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

February 28, 1997

Invited paper to *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, to appear, 1998.
Also UCB/ERL Memorandum M98/7, March 4th 1998.

MODELING CONCURRENT REAL-TIME PROCESSES USING DISCRETE EVENTS

Edward A. Lee
Professor
EECS, University of California, Berkeley, CA, USA 94720.

Abstract

We give a formal framework for studying real-time discrete-event systems. It describes concurrent processes as sets of possible 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 denote time. Zeno conditions are defined and methods are given for avoiding them. Strict causality ensures determinacy under certain technical conditions, and delta-causality ensures the absence of Zeno conditions.

1. Introduction

Discrete-event systems, where atomic events occur along a physical time line, provide a useful abstraction for many real-time digital systems. This paper gives a formal framework for talking about such systems. Unlike temporal logics [16], which focus on "eventually" and "always," this methodology focuses on "when." Unlike models based on transition systems [2], this one is input/output oriented, and is more concerned with simulation than with verification. As such, the focus of the paper is on definability and determinism (existence and uniqueness of solutions), although hints are given at extensions that support nondeterminism. Some aspects of the modeling technique are inspired by

Yates [26] and Broy [6]. The mathematical framework that is used here was introduced in [15], but we have repeated the essential material in order to make this paper self-contained.

2. Discrete-Event Systems

2.1 SIGNALS

2.1.1 Values and tags

Given a set of *values* V and *tags* $T = \Re$, the reals, we define an event e to be a member of $E = T \times V$. I.e., an event has a tag and a value. We use tags to model time. The values can represent the operands and results of computation. For some applications, $|V| = 1$, in which case the events are said to be *pure*. They carry no value (of interest). Sometimes it is useful to construct models with an earliest time, in which case we use $T = [0, \infty)$.

2.1.2 Signals and tuples of signals

We define a *signal* $s \in S$ to be a set of events, so the set of all signals is $S = \wp(E)$ (the *powerset*, or the set of all subsets of E). Note that by this definition, a signal cannot contain two identical events. They are modeled as a single event[1]. A *functional signal* is a partial function from T to V. By "partial function" we mean a function that is defined 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 $v_1 = v_2$.

Given a tag $t \in T$ and signal $s \in S$, we define $s(t) \subseteq s$ to be the subset of events with tag t. A signal is functional if and only if $|s(t)| \le 1$ for all $t \in T$.

It is often useful to form a *tuple* $\mathbf{s}$ of N signals, written $\mathbf{s} = [s_1, \ldots, s_N]$. The set of all such tuples will be denoted S^N. Position in the tuple serves the same purposes as naming of signals in process calculi [13][22]. Reordering of the tuple serves the same purposes as renaming. A similar use of tuples is

1. An alternative is to define a signal as a multiset, as done by Pratt [23].

found in the interaction categories of Abramsky [1]. We define $\mathbf{s}(t) = [s_1(t), \ldots, s_N(t)]$.

The empty signal (one with no events) will be denoted by λ, and the N-tuple of empty signals by Λ_N. These are signals like any other, so $\lambda \in S$ and $\Lambda_N \in S^N$. For any signal s, $s \cup \lambda = s$ (ordinary set union). For any tuple $\mathbf{s} \in S^N$, $\mathbf{s} \cup \Lambda_N = \mathbf{s}$, where by the notation $\mathbf{s} \cup \Lambda_N$ we mean the pointwise union of the sets in the tuple.

Following Birkhoff and Mac Lane [5], we define S^0 to be a set with a single element, which we denote σ.

2.1.3 Continuous-time, discrete, and Zeno signals

Let $T(s) \subseteq T$ denote the set of distinct tags in a signal s. A *continuous-time* signal s satisfies $T(s) = T$. A *discrete-event signal* or *discrete signal* is one where $T(s)$ is *order-isomorphic* to a subset of the integers[1]. The set of discrete signals is denoted $S_d \subset S$. 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 signals corresponds well with intuition. It says that the tags that appear in any signal can be enumerated in chronological order. Note that it is not sufficient to just be able to enumerate the tags (the ordering is important). It captures the intuitively appealing concept that between any two finite tags there will be a finite number of tags. Mazurkiewicz gives a considerably more complicated but equivalent notion of discreteness in terms of relations [20].

Let $T(\mathbf{s})$ denote the set of tags appearing in any signal in the tuple $\mathbf{s}$. Clearly $T(\mathbf{s}) \subseteq T$. A *dis-*

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

crete-event tuple or *discrete tuple* $\mathbf{s}$ is one where $T(\mathbf{s})$ is order-isomorphic with a subset of the integers. Let $S_d^{(N)}$ denote the set of all discrete N tuples. Note that $S_d^{(N)} \neq S_d^N$ (hence the paretheses in the superscript). Consider for example the two signals

$$s_1 = \{(t, v) \colon t = 0, 1, 2, \ldots\} \tag{1}$$

$$s_2 = \left\{(t, v) \colon t = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots\right\}$$

While each is a member of S_d, the tuple $\mathbf{s} = [s_1, s_2] \notin S_d^{(2)}$. Such a tuple, called a *Zeno tuple*, can cause major difficulties in simulation because the mere presence of signal s_2 implies a need to process an infinite number of events before time can advance beyond $t = 1$, assuming events are processed in chronological order.

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 [8][12]. This helps to keep the state space finite in certain circumstances, which can be a big help in formal analysis. Nonetheless, 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.

2.1.4 Merging signals

The *merge* of m signals is defined to be

$$M(s_1, s_2, \ldots, s_m) = s_1 \cup s_2 \cup \ldots \cup s_m . \tag{2}$$

The *merge* of an m tuple is the merge of its component signals,

$$M(\mathbf{s}) = M([s_1, s_2, \ldots, s_m]) = M(s_1, s_2, \ldots, s_m) . \tag{3}$$

Note that $M(s_1, s_2)$, where s_1 and s_2 are given by (1), is not discrete, despite the fact that s_1 and s_2 are discrete. $M(s_1, s_2)$ is called a *Zeno signal*.

Note further that if s_1 and s_2 are functional signals, that does not imply that $M(s_1, s_2)$ is a functional signal. It could have two values for the same tag. Define the *two-way biased merge* by

$$M_{2b}(s_1, s_2) = s_1 \cup (s_2 - \hat{s}_2) \tag{4}$$

where $\hat{s}_2$ is the largest subset of s_2 such that $T(\hat{s}_2) \subseteq T(s_1)$. In other words, if s_1 and s_2 have events with the same tag, the biased merge includes only the event from s_1. The m way *biased merge* for $m > 2$ is

$$M_b(s) = s \tag{5}$$

$$M_b(s_1, s_2, \ldots, s_m) = M_{2b}(s_1, M_b(s_2, \ldots, s_m)). \tag{6}$$

The biased merge of a tuple is the biased merge of its component signals. The biased merge of functional signals is functional.

2.2 PROCESSES

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*. For $N \geq 2$, process may also be viewed as a *relation* between the N signals in $\mathbf{s}$.[1] The merge and biased merge are processes with $N = m + 1$.

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. Before we can form such an intersection, each process to be composed must be defined as a subset of the same set of signals S^N, called by some researchers its *sort* [3].

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^4, but they are of different sorts. P_1 relates an entirely different set of signals than P_2. The composition involves eight signals, so to form the composition, we must first augment P_1 and P_2 to define them in terms of subsets of S^8. Let

$$\begin{aligned} \hat{P}_1 &= P_1 \times S^4 \\ \hat{P}_2 &= S^4 \times P_2 \end{aligned} \quad . \tag{7}$$

We call these transformations of sort *augmentation*. Below we will give a notation for such transformations in general. Since $\hat{P}_1$ and $\hat{P}_2$ are now of the same sort, and composition is simply their intersection,

$$Q = \hat{P}_1 \cap \hat{P}_2 = (P_1 \times S^4) \cap (S^4 \times P_2) . \tag{8}$$

This can be simplified to

$$Q = P_1 \times P_2 . \tag{9}$$

This parallel composition of non-interacting processes is simply the cross product[1] of the sets of

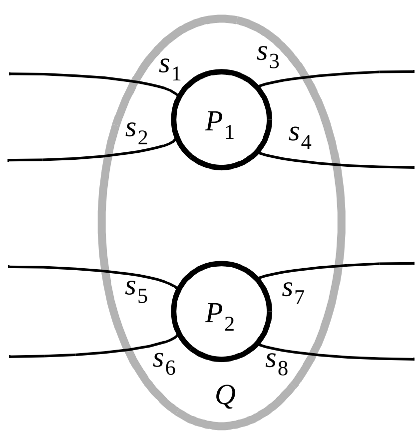


FIGURE 1. Composition of independent processes.

1. The tensor product is used in the interaction categories of Abramsky [1] for the same composition. Here it follows from the intersection of behaviors.

behaviors. 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.

2.2.2 Interacting processes

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

$$\mathbf{s} = [s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8] \in C_{4,5} \text{ if } s_4 = s_5 . \tag{10}$$

$C_{2,7}$ 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 process may be given as

$$Q = (P_1 \times P_2) \cap C_{4,5} \cap C_{2,7} , \tag{11}$$

where the first set is given by (9).

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

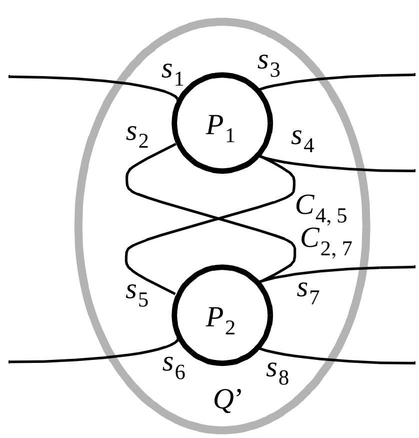


FIGURE 2. An interconnection of processes.

$$Q = \bigcap_{P_i \in \mathbf{P}} P_i\,, \tag{12}$$

where **P** is the collection of processes $P_i \subseteq S^N$, $1 \le i \le M$.

2.2.3 Projection

As suggested by the gray outline in figure 2, often it makes little sense to expose all the signals of a composite process. In figure 2, for example, since signals s_2 and s_5 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. This changes the sort of the composite, which may make it easier to compose it again.

Let $I = [i_1, ..., i_M]$ be an ordered set of M distinct indexes in the range $1 \le i \le N$, and define the *projection* $\pi_I(\mathbf{s})$ of $\mathbf{s} = [s_1, ..., s_N] \in S^N$ onto S^M by

$$\pi_I(\mathbf{s}) = [s_{i_1}, ..., s_{i_M}]\,. \tag{13}$$

Thus, the ordered set of indexes defines the signals that are part of the projection and the order in which they appear in the resulting tuple. If $I = \varnothing$, define $\pi_I(\mathbf{s}) = \sigma \in S^0$.

The projection can be generalized to processes. Given a process $P \subseteq S^N$, define the projection onto S^M by

$$\pi_I(P) = \{\mathbf{s} \in S^M : \exists\, \hat{\mathbf{s}} \in P \text{ where } \pi_I(\hat{\mathbf{s}}) = \mathbf{s}\}. \tag{14}$$

Thus, in figure 2, we can define the composite process

$$Q' = \pi_I((P_1 \times P_2) \cap C_{4,5} \cap C_{2,7}) \subseteq S^6\,, \tag{15}$$

where $I = [1, 3, 4, 6, 7, 8]$. Projection then facilitates composition of this process with others, since the others will not need to be augmented to involve irrelevant signals.

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, $Q = \pi_I(P \cap C_{1,3})$, 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. This is simply a syntactic shorthand; if two signals are constrained to be identical, we lose nothing by considering only one of the signals.

2.2.4 Transformations of sort

Composition is set intersection. Augmentation and projection are syntactic operations that merely give process definitions the right sort to enable composition by intersection. They play no semantic role in composition. Moreover, they can be unified and generalized, providing a notation for arbitrary transformations of sort.

Let H be a map $H: \{1, \ldots, N\} \rightarrow \{1, \ldots, M\}$ such that

$$H(n) = H(n) \Rightarrow n = n. \tag{16}$$

Define the transformation of sort based on H by

$$\pi_H(\mathbf{s}) = \{\hat{\mathbf{s}} \in S^M : H(j) = i \Rightarrow \hat{s}_i = s_j\}. \tag{17}$$

Augmentation is now a special case where $M > N$ and H is a total function, in which case condition (16) is equivalent to H being one-to-one. *Projection* is the special case where $M < N$ and H is an onto partial function. We can also define *permutation*, where $M = N$ and H is a bijective total function.

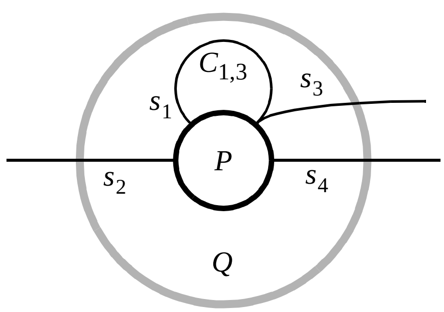


FIGURE 3. A self loop.

Because of condition (16), H can be represented compactly by a tuple $[h_1, \ldots, h_M]$ where $h_i \in \{\varepsilon, 1, 2, \ldots, N\}$. The symbol $h_i = \varepsilon$ indicates that there is no $j \in \{1, 2, \ldots, N\}$ such that $H(j) = i$. Otherwise, $H(h_i) = i$.

Note that this sort transformation 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. In other process calculi, where names are used instead of indexes, *scope* is analogous to our sort. Condition (16) is equivalent to the requirement for unique names within a scope. The sort transformation accomplishes the same end as renaming and hiding in other process calculi.

Some basic examples are shown in figure 4. Note that the indexing of signals (vs. names) affects

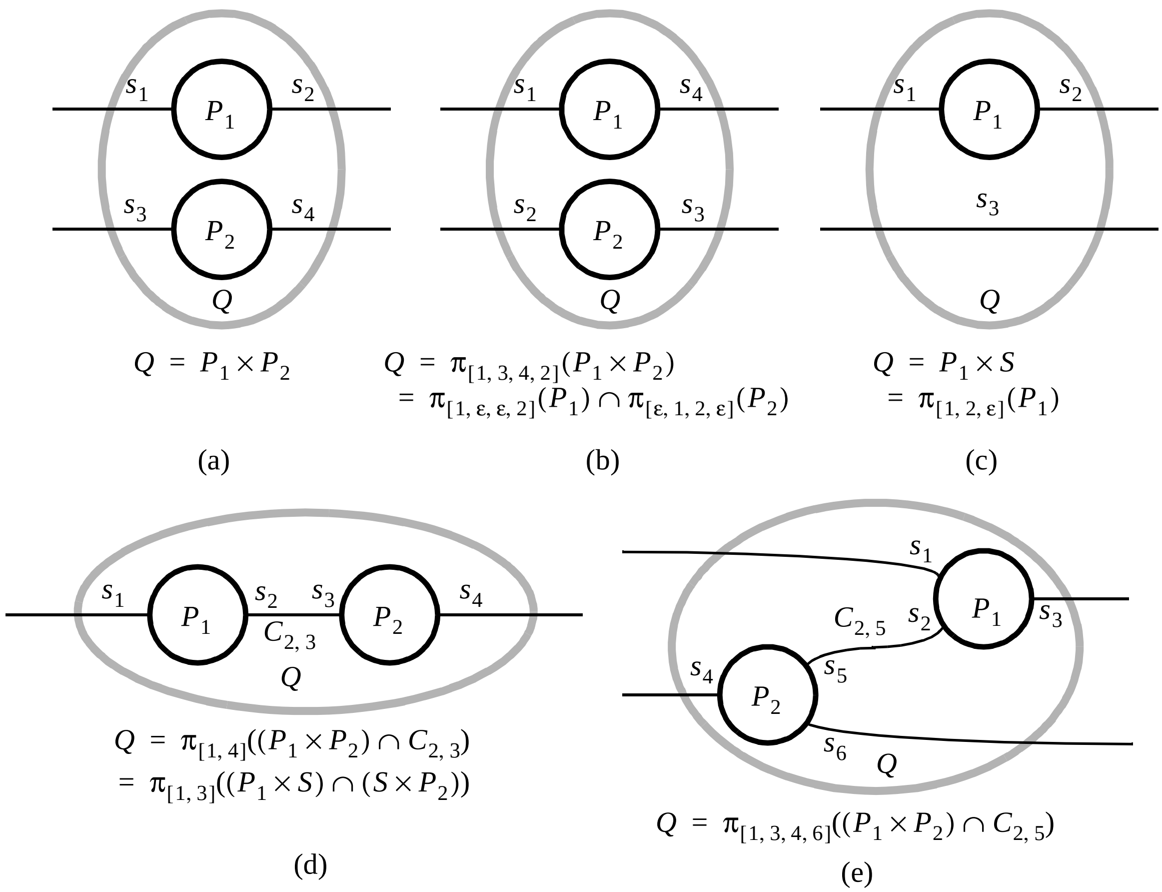


FIGURE 4. Examples of composition of processes.

the manipulation of processes to give them compatible sorts. Note further that figure 4d shows that the connection processes are easily replaced by more carefully constructed intersections.

2.2.5 Inputs, outputs, and functional processes

Consider processes that have input and output signals, where the output signals are given as a function of the input signals. Such processes are called *functional*. Intuitively, input signals are not constrained by the process, while output signals are. Because of the need to compose processes of the same sort, process definitions will typically involve some unconstrained signals that have no effect on the outputs. For convenience, we consider these distinct from the input signals and call them *irrelevant signals*.

Formally, we can partition the index set $\{1, ..., N\}$ of its sort into disjoint subsets I, O, and R such that

$$\{1, ..., N\} = I \cup O \cup R. \tag{18}$$

I is an ordered set of indexes of the input signals, O is an ordered set of indexes of the output signals, and R is an ordered set of indexes of the irrelevant signals. The union here is interpreted as an ordered merge. Given an *input tuple*, $\mathbf{u} \in S^{|I|}$, where $|I|$ is the number of input signals, let

$$U = \{\mathbf{s} \in S^N : \pi_I(\mathbf{s}) = \mathbf{u}\}. \tag{19}$$

Then $B = U \cap P$ is the set of behaviors consistent with this input. Equivalently, $B \subseteq P$ satisfying

$$\begin{aligned} \pi_I(B) &= \{\mathbf{u}\} \\ \pi_R(B) &= S^{|R|} \\ \pi_O(B) &= \{F(\mathbf{u})\} \end{aligned} \tag{20}$$

where $F: S^{|I|} \to S^{|O|}$ is a function relating the output signals to the input signals. A functional process therefore is completely characterized by the tuple

$$(F, I, O, R). \tag{21}$$

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 a functional process with $(F, I, O, R) = (F, \{1, 2\}, \{3, 4\}, \{5, 6, 7, 8\})$ for some function $F: S^2 \to S^2$.

2.2.6 Nondeterminacy

A process is *determinate* if given the inputs it has exactly one behavior. Otherwise, it is *nondeterminate*. Thus, whether a process is determinate or not depends on how we define inputs. A functional process is obviously determinate. The same structure as that of a functional process can be used for some nondeterminate processes. We define a *quasi-functional process* to be one given by

$$(\Phi, I, O, R) \tag{22}$$

where Φ is a set of functions of the form $F: S^{|I|} \to S^{|O|}$. Given an input tuple $\mathbf{u} \in S^{|I|}$, the set of behaviors is $B \subseteq P$ such that

$$\begin{aligned} \pi_I(B) &= \{\mathbf{u}\} \\ \pi_R(B) &= S^{|R|} \\ \pi_O(B) &= \{\mathbf{s} \in S^{|O|} : \exists F \in \Phi \text{ where } \mathbf{s} = F(\mathbf{u})\} \end{aligned} \quad . \tag{23}$$

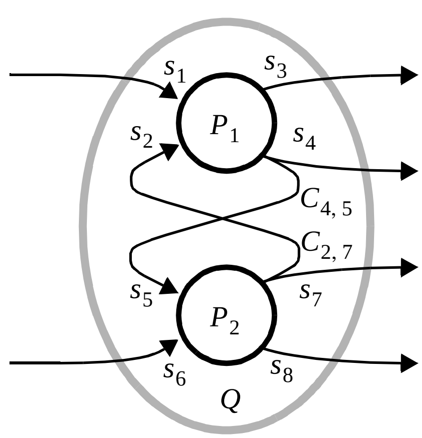


FIGURE 5. A partitioning of the signals in figure 1 into inputs and outputs.

2.2.7 Source processes

A slightly more subtle situation involves *source* processes (processes with outputs but no inputs), like P_2 in figure 6. There, if P_2 is functional, then it would be characterized by

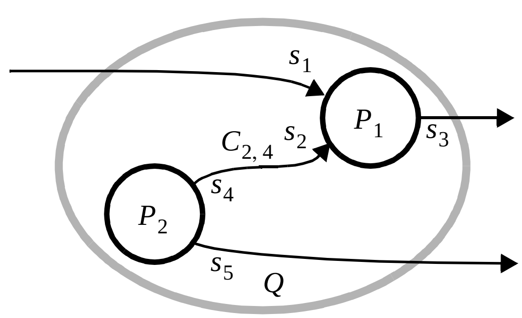
$$(F, I, O, R) = (F, \varnothing, \{4, 5\}, \{1, 2, 3\}) \tag{24}$$

where $F: S^0 \to S^2$. Since S^0 is a set with a single element, the function F always returns the same signal pair. Thus, a functional source is simply a determinate source.

Of course, we can also define a process that is a sink, where $O = \varnothing$. A sink process is trivially determinate and functional.

3. Composition of Functional Processes

In Section 2.2.1, where we composed processes according to equation (12), tags, inputs, outputs, and functions played no evident role. Composition was treated there as combining constraints. However, set intersection gives us no direct way to answer certain key questions about composition, such as whether the composition of two functional processes is functional. We develop in this section a framework within which we can answer this and several other compositionality questions. In particular, we will focus on the notion of causality in discrete-event systems and the role that causality plays in compositionality.



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

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

Intuition alone is sufficient to be convinced that the compositions in figure 4 result in functional processes if the component processes are functional. A more complicated situation involves feedback, as illustrated by the example in figure 7. Whether the composition is functional depends on the tag system and more details about the process. Most interesting discrete-event systems include feedback.

3.1 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 models and 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.

3.1.1 The Cantor Metric

Assume the discrete-event tag system where $T = \Re$, the reals. Consider an n-tuple of signals $\mathbf{s} = [s_1, \ldots, s_n] \in S^n$. Let $\mathbf{s}(t) = [s_1(t), \ldots, s_n(t)]$ where $s_i(t) \subseteq s_1$ is the subset of events in signal s_i with tag t. Thus, $s_i(t) = \lambda$ means that $t \notin T(s_i)$ (there are no events with tag t). We can define a metric on the set S^n of n-tuples of signals as follows[1]:

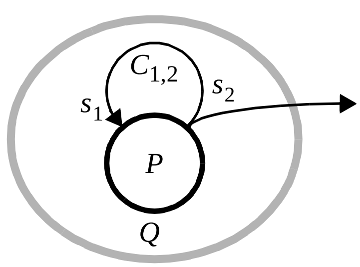


FIGURE 7. Feedback (a directed self-loop).

1. Reed and Roscoe [25] use an infimum over times where the two signals are identical. For discrete signals, the two metrics are identical.

$$d(\mathbf{s}, \mathbf{s}') = sup\left\{\frac{1}{2^t} : \mathbf{s}(t) \neq \mathbf{s}'(t), t \in T\right\}. \tag{25}$$

If we define τ such that

$$d(\mathbf{s}, \mathbf{s}') = \frac{1}{2^\tau} \tag{26}$$

then τ is the smallest tag where $\mathbf{s}$ and $\mathbf{s}'$ differ (if such a tag exists), or the greatest lower bound on the tags where they differ (if there is no smallest tag). Such a smallest tag always exists if $\mathbf{s}$ and $\mathbf{s}'$ are not identical and are discrete. For identical signals, we define

$$d(\mathbf{s}, \mathbf{s}) = 0, \tag{27}$$

a sensible extrapolation from (25) (let $\tau \to -\infty$ in (26)).

It is easy to verify that (25) is a metric. In fact, it is an *ultrametric*, meaning that instead of the triangle inequality,

$$d(\mathbf{s}, \mathbf{s}') + d(\mathbf{s}', \mathbf{s}'') \geq d(\mathbf{s}, \mathbf{s}'') \tag{28}$$

it satisfies the stronger condition

$$max\{d(\mathbf{s}, \mathbf{s}'), d(\mathbf{s}', \mathbf{s}'')\} \geq d(\mathbf{s}, \mathbf{s}''). \tag{29}$$

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 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 $\log_2(r^{-1})$.

1. The applicability of this metric in this context was pointed out to us by Gerard Berry.

3.1.2 Causal, strictly causal, and delta causal functions

We can use this metric to classify three different forms of causality. A function $F : S^m \to S^n$ is *causal* if for all $\mathbf{s}, \mathbf{s}' \in S^m$,

$$d(F(\mathbf{s}), F(\mathbf{s}')) \le d(\mathbf{s}, \mathbf{s}') . \tag{30}$$

In other words, two possible outputs differ no earlier than the inputs that produced them. A causal function is said to be *non-expansive* in this metric space.

A function $F : S^m \to S^n$ is *strictly causal* if for all $\mathbf{s}, \mathbf{s}' \in S^m$,

$$d(F(\mathbf{s}), F(\mathbf{s}')) < d(\mathbf{s}, \mathbf{s}') . \tag{31}$$

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

A function $F : S^m \to S^n$ is *delta causal* if there exists a real number $\delta < 1$ such that for all $\mathbf{s}, \mathbf{s}' \in S^m$,

$$d(F(\mathbf{s}), F(\mathbf{s}')) \le \delta d(\mathbf{s}, \mathbf{s}') . \tag{32}$$

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. Inequality (32) is recognizable as the condition satisfied by a *contraction mapping*.

The *merge* function, defined in (3), satisfies

$$d(M(\mathbf{s}), M(\mathbf{s}')) = d(\mathbf{s}, \mathbf{s}') . \tag{33}$$

Hence, merge is causal, but not strictly or delta causal. The baised merge, defined in (6), satisfies

$$d(M_b(\mathbf{s}), M_b(\mathbf{s}')) \le d(\mathbf{s}, \mathbf{s}') , \tag{34}$$

and thus is also causal.

Consider a source, like $P_2 : S^0 \to S^2$ in figure 6. Since S^0 has only one element, all $\mathbf{s}, \mathbf{s}' \in S^0$ are equal. Thus, every functional (determinate) source is delta causal with $\delta = 0$.

3.1.3 Fixed points

Causality turns out to play a central role in the existence and uniqueness of behaviors under feedback composition. To understand this, we review some basic properties of metric spaces.

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 [7]) states that if $F: X \to X$ 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

$$x_1 = F(x_0), x_2 = F(x_1), x_3 = F(x_2) \ldots \tag{35}$$

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 [26], although he used somewhat different methods. Moreover, the 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. We will study the operational semantics of simulators in more detail below.

The contraction mapping condition prevents so-called *Zeno* conditions where between two finite

1. This is sometimes called the *full abstraction* property.

tags there can be an infinite number of other tags. Such Zeno conditions are not automatically prevented in VHDL, for example.

The constraint that processes be delta causal is fairly severe. We can slightly relax the delta causal condition by observing that it is sufficient that there exist a finite N such that F^N is delta causal. I.e., N cycles around a feedback loop introduce at least Δ delay. This is still not ensured by VHDL simulators, for example, nor by many other discrete-event simulators in practical use.

It is possible to reformulate things so that VHDL processes are correctly modeled as strictly causal (not delta causal) (see [15] for details). Fortunately, a closely related theorem (see [7], chapter 4) states that if $F: X \to X$ 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 (35) will work.

If the metric space is *compact* rather than just complete, then strict causality is enough to ensure the existence of a fixed point and the validity of the constructive procedure (35) [7]. 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 or a finite power of the function within any feedback loop is delta causal.

3.2 COMPOSITIONALITY

We can now formulate precisely what conditions we wish the composition of processes to satisfy. We denote a composition of two processes $P_1 \subseteq S^N$ and $P_2 \subseteq S^N$ of the same sort by a function

$$\phi\colon \wp(S^N) \times \wp(S^N) \to \wp(S^N) \tag{36}$$

where

$$\phi(P_1, P_2) = P_1 \cap P_2 \cap C \tag{37}$$

where C is the intersection of any number of connections of the same sort. We say that ϕ is *compositional* if is satisfies the following four conditions:

1. If P_1 and P_2 are functional, then $\phi(P_1, P_2)$ is functional.
2. If P_1 and P_2 are causal, then $\phi(P_1, P_2)$ is causal.
3. If P_1 and P_2 are strictly causal, then $\phi(P_1, P_2)$ is strictly causal.
4. If P_1 and P_2 are delta causal, then $\phi(P_1, P_2)$ is delta causal.

3.2.1 Acyclic Compositions

First we address the easy case of acyclic compositions of two processes. The general form of these is shown in figure 8. In that figure, the arcs that are shown represent an arbitrary number of signals (including zero) with indexes given by the sets adjacent to the arcs. These sets satisfy the following constraints:

1. They are disjoint.
2. $|O_3| = |I_3|$.
3. Their union is $\{1, \ldots, N\}$.

The composition is given by

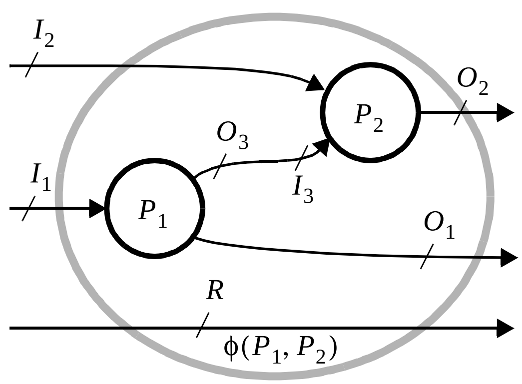


FIGURE 8. Generalized acyclic composition of two functional processes.

$$\phi(P_1, P_2) = P_1 \cap P_2 \cap C \quad . \tag{38}$$
$$C = \{\mathbf{s}: \pi_{O_3}(\mathbf{s}) = \pi_{I_3}(\mathbf{s})\}$$

This composition generalizes the ones shown in figures 1, 4, and 6, which is all acyclic compositions we have considered. We wish to show that if P_1 and P_1 are functional then $\phi(P_1, P_2)$ is functional. Let $\mathbf{s}, \mathbf{s}' \in \phi(P_1, P_2)$. ϕ is functional if

$$\pi_{I_1 \cup I_2}(\mathbf{s}) = \pi_{I_1 \cup I_2}(\mathbf{s}') \Rightarrow \pi_{O_1 \cup O_2}(\mathbf{s}) = \pi_{O_1 \cup O_2}(\mathbf{s}') . \tag{39}$$

The left hand side implies

$$\pi_{I_1}(\mathbf{s}) = \pi_{I_1}(\mathbf{s}') \quad . \tag{40}$$
$$\pi_{I_2}(\mathbf{s}) = \pi_{I_2}(\mathbf{s}')$$

Since P_1 is functional,

$$\pi_{O_3}(\mathbf{s}) = \pi_{O_3}(\mathbf{s}') \quad . \tag{41}$$
$$\pi_{O_1}(\mathbf{s}) = \pi_{O_1}(\mathbf{s}')$$

Since P_2 is functional, (40) and (41) imply

$$\pi_{O_2}(\mathbf{s}) = \pi_{O_2}(\mathbf{s}') \tag{42}$$

which together with (41) implies the right hand side of (39), completing the proof.

Similar methods can be used to show that causality properties are preserved. For example, to show that if P_1 and P_2 are causal that $\phi(P_1, P_2)$ is causal, we need to show that

$$d(\pi_{O_1 \cup O_2}(\mathbf{s}), \pi_{O_1 \cup O_2}(\mathbf{s}')) \le d(\pi_{I_1 \cup I_2}(\mathbf{s}), \pi_{I_1 \cup I_2}(\mathbf{s}')) . \tag{43}$$

To do this, we use the observation that for any two sets $A, B \subseteq \{1, \ldots, N\}$, and $\mathbf{s}, \mathbf{s}' \in S^N$,

$$d(\pi_{A \cup B}(\mathbf{s}), \pi_{A \cup B}(\mathbf{s}')) = max\{d(\pi_A(\mathbf{s}), \pi_A(\mathbf{s}')), d(\pi_B(\mathbf{s}), \pi_B(\mathbf{s}'))\} . \tag{44}$$

We leave the details of the proof as an excercise for the reader. Similar proofs work for strict and delta causality, so compositionality follows.

3.2.2 Feedback Compositions

Consider a general form of feedback composition, shown in figure 9. We have omitted the irrelevant signals. We wish to show that if P_1 is functional and delta causal, then $\psi(P_1)$ is functional and strictly causal. To show that it is functional, we need to show that

$$\pi_{I_1}(\mathbf{s}) = \pi_{I_1}(\mathbf{s}') \Rightarrow \pi_{O_1}(\mathbf{s}) = \pi_{O_1}(\mathbf{s}'). \tag{45}$$

In order to show this, we have to construct an appropriate delta causal functional process with associated function $F: S^M \to S^M$, and then invoke the Banach fixed point theorem. This can be done for a particular input tuple $\pi_{I_1}(\mathbf{s}) = \mathbf{q}$. In figure 10, we redraw the feedback composition, inserting a source process Q to produce the constant signal $\mathbf{q}$, and looping back the outputs with indexes O_1, so that they become inputs to the composition $\varphi(P_1, Q)$. Although they are inputs, they are ignored. But this device allows us to observe that since the composition is $\varphi(P_1, Q)$ is similar to those represented

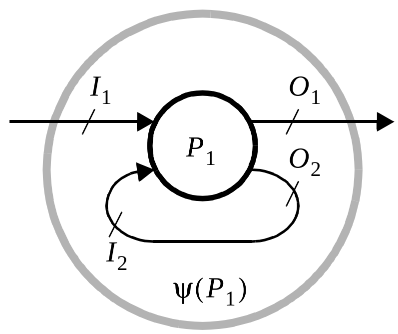


FIGURE 9. General form of feedback composition.

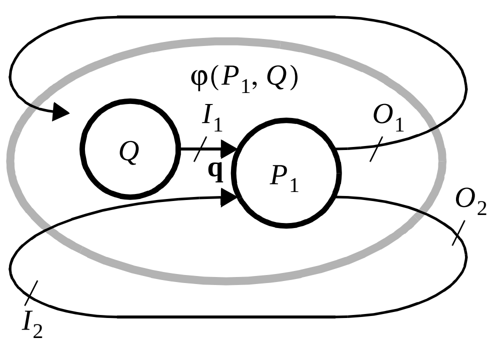


FIGURE 10. Temporary construction to analyze the feedback composition.

by figure 8, then if P_1 is delta causal, the composition will be functional and delta causal, and therefore can be described by a contraction mapping $F: S^M \to S^M$, where $M = |O_1| + |O_2|$. Thus, it has a unique fixed point $\mathbf{s}$ and $\pi_{O_1}(\mathbf{s}) = \pi_{O_1}(\mathbf{s}')$ as desired in (45).

Note that in this case, it is not sufficient for P_1 to be merely strictly causal, because in this case we would not be assured of the existence of a fixed point. If it is merely causal, then we are not assured of either existence or uniqueness. As a result, it may be that $\psi(P_1)$ is not even functional.

We conclude that discrete-event systems are compositional under acyclic composition, but not under cyclic composition. Under cyclic composition, they are only compositional if the process in the feedback loop is delta causal, or some finite power of its function is delta causal.

4. Simulation

The discrete-event model of computation is frequently used in simulators for such real-time applications as circuit design, communication network modeling, transportation systems, etc. A typical 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. What we mean by "processed" is that any process that sees that event on any of its input signal *fires*, performing some computation in reaction to the event. In the course of processing the event, new events may be generated. This simulation procedure provides an operational semantics for DE systems. We are interested in whether the operational semantics matches the denotational semantics we have been studying.

4.1 SEQUENCES OF FIRINGS

Formally, the operational semantics is given in terms of a *firing function* for each process (we assume all processes are functional). The set of firing functions for m-input, n-output processes has the form

$$\Gamma = \{f : S^m \times T \to S^n \times \Gamma\} . \tag{46}$$

That is, a firing function f takes as arguments a tuple of signals and a time stamp, and returns a tuple of signals an a new firing function (called a *continuation*). It is required to satisfy the following *stuttering condition*

$$f(\Lambda_m, t) = (\Lambda_n, f) \text{ for all } t \in T . \tag{47}$$

This condition states simply that nothing changes if no input events are offered to the firing.

We can relate the firing function f to the process function F as follows. Let $\mathbf{s} \in S^m$ be the input. Construct $\mathbf{s}' = F(\mathbf{s})$, $\mathbf{s}' \in S^n$, according to the following sequential procedure:

$$
\begin{array}{l}
\mathbf{s}' = \Lambda_n \\
\text{while } (\mathbf{s} \neq \Lambda_m) \ \{ \\
\qquad \text{let } \tau = min(T(M(\mathbf{s}))) \\
\qquad \text{let } (\mathbf{s}, f) = f(\mathbf{s}(\tau), \tau) \\
\qquad \text{let } \mathbf{s} = \mathbf{s} - \mathbf{s}(\tau) \\
\qquad \text{let } \mathbf{s}' = \mathbf{s}' \cup \mathbf{s} \\
\}
\end{array} \tag{48}
$$

The first statement initializes an empty result. The while loop processes pending events. Within the while loop, the first statement uses the merge operator to identify the smallest pending time stamp τ. The second line fires the process, offering as input events $\mathbf{s}(\tau)$, the events with time stamp τ. The third line uses set subtraction to remove processed events, and the fourth line uses set union to append resulting events to the result.

4.2 RELATIONSHIP BETWEEN THE FIRING FUNCTION AND THE PROCESS FUNCTION

This procedure can be viewed as a functional that, given f, returns F. Consider the simple special case where

$$f(\mathbf{s}, \tau) = (\mathbf{s}', f) . \tag{49}$$

I.e., the continuation is always the same firing function. In this case, it is easy to see that if f is causal, strictly causal, or delta causal, then so is F.

For the more general case, given a firing function f, define it's closure

$$\bar{f} \subseteq \Gamma \tag{50}$$

to be the set of all firing functions reachable from f. Then F is causal or strictly causal if all functions in $\bar{f}$ are causal, strictly causal.

Delta causality is slightly trickier in this case because we need a uniform contraction. F is delta causal if there exists a $\delta < 1$ s.t. for all $f' \in \bar{f}$, $\mathbf{s}, \mathbf{s}' \in S^m$, and $\tau \in T$,

$$\begin{array}{l} d(\mathbf{q}, \mathbf{q}') \le d(\mathbf{s}, \mathbf{s}') \\ (\mathbf{q}, f'') = f'(\mathbf{s}, \tau) \\ (\mathbf{q}', f''') = f'(\mathbf{s}', \tau) \end{array} \quad . \tag{51}$$

4.3 SOURCES

Procedure (48) can be adapted for sources (where $m = 0$) as follows:

$\mathbf{s}' = \Lambda_n$

$\tau = 0$

while (*true*) {

let $(\mathbf{s}, f) = f(\sigma, \tau)$

let $\tau = min(T(M(\mathbf{s})))$

let $\mathbf{s}' = \mathbf{s}' \cup \mathbf{s}$

} (52)

Notice that the time stamp is tracking the output events rather than the input events now.

It is fairly common in discrete-event simulators to disallow sources, requiring them instead to be implemented using feedback loops like that in figure 11. We assume for simplicity in the sequel that this is the case.

4.4 OPERATIONAL SEMANTICS

Procedure (48) can now be used as a basis for an operational semantics for a network of discrete-event processes. Suppose there are N signals and M actors with firing functions $f_1, \ldots, f_M$, with non-empty input index sets $I_1, \ldots, I_M$ and output index sets $O_1, \ldots, O_M$. Let $\mathbf{s} \in S^N$ denote the events initially present (note that there must be some to get things started in this semantics). The procedure is

$$
\begin{array}{l}
\text{while } (\mathbf{s} \neq \Lambda_N) \; \{ \\
\quad \mathbf{s}' = \Lambda_N \\
\quad \text{let } \tau = min(T(M(\mathbf{s}))) \\
\quad \text{for each } i \in \{1, \ldots, M\} \; \{ \\
\quad\quad \text{let } (\mathbf{s}, f_i) = f_i(\pi_{I_i}(\mathbf{s}(\tau)), \tau) \\
\quad\quad \text{let } \mathbf{s}' = \mathbf{s}' \cup A_{O_i, N}(\mathbf{s}) \\
\quad \} \\
\quad \text{let } \mathbf{s} = (\mathbf{s} - \mathbf{s}(\tau)) \cup \mathbf{s}' \\
\}
\end{array}
\tag{53}
$$

where

$$A_{O_i, N}(\hat{\mathbf{s}}) = [p_1, \ldots, p_N] \in S^N \tag{54}$$

where

$$p_j = \begin{cases} \lambda \text{ if } j \notin O_i = [o_1, \ldots, o_m] \\ \pi_{[h]}(\hat{\mathbf{s}}) \text{ if } o_h = j \end{cases} . \tag{55}$$

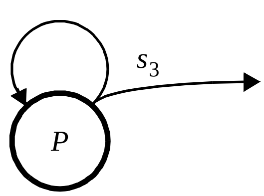


FIGURE 11. A self loop used to realize a source.

Although notationally difficult, the operator $A_{O_i, N}(\mathbf{s})$ is conceptually simple. It change the sort of $\mathbf{s}$, augmenting it to dimension N by inserting empty signals into the tuple.

4.5 DISCUSSION

If the firing functions or a finite power are all delta causal, then the operational semantics matches the denotational semantics (the simulation procedure does “the right thing”). If there is a firing function or interconnection of firing functions that is only strictly causal in a feedback loop, then Zeno signals become a possibility. In this case, a simulator may fail to progress beyond a finite point in time. If there is a firing function that is only causal in a feedback loop, then we have no assurance of their being a denotational solution, much less an operational one.

In the latter case, lessons could be taken from the synchronous languages [4] to define a fixed-point semantics at each time stamp. This could be done with functional signals and firing functions that are monotonic over a Scott order on the event values. Efficient procedures exist for finding such fixed points at run time [9], so this is by no means a far-fetched approach.

5. Conclusions

We have given a formal framework for a class of models of real-time systems based on tagging events with the time at which they occur. The framework supports answering questions of compositionality and correctness of an operational semantics.

Discrete-event models are popular and intuitive, since events must occur at a particular time. If we accept that time is uniform (neglecting relativistic effects), then our model reflects the global ordering of events intrinsic in an interleaving semantics. 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][14][21][24]. 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). Timed models based on branching time and partial orders may be more appropriate [23][14][11][17]. Some preliminary steps have been taken by Mathews towards unifying partial order methods with metric space methods like the ones used here [18][19].

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].

6. Acknowledgments

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laborarory, the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

7. References

[1] S. Abramsky, S. J. Gay, and R. Nagarajan, “Interaction Categories and the Foundations of Typed Concurrent Programming,” In: *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*, (M. Broy, ed.), NATO ASI Series F, Springer-Verlag, 1995.

[2] R. Alur and T. A. Henzinger, “Real-Time System = Discrete System + Clock Variables,” Prof. of the First AMAST Workshop on Real Time, in *Theories and Experiences for Real-Time System Development* (T. Rus and C. Rattray, eds.) AMAST Series in Computing 2, World Scientific, 1994, pp. 1-29.

[3] A. Benveniste, “Compositional and Uniform Modeling of Hybrid Systems,” to appear, *IEEE Tr. on Automatic Control*, 1998.

[4] 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.

[5] G. Birkhoff and S. MacLane, A survey of modern algebra, 4th ed., Macmillan, New York, 1977.

[6] M. Broy, “Functional Specification of Time Sensitive Communicating Systems,” in *Programming and Methematical Method*, M. Broy, ed., NATO ASI Series, Springer-Verlag, 1992.

[7] V. Bryant, *Metric Spaces*, Cambridge University Press, 1985.

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

[9] S. A. Edwards, “The Specification and Execution of Heterogeneous Synchronous Reactive Systems,” **Ph.D. thesis**, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31.
(http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/)

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

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

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

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

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

[15] E. A. Lee and A. Sangiovanni-Vincentelli, “A Denotational Framework for Comparing Models of Computation,” ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997. (http://ptolemy.eecs.berkeley.edu/papers/97/denotational/)

[16] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

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

[18] S.G. Matthews, “An Extensional Treatment of Lazy Dataflow Deadlock,” *Theoretical Computer Science*, Vol. 151, pp. 195-205, 1995.

[19] S.G. Matthews, “Partial Metric Topology,” in *General Topology & its Applications*, eds. S. Andima et al, Proc. of the 8th Summer Conf., Queen’s College (1992), Annals of the New York Academy of Science, Vol. 728 (New York Academy of Sci., 1994), pp. 183-197.

[20] 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.

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

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

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

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

[25] G. M. Reed and A. W. Roscoe, “Metric Spaces as Models for Real-Time Concurrency,” Mathematical Foundations of Programming Language Semantics, 3rd Workshop, New Orleans, Springer-Verlag LNCS 298 (1987) pp.331-343.

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