool.

6. Acknowledgments

We wish to acknowledge useful feedback from Gerard Berry, Frédéric Boussinot, Wan-Teh Chang, Stephen Edwards, Simon Gay, Alain Girault, Luciano Lavagno, Praveen Murthy, Rajagopal Nagarajan, and Dick Stevens. All remaining errors and omissions are the fault of the authors.

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, Lockheed...
Martin, Mentor Graphics, Mitsubishi, Motorola, NEC, Philips, and Rockwell.

7. References


