Invited paper, in Proceedings of the International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS), Paris, France, September 25-27, 2006 published as LNCS XXXX, Springer.

Concurrent Semantics without the Notions of State or State Transitions

Edward A. Lee

University of California, Berkeley, eal@eecs.berkeley.edu

Abstract. This paper argues that basing the semantics of concurrent systems on the notions of state and state transitions is neither advisable nor necessary. The tendency to do this is deeply rooted in our notions of computation, but these roots have proved problematic in concurrent software in general, where they have led to such poor programming practice as threads. I review approaches (some of which have been around for some time) to the semantics of concurrent programs that rely on neither state nor state transitions. Specifically, these approaches rely on a broadened notion of computation consisting of interacting components. The semantics of a concurrent compositions of such components generally reduces to a fixed point problem. Two families of fixed point problems have emerged, one based on metric spaces and their generalizations, and the other based on domain theories. The purpose of this paper is to argue for these approaches over those based on transition systems, which require the notion of state.

1 Introduction

In this paper, I argue that basing the semantics of concurrent systems on the notion of state and state transitions is problematic. The resulting models often have unnecessary nondeterminism in the sense that the nondeterminism is an accident of the choice of modeling technique, rather than an intrinsic or interesting property of the system under study. This complicates analysis and impedes understanding. Moreover, such models fail to be compositional, in the sense that deterministic transition systems, when composed, often become nondeterministic, and that this nondeterminism reveals few if any insights about the system. The result is poor descriptions of the composite behavior.

Introducing time into concurrent models can help, but it either reduces the problem to sequential computation or it relies on a fictional abstraction, Newtonian universal time. In Newtonian universal time, every component in a distributed system shares a common notion of time. Networked computational systems have no mechanism for establishing this common notion of time. Considerable effort is required to establish even an approximate common notion of time [26], and since it is necessarily approximate, the resulting models will be either inaccurate or unnecessarily nondeterministic. When a strongly common notion of time is assumed in the semantics, as in for example discrete-event systems, then distributed or parallel execution becomes a major challenge [19, 58].

There is a strong draw, however, towards using the notions of state and state transitions in semantics. These notions are deeply rooted in our understanding of computation. Programming languages are based on these concepts (and as a consequence, adapt poorly to concurrent computation [33]). Even our foundational notions of semantics are wedded to state. In [56], for example, Winskel explains denotational semantics to be a description of commands as functions mapping a syntax into a function that maps state into state. Winskel observes, in fact, that this notation appears to not be powerful enough for parallelism and fairness (presumably because of its focus on a single global state). When it comes to modeling parallelism, in [56] Winskel focuses on Hoare's CSP [25] and Milner's CCS [43]. He gives both in terms of labeled transition systems, where the labels can include input/output operations. But labeled transition systems are intrinsically based on the notion of state. Coupling two deterministic non-interacting processes (a trivial composition) under either CSP or CCS semantics will yield a nondeterministic semantic model. This nondeterminism contributes nothing to the understanding of the system. It reflects the uninteresting and inconsequential multiplicity of possible interleavings of independent actions.

The focus on transition systems follows naturally from our core imperative notions of computation. There has been, of course, considerable exploration of concurrent alternatives to imperative models of computation. For example, in [20], Goldin et al. describe a persistent Turing machine (PTM), which has three tapes: input, working, and output. It continually processes data from the input, producing outputs. Networks of these can interact. A "universal PTM" can simulate the actions of any PTM. Goldin et al. argue that any "sequential interactive computation" (which has only a causality restriction) can be represented by a PTM. PTM's have stream and actor semantics, and are intrinsically concurrent. But they are not compositional. Consider again two non-interacting PTSs. Without synchronization between these, the resulting composition is not usefully modeled as a PTM. Similarly, in [22], Gössler and Sifakis argue eloquently for a separation of behavior models from interaction models. But "behavior" is again given as transition systems. Once again, without imposing strong synchronization, a composition of behaviors is not usefully modeled as a behavior.

Of course, one can introduce synchronization to alleviate these problems. The synchronous languages [8] take a particularly strong stance on this, where concurrent computations are simultaneous and instantaneous, and at each "tick" of a global "clock" variables have a value given denotationally as a fixed point. This model is clean and compositional, but has proved notoriously difficult to implement in parallel or distributed systems. This has led to a focus on "global asynchronous, locally synchronous" (GALS) models [9]. These are, by definition, not compositional, since compositions of asynchronously interacting components cannot be again made synchronous without introducing unnecessary nondeterminism.

Even our notions of equivalence between systems are deeply connected to the notion of state. In [44], Milner originated the idea of observational equivalence as mutual simulation (and bisimulation). Simulation and bisimulation are relations on states, and hence become much less useful when system state is not a well-defined concept. To be sure, the formalisms apply, but only at the expense of the introduction of nondeterminism that is likely not intrinsic to the system being modeled.

To get a sense of just how deeply rooted the concept of state and state transitions are, consider next the very notion of computation.

2 Computation as Transformation of State

Let $\mathbb{N} = \{0, 1, 2, \cdots\}$ represent the natural numbers. Let $B = \{0, 1\}$ be the set of binary digits. Let B^* be the set of all finite sequences of bits, and

$$B^\omega = (\mathbb{N} \to B)$$

be the set of all infinite sequences of bits (each of which is a function that maps $\mathbb{N}$ into B). Following [15], let $B^{**} = B^* \cup B^\omega$. We will use B^{**} to represent the state of a computing machine, its (potentially infinite) inputs, and its (potentially infinite) outputs. Let

$$Q = (B^{**} \rightharpoonup B^{**})$$

denote the set of all partial functions with domain and codomain B^{**}.

An *imperative machine* (A, c) is a finite set $A \subset Q$ of *atomic actions* and a *control function* $c\colon B^{**} \to \mathbb{N}$. The set A represents the atomic actions (typically instructions) of the machine and the function c represents how instructions are sequenced. We assume that A contains one *halt* instruction $h \in A$ with the property that

$$\forall\, b \in B^{**}, \quad h(b) = b.$$

That is, the halt instruction leaves the state unchanged.

A *sequential program* of length $m \in \mathbb{N}$ is a function

$$p\colon \mathbb{N} \to A$$

where

$$\forall\, n \geq m, \quad p(n) = h.$$

That is, a sequential program is a finite sequence of instructions tailed by an infinite sequence of halt instructions. Note that the set of all sequential programs, which we denote P, is a countably infinite set.

An execution of this program is a *thread*. It begins with an initial $b_0 \in B^{**}$, which represents the initial state of the machine and the (potentially infinite) input, and for all $n \in \mathbb{N}$,

$$b_{n+1} = p(c(b_n))(b_n). \tag{1}$$

Here, $c(b_n)$ provides the index into the program p for the next instruction $p(c(b_n))$. That instruction is applied to the state b_n to get the next state b_{n+1}. If for any $n \in \mathbb{N}$ $\quad c(b_n) \geq m$, then $p(c(b_n)) = h$ and the program halts in state b_n (that is, the state henceforth never changes). If for all initial states $b_0 \in B$ a program p halts, then p defines a total function in Q. If a program p halts for some $b_0 \in B$, then it defines a partial function in Q.[1]

We now get to the core appeal that sequential programs have. Given a program and an initial state, the sequence given by (1) is defined. If the sequence halts, then the function computed by the program is defined. Any two programs p and p' can be compared. They are equivalent if they compute the same partial function. That is, they are equivalent if they halt for the same initial states, and for such initial states, their final state is the same.[2] Such a theory of equivalence is essential for any useful formalism. But in extending this theory of equivalence to concurrent systems, we are tempted to expose the internal sequence of state transformations, which proves to be an enormous mistake. This mistake underlies the technology of multithreaded programming, on which most concurrent software is built.

The essential and appealing properties of programs are lost when multiple threads are composed. Consider two programs p_1 and p_2 that execute concurrently in a multithreaded fashion. What we mean by this is that (1) is replaced by

$$b_{n+1} = p_i(c(b_n))(b_n) \quad i \in \{1, 2\}. \tag{2}$$

At each step n, either program may provide the next (atomic) action. Consider now whether we have a useful theory of equivalence. That is, given a pair of multithreaded programs (p_1, p_2) and another pair (p'_1, p'_2), when are these two pairs equivalent? A reasonable extension of the basic theory defines them to be equivalent if *all interleavings* halt for the same initial state and yield the same final state. The enormous number of possible interleavings makes it extremely difficult to reason about such equivalence except in trivial cases (where, for example, the state B^{**} is partitioned so that the two programs are unaffected by each others' partition). Even in such trivial cases, we need to extend the semantics in some way to explicitly talk about isolation of state.

Even worse, given two programs p and p' that are equivalent when executed according to (1), if they are executed in a multithreaded environment, we can no longer conclude that they are equivalent. In fact, we have to know about all other threads that might execute (something that may not itself be well defined), and we would have to analyze all possible interleavings. We conclude that with threads, there is no useful theory of equivalence.

[1] Note that a classic result in computing is now evident. It is easy to show that Q is not a countable set. (Even the subset of Q of constant functions is not countable, since B^{**} itself is not countable. This can be easily demonstrated using Cantor's diagonal argument.) Since the set of all finite programs P is countable, we can conclude that not all functions in Q can be given by finite programs. That is, any sequential machine has limited expressiveness. Turing and Church [55] demonstrated that many choices of sequential machines (A, c) result in programs P that can give exactly the same subset of Q. This subset is called the *effectively computable functions*.

[2] In this classical theory, programs that do not halt are all equivalent. This creates serious problems when applying the theory of computation to embedded software, where useful programs do not halt [31].

This problem shows up in practical situations. Building non-trivial multithreaded program with predictable behavior is notoriously difficult [33]. Moreover, implementing a multithreaded model of computation is extremely difficult. Witness, for example, the deep subtleties with the Java memory model (see for example [48] and [21]), where even astonishingly trivial programs produce considerable debate about their possible behaviors.

The core abstraction of computation given by (1), on which all widely-used programming languages are built, emphasizes deterministic composition of deterministic components. The actions are deterministic and their sequential composition is deterministic. Sequential execution is, semantically, function composition, a neat, simple model where deterministic components compose into deterministic results.

Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism. We have, of course, developed tools to assist in the pruning. Semaphores, monitors, and more modern overlays on threads (see [33] for a discussion of these) offer the programmer ever more effective pruning.

A model based on nondeterministic interleavings of state transformations is not a useful model. We should build neither programming techniques nor semantics on it. In [33], I have discussed alternative programming techniques. Here I discuss alternative approaches to semantics. As with with the programming techniques, these alternative approaches are not new (mostly). The purpose of this paper is to argue their superiority, not to introduce new techniques.

3 Tagged Signal Model

Instead of functions of the form

$$f\colon B^{**} \to B^{**}$$

concurrent computation can be given in terms of functions of the form

$$f\colon (\mathcal{T} \to B^{**}) \to (\mathcal{T} \to B^{**}), \tag{3}$$

where $(\mathcal{T} \to B^{**})$ is the set of functions with domain $\mathcal{T}$ and codomain B^{**}. In the above, $\mathcal{T}$ is a partially or totally ordered set of *tags*, where the ordering can represent time, causality, or more abstract dependency relations. A computation viewed in this way maps an evolving bit pattern into an evolving bit pattern. This formulation is based on the "tagged signal model" [36], and it has been shown adaptable to many concurrent models of computation [9, 12, 38]. The mathematical structure of $\mathcal{T}$ can be highly variable, depending on the concurrency model, and can model tight synchronization (as in the synchronous languages) and loose synchronization (as in stream processing).

The tagged signal model is similar in objectives to the coalgebraic formalism of abstract behavior types in [4], interaction categories [1], and interaction semantics [52]. As with all three of these, the tagged signal model seeks to model a variety of interaction styles between concurrent components, without focusing on their state. Components may have state, and may execute through state transformations, but this is an irrelevant implementation detail. The notion of state is not exposed at the interface, and consequently the same model can be used for compositions of components where the state of the composition is not well-defined.

When computational components are given in the form of (3), we call the components *actors* [32]. Their environment (which can include other actors) provides them with data, and they react and possibly provide the environment with additional data. As suggested by the name, the classical "actor model" [3, 24] is actor-oriented in our sense. In the actor model, components have their own thread of control and interact via message passing. We are using the term "actors" more broadly, inspired the analogy with the physical world, where actors control their own actions. In this sense, the synchronous languages [8], for example, are also actor-oriented. Asynchronous dataflow models are also actor-oriented in our sense, including both Kahn-MacQueen process networks [28], where each component has its own thread of control, and Dennis-style dataflow [17], where components (also called "actors" in the original literature) "fire" in response to the availability of input data. In our conception, however, compositions of actors are also actors. It does not matter whether the composition has a single thread of control or a well-defined "firing."

A number of component architectures that are not commonly considered in software engineering also have an actor-oriented nature and are starting to be used as source languages for embedded software [34, 31]. Discrete-event (DE) systems, for example, are commonly used in circuit design and in modeling and design of communication networks [13, 5]. In DE, components interact via events, which carry data and a time stamp, and reactions are chronologically ordered by time stamp. In continuous-time (CT) models, such as those specified in Simulink (from The MathWorks) and Modelica [54], components interact via (semantically) continuous-time signals, and execution engines approximate the continuous-time semantics with discrete traces.

When computational components are given in the form of (3), concurrent composition can take the form of ordinary function composition. By contrast, under the imperative model, the clean and simple mechanism of function composition is only applicable to sequential composition.

For components of form of (3), the domain and range of the function f are themselves functions, with form $x\colon \mathcal{T} \to B^{**}$. We call these functions "signals." They represent a potentially infinite evolving bit pattern. To build a useful semantic model for general concurrent systems, however, we need to broaden this simplistic model. For one, the model is much more usable if components have multiple inputs and outputs. This is easy to accomplish.

First, we generalize the notion of a signal. An *event* is a pair (t, v), where $t \in \mathcal{T}$ and $v \in \mathcal{V}$, a set of values. The set of events is $\mathcal{E} = \mathcal{T} \times \mathcal{V}$. A *signal* s is a subset of $\mathcal{E}$. So the set of all signals is $\mathcal{P}(\mathcal{E})$, the power set. A *functional signal* s is a partial function from $\mathcal{T}$ to $\mathcal{V}$, meaning that if $(t, v_1) \in s$ and $(t, v_2) \in s$, then $v_1 = v_2$. We denote the set of all functional signals by $S = (\mathcal{T} \rightharpoonup \mathcal{V})$. We will only consider functional signals here, so when we say "signal" we mean "functional signal."

Next we generalize the notion of an actor. An actor is associated with a set of *ports*. Actors receive and produce events on ports. Thus, a port is associated with a signal, which is a set of events. Given a set P of ports, a *behavior* is a function

$$\sigma \colon P \to S.$$

That is, a behavior for a set of ports assigns to each port $p \in P$ a signal $\sigma(p) \in S$.

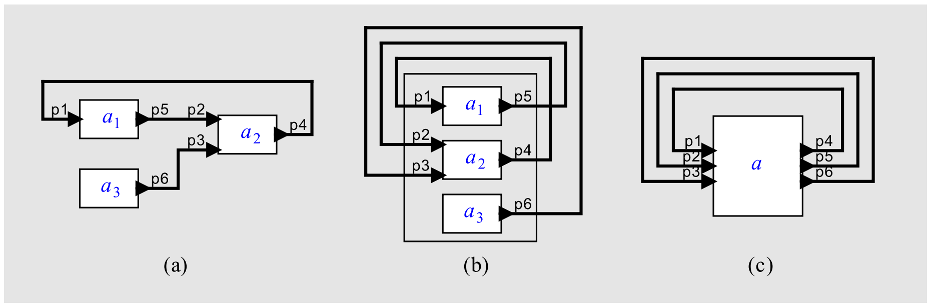

Fig. 1. A composition of three actors and its interpretation as a feedback system.

Three actors with ports are depicted graphically in figure 1(a). The actors are represented by rectangular boxes and the ports by small black triangles. At each port, there is a signal. Note that nothing in our formalism so far constrains the set $\mathcal{V}$ of values. In particular, there is nothing to keep us from including in $\mathcal{V}$ representations of actors themselves, which would yield a higher-order formalism. In particular, since a signal represents an evolving set of values, such a higher-order formalism supports evolving actor composition structures.

An *actor* a with ports P_a is a set of behaviors,

$$a \subset (P_a \to S).$$

That is, an actor can be viewed as constraints on the signals at its ports. A signal $s \in S$ at port p is said to satisfy an actor a if there is a behavior $\sigma \in a$ such that $s = \sigma(p)$.

A *connector* c between ports P_c is also a set of behaviors,

$$c \subset (P_c \to S),$$

but with the constraint that for each behavior $\sigma \in c$, there is a signal $s \in S$ such that

$$\forall\ p \in P_c, \quad \sigma(p) = s.$$

That is, a connector asserts that the signals at a set of ports are identical. In figure 1(a), the connectors are represented as wires between ports.

Given two sets of behaviors, a with ports P_a and b with ports P_b, the *composition behavior set* is the intersection, defined as

$$a \wedge b \subset ((P_a \cup P_b) \to S),$$

where

$$a \wedge b = \{\sigma \mid \sigma \downarrow_{P_a} \in a \text{ and } \sigma \downarrow_{P_b} \in b\},$$

where $\sigma \downarrow_P$ denotes the restriction of σ to the subset P of ports.

A set A of actors (each of which is a set of behaviors) and a set C of connectors (each of which is also a set of behaviors) defines a *composite actor*. The composite actor is defined to be the composition behavior set of the actors A and connectors C. Figure 1(a) is such a composite actor.

In many such concurrent formalisms, ports are either inputs or outputs to an actor but not both. Consider an actor a with ports $P_a = P_i \cup P_o$, where P_i are the input ports and P_o are the output ports. In figure 1(a), triangles pointing into the actor rectangles represent input ports, and triangles pointing out from the rectangles represent output ports. The actor is said to be *functional* if

$$\forall\ \sigma_1, \sigma_2 \in a, \quad (\sigma_1 \downarrow_{P_i} = \sigma_2 \downarrow_{P_i}) \Rightarrow (\sigma_1 \downarrow_{P_o} = \sigma_2 \downarrow_{P_o}).$$

Such an actor can be viewed as a function from input signals to output signals. Specifically, given a functional actor a with input ports P_i and output ports P_o, we can define a function

$$F_a \colon (P_i \to S) \rightharpoonup (P_o \to S). \tag{4}$$

This function is total if any signal at an input port satisfies the actor. Otherwise it is partial. If the function is total, the actor is said to be *receptive*. A connector, of course, is functional and receptive.

An actor with no input ports (only output ports) is functional if and only if its behavior set is a singleton set. That is, it has only one behavior. An actor with no output ports (only input ports) is always functional.

A composition of actors and connectors is itself an actor. The input ports of such a composition can include any input port of a component actor that does not share a connection with an output port of a component actor. If the composition has no input ports, it is said to be *closed*. Figure 1(a) is a closed composition.

A composition is *determinate* if it is functional. Note that now a composition is determinate if and only if its observable behavior is determinate. There is no accidental nondeterminism due to the choice of modeling technique.

A key question in many such concurrent formalisms is, given a set of total functional actors and connectors, is the composition functional and total? This translates into the question of existence and uniqueness of behaviors of compositions. It determines whether a composition is determinate and whether it is receptive. This is a question of semantics.

4 Semantics

First, we observe that the semantics of any network of functional actors is the signal values. This reduces to a fixed point problem. In particular, any composition of functional actors can be restructured as a single actor with feedback connections. The composition in figure 1(a) can be redrawn as shown in figure 1(b), which suggests the abstraction shown in figure 1(c). It is easy to see that any diagram of this type can be redrawn in this way and abstracted to a single actor with the same number of input and output ports, with each output port connected back to a corresponding input port.

It is also easy to see that if actors a_1, a_2, and a_3 in figure 1(b) are functional, then the composite actor a in figure 1(c) is functional. Let F_a denote the function of the form (4) giving the behaviors of a. Then the behavior of the feedback composition in figure 1(c) is a function

$$f \colon \{p1, p2, p3\} \to S$$

that is a fixed point of F_a. That is,

$$F_a(f) = f.$$

A key question, of course, is whether such a fixed point exists (does the composition have a behavior?) and whether it is unique (is the composition determinate?).

For some models of computation, a unique semantics is assigned even when there are multiple fixed points by associating a partial order with the set S of signals and choosing the least or greatest fixed point. For dataflow models [27, 11, 35], a prefix order on the signals turns the set of signals into a complete partial order (CPO). Given such a CPO, we define the semantics of the diagram to be the least fixed point. The least fixed point is assured of existing if a is monotonic, and a constructive procedure exists for finding that least fixed point if a is also continuous (in the prefix order) [27]. It is easy to show that if a_1, a_2, and a_3 in figure 1(b) are continuous, then so is a in figure 1(c). Hence, continuity is a property that composes easily.

This approach builds on domain theory [2], developed for the denotational semantics of programming languages [56, 51]. But unlike many semantics efforts that focus on system state and transformation of that state, it focuses on concurrent interactions, and does not even assume that there is a well-defined notion of "system state."

Note that even when a unique fixed point exists and can be found, the result may not be desirable. Suppose for example that in figure 1(c) F_a is the identity function. This function is continuous, so under the prefix order, the least fixed point exists and can be found constructively. In fact, the least fixed point assigns to each port the empty signal. We interpret this result as deadlock, because an execution of the program cannot proceed beyond the empty signals. Whether such a deadlock condition exists is much harder to determine than whether the composition yields a continuous function. In fact, it can be shown that in general this question is undecidable for dataflow models [35]. An approach that analyzes such networks for such difficulties is given in [37] and [59]. This approach defines *causality interfaces* for actors and gives an algebra for composing these interfaces.

In synchronous languages, the problem of existence and uniqueness reduces to determining existence and uniqueness at each tick of the global clock, rather than over the entire execution. In this case, we can use a flat CPO (rather than one based on a prefix order) and similarly assign a least fixed point semantics [50, 10, 18]. In this CPO, all monotonic functions are continuous. As in the dataflow case, continuity composes easily, but does not tell the whole story. In particular, the least fixed point may include the bottom $\perp$ of the CPO, which represents an "unknown" value. When this occurs, the program is said to have a *causality loop*. Whether a program has a causality loop can be difficult to determine in general, but one can define a conservative "constructive semantics" that enables a finite static analysis of programs to determine whether a program has a causality loop [10]. One can further define a language that needs to know very little about the actors to determine whether such a causality loop exists [18]. The causality interfaces of [37] and [59] can again be used to analyze these models for causality loops.

When $\mathcal{T}$ represents time, it is customary to define semantics using metric space approaches [6, 49, 57, 7, 16, 29]. In such formulations, causality plays a central role. Causality intuitively defines the dependence that outputs from a concurrent component have on inputs to that component. In metric-space formulations, causal components are modeled as contracting functions in the metric space, conveniently enabling us to leverage powerful fixed-point theorems.

In [40], Liu, Matsikoudis, and myself recently showed that the standard metric-space formulation excessively restricts the models of time that can be used. In particular, it cannot handle super-dense time [41, 42], used in hardware description languages, hybrid systems modeling, and distributed discrete-event models. Super-dense time is essential to cleanly model simultaneous events without unnecessary nondeterminism. Moreover, the metric-space approaches do not handle well finite time lines, and time with no origin. Moreover, if we admit continuous-time and mixed signals (essential for hybrid systems modeling) or certain Zeno signals, then causality is no longer equivalent to its formalization in terms of contracting functions. In [40], Liu et al. give an alternative semantic framework using a generalized ultrametric [46] that overcomes these limitations. The existence and uniqueness of behaviors for such systems comes from the fixed-point theorem of [47], but this theorem gives no constructive method to compute the fixed point. In [14] we go a step further, and for the particular case of super-dense time, we define *petrics*, a generalization of metrics, which we use to generalize the Banach fixed-point theorem to provide a constructive fixed-point theorem.

Domain-theoretic approaches, it turns out, can also be applied to timed systems, obviating the need for a metric space. The following approach is described in [39], which is based on [38]. This approach constrains the tagged signal model described above in a subtle but important way. Specifically, it assumes that the tag set is a poset $(\mathcal{T}, \leq)$, and that a signal is a partial function defined on a down set of $\mathcal{T}$ (a similar restriction is made in [45]). A subset $\mathcal{T}'$ of $\mathcal{T}$ is a down set if for all $t' \in \mathcal{T}'$ and $t \in \mathcal{T}$, $t \leq t'$ implies $t \in \mathcal{T}'$. Down sets are also called initial segments in the literature [23]. Under this approach, a signal $s : \mathcal{T} \rightharpoonup V$ is a partial function from $\mathcal{T}$ to V such that $\mathrm{dom}(s)$ is a down set of $\mathcal{T}$, where $\mathrm{dom}(s)$ is the subset of $\mathcal{T}$ on which s is defined.

This constraint leads to a natural prefix order on signals. For any $s_1, s_2 \in \mathcal{S}$, s_1 is a prefix of s_2, denoted by $s_1 \sqsubseteq s_2$, if and only if $\mathrm{dom}(s_1) \subseteq \mathrm{dom}(s_2)$, and $s_1(t) = s_2(t),\ \forall t \in \mathrm{dom}(s_1)$. That is, a signal s_1 is a prefix of another signal s_2 if the graph of the function s_1 is a subset of the graph of the function s_2. The prefix order on signals is a natural generalization of the prefix order on strings or sequences, and the extension order on partial functions [53]. It is shown in [39] that existence and uniqueness of behaviors are ensured by continuity with respect to this prefix order. Causality conditions are also defined that ensure liveness and freedom from Zeno conditions. In this formulation, causality does not require a metric and can embrace a wide variety of models of time in timed concurrent systems.

In summary, approaches to semantics based on the tagged signal model, rather than on the notion of state and state transitions, appear to accomplish the key objectives of studies semantics. Since they do not explicitly depend on the notion of state, they do not introduce unnecessary and uninteresting nondeterminism due to irrelevant interleavings of state transitions.

5 Conclusion

I have argued that basing the semantics of concurrent systems on the notions of state and state transitions is neither advisable nor necessary. The tendency to do this is deeply rooted in our notions of computation, but these roots have also proved problematic in concurrent programming, where they have led to such poor programming foundations as threads. I have outlined some approaches to the semantics of concurrent programs that rely on neither state nor state transitions.

6 Acknowledgements

This paper describes work that is part of the Ptolemy project, and reflects the ideas of many contributors. Most notably, I would like to acknowledge Adam Cataldo, Elaine Cheong, Thomas Huining Feng, Xiaojun Liu, Elefterios Matsikoudis, Yang Zhao, Haiyang Zheng, Gang Zhou, and Rachel Zhou. The Ptolemy project is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Microsoft, and Toyota.

References

1. S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In M. Broy, editor, *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, NATO ASI Series F. Springer-Verlag, 1995.
2. S. Abramsky and A. Jung. Domain theory. In *Handbook of logic in computer science (vol. 3): semantic structures*, pages 1–168. Oxford University Press, Oxford, UK, 1995.
3. G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–140, 1990.
4. F. Arbab. Abstract behavior types : A foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, 2005.
5. J. R. Armstrong and F. G. Gray. *VHDL Design Representation and Synthesis.* Prentice-Hall, second edition, 2000.
6. A. Arnold and M. Nivat. Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *Fundamenta Informaticae*, 11(2):181–205, 1980.
7. C. Baier and M. E. Majster-Cederbaum. Denotational semantics in the cpo and metric approach. *Theoretical Computer Science*, 135(2):171–220, 1994.
8. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
9. A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *EMSOFT*. Springer, 2003.
10. G. Berry. *The Constructive Semantics of Pure Esterel.* Book Draft, 1996.
11. M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258:99–129, 2001.
12. J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Notes on agent algebras. Technical Report UCB/ERL M03/38, University of California, November 2003.
13. C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis.* Irwin, 1993.
14. A. Cataldo, E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan, 2006.
15. B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order.* Cambridge University Press, 1990.
16. J. W. de Bakker and E. P. de Vink. Denotational models for programming languages: Applications of banachs fixed point theorem. *Topology and its Applications*, 85:35–52, 1998.
17. J. B. Dennis. First version data flow procedure language. Technical Report MAC TM61, MIT Laboratory for Computer Science, 1974.
18. S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1), 2003.
19. G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer-Verlag, 2001.
20. D. Goldin, S. Smolka, P. Attie, and E. Sonderegger. Turing machines, transition systems, and interaction. *Information and Computation*, 194(2):101–128, 2004.

21. A. Gontmakher and A. Schuster. Java consistency: nonoperational characterizations for Java memory behavior. *ACM Trans. Comput. Syst.*, 18(4):333–386, 2000.
22. G. Gssler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55, 2005.
23. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–37. 1994.
24. C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323363, 1977.
25. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.
26. S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.
27. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.
28. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*, pages 993–998. North-Holland Publishing Co., 1977.
29. E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.
30. E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.
31. E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, 2003.
32. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
33. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
34. E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
35. E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.
36. E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT), Satellite to CONCUR*, San Francisco, CA, 2005.
37. X. Liu. Semantic foundation of the tagged signal model. Phd thesis, EECS Department, University of California, December 20 2005.
38. X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report EECS-2006-67, UC Berkeley, May 18 2006.
39. X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In *CONCUR*, Bonn, Germany, 2006. LNCS.
40. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.
41. Z. Manna and A. Pnueli. Verifying hybrid systems. *Hybrid Systems*, pages 4–35, 1992.
42. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
43. R. Milner. Elements of interaction. *Communications of the ACM*, 36:78–89, 1993.

44. H. Naundorf. Strictly causal functions have a unique fixed point. *Theoretical Computer Science*, 238(1-2):483–488, 2000.
45. S. Priess-Crampe and P. Ribenboim. Generalized ultrametric spaces I. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 66:55–73, 1996.
46. S. Priess-Crampe and P. Ribenboim. Fixed point and attractor theorems for ultrametric spaces. *Forum Mathematicum*, 12:53–64, 2000.
47. W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98, San Francisco, California, United States, 1999. ACM Press.
48. G. M. Reed and A. W. Roscoe. Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pages 331–343, London, UK, 1988.
49. K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Washington DC, USA, 2004.
50. J. E. Stoy. *Denotational Semantics.* MIT Press, Cambridge, MA, 1977.
51. C. L. Talcott. Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 1996.
52. P. Taylor. *Practical Foundations of Mathematics.* Cambridge University Press, 1999.
53. M. M. Tiller. *Introduction to Physical Modeling with Modelica.* Kluwer Academic Publishers, 2001.
54. A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.
55. G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, MA, USA, 1993.
56. R. K. Yates. Networks of real-time processes. In E. Best, editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag, 1993.
57. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation.* Academic Press, 2nd edition, 2000.
58. Y. Zhou and E. A. Lee. A causality interface for deadlock analysis in dataflow. Technical Report UCB/EECS-2006-51, EECS Department, UC Berkeley, May 12 2006.