Causality Interfaces for Actor Networks

Ye Zhou
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-148
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-148.html

November 16, 2006

Copyright © 2006, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This paper describes work that is part of the Ptolemy project, which 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, Infineon, Microsoft, National Instruments, and Toyota.

Causality Interfaces for Actor Networks

YE ZHOU and EDWARD A. LEE
UC Berkeley

We consider concurrent models of computation where "actors" (components that are in charge of their own actions) communicate by exchanging messages. The interfaces of actors principally consist of "ports," which mediate the exchange of messages. Actor-oriented architectures contrast with and complement object-oriented models by emphasizing the exchange of data between concurrent components rather than transformation of state. Examples of such models of computation include the classical actor model, synchronous languages, dataflow models, process networks, and discrete-event models. Many experimental and production languages used to design embedded systems are actor oriented and based on one of these models of computation. Many of these models of computation benefit considerably from having access to causality information about the components. This paper augments the interfaces of such components to include such causality information. It shows how this causality information can be algebraically composed so that compositions of components acquire causality interfaces that are inferred from their components and the interconnections. We illustrate the use of these causality interfaces to statically analyze timed models and synchronous language compositions for causality loops and dataflow models for deadlock. We also show that that causality analysis only needs to be performed for one port in each directed communication cycle, and we give a conservative approximation technique for handling dynamically changing causality properties.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Reliability, Theory, Verification

Additional Key Words and Phrases: Actors, Behavioral types, Causality, Dataflow, Deadlock, Discrete-event models, Interfaces, Synchronous languages, Timed systems

1. INTRODUCTION

Although prevailing component architecture techniques in software are object oriented, a number of researchers have been advocating a family of complementary approaches that we collectively call *actor oriented* [Lee 2003]. In practice (as realized in UML, C++, Java and C#), the components of object-oriented design interact principally through transfer of control (method calls) and transformation of state. The components are passive, and things get done to them, much like physical "objects" from which the name arises.[1] "Actors" react to stimulus provided by

[1]So called "active objects" add to the basic object-oriented model threads, but as a component technology, active objects are semantically weak compared to the actor-oriented techniques we

Authors' address: {zhouye, eal}@eecs.berkeley.edu, Department of Electrical Engineering and Computer Sciences University of California, Berkeley Berkeley, CA 94720, USA.
This paper describes work that is part of the Ptolemy project, which 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, Infineon, Microsoft, National Instruments, and Toyota.

their environment, which can include other actors. As a component architecture, the difference is one of emphasis and interpretation: objects interact principally through transfer of control, whereas actors interact principally through exchange of data. An immediate consequence is that actor-oriented designs tend to be highly concurrent.

Several distinct research communities fall within this broad framework. As suggested by the name, the classical "actor model" [Agha 1990; Hewitt 1977] falls into this category. 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.[2] In fact, several other communities use similar ways of defining components. In the synchronous/reactive languages [Benveniste and Berry 1991], which are principally used for embedded software, components react at ticks of a global clock, rather than reacting when other components invoke their methods. In the synchronous language Esterel [Berry and Gonthier 1992], components exchange data through variables whose values are (semantically) determined by solving fixed point equations. The Lustre [Halbwachs et al. 1991] and Signal [Benveniste and Guernic 1990] languages have a more dataflow flavor, but they have similar semantics. Asynchronous dataflow models are also actor-oriented in our sense, including Kahn-MacQueen process networks [Kahn and MacQueen 1977], where each component has its own thread of control, extensions to nondeterministic systems [de Kock et al. 2000], and Dennis-style dataflow [Dennis 1974]. In dataflow, components (which are also called "actors" in the original literature) "fire" in response to the availability of input data. Process networks have also been used for embedded system design [de Kock et al. 2000].

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 [Lee et al. 2003; Lee 2002]. Discrete-event (DE) systems, for example, are commonly used in circuit design and in modeling and design of communication networks [Cassandras 1993; Armstrong and Gray 2000]. 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 [Tiller 2001], components interact via (semantically) continuous-time signals, and execution engines approximate the continuous-time semantics with discrete traces.

Surrounding the actor-oriented approach are a number of semantic formalisms that complement traditional Turing-Church theories of computation by emphasizing interaction of concurrent components rather than sequential transformation of data. These include stream formalisms [Kahn 1974; Broy and Stefanescu 2001; Rutten 2005], discrete-event formalisms [Yates 1993; Lee 1999], and semantics for continuous-time models [Lee and Zheng 2005]. A few formalisms are rich enough to embrace a significant variety of actor-oriented models, including interaction

describe.

[2]The term "agents" is equally good, but we avoid it because in the mind of many researchers, agents include a notion of mobility, which is orthogonal to interaction and irrelevant to our current discussion.

categories [Abramsky et al. 1995], behavioral types [Lee and Xiong 2004; Arbab 2005], interaction semantics [Talcott 1996], and the tagged-signal model [Lee and Sangiovanni-Vincentelli 1998].

Some software frameworks also embrace a multiplicity of actor-oriented component architectures, including Reo [Arbab 2004], Ptolemy II [Eker et al. 2003], PECOS [Winter et al. 2002], and Metropolis [Göessler and Sangiovanni-Vincentelli 2002]. Finally, a number of researchers have argued strongly for separation between the semantics of functionality (what is computed) from that of interaction between components [Buck et al. 1994; Keutzer et al. 2000; Göessler and Sifakis 2005; Wegner et al. 2005].

In the object-oriented world, a great deal of time and effort has gone into defining interfaces for components. Relatively little of this has been done for actor-oriented models. In [Xiong 2002] Xiong extends some basic object-oriented typing concepts to actor-oriented designs by clarifying subtyping relationships when interfaces consist of ports (which represent senders or receivers of messages) rather than methods. This is extended further in [Lee and Neuendorffer 2004] with inheritance mechanisms.

In this paper, we give an *interface theory* [de Alfaro and Henzinger 2001], similar in spirit to resource interfaces [Chakrabarti et al. 2003] and behavioral type systems [Lee and Xiong 2004]. Our theory captures *causality* properties of actor-oriented designs.[3] Causality properties reflect in the interface the dependence that particular outputs have on particular inputs. In this paper, we build a rather specialized theory (of causality only) that is orthogonal to other semantic properties. Our work can be applied to many concurrent semantics such as that of the synchronous languages, discrete-event models, continuous-time models, and dataflow models.

Following [de Alfaro and Henzinger 2001] and common practice in object-oriented design, an actor can have more than one interface. We consider actors with input and output ports, where each input port receives zero or more messages, and the actor reacts to these messages by producing messages on the output ports. One interface of the actor defines the number of ports, gives the ports names or some other identity, and constrains the data types of the messages handled by the port [Xiong 2002]. Another interface of the actor defines behavioral properties of the port, such as whether it requires input messages to be present in order to react [Lee and Xiong 2004].

A causality interface declares the dependency that output messages have on input messages. How this information is used depends on the model of computation. In this paper, we focus on several models of computation with least fixed point semantics. In stream-oriented dataflow models, our causality interface can be used to analyze compositions of actors for deadlock [Broy and Stefanescu 2001; Lee and Parks 1995]. In discrete-event models, it can be used to ensure deterministic processing of simultaneous events, and to identify causality loops [Lee 1999; Yates 1993]. In synchronous languages, it can be used to identify whether a combinational cycle has a reactive and deterministic behavior for all possible combinations of input values [Schneider et al. 2004; Berry 1996; Edwards and Lee 2003]. In all three cases, the causality properties of components determine whether a particular composition

[3]A preliminary form of causality interfaces is given in [Lee et al. 2005].

is live.

2. ACTORS AND THEIR COMPOSITION

We begin by giving a formal structure for actors that is sufficiently expressive to embrace all of the models of computation of interest. We then discuss briefly syntaxes that are amenable to actor models and define the visual syntax used in this paper. We then review fixed point semantics, which is used in quite a few models of computation and serves as the semantic foundation for our causality interfaces.

2.1 The Tagged Signal Model

The tagged-signal model [Lee and Sangiovanni-Vincentelli 1998] provides a formal framework for considering and comparing actor-oriented models of computation. It is similar in objectives to the coalgebraic formalism of abstract behavior types in [Arbab 2005], interaction categories [Abramsky et al. 1995], and interaction semantics [Talcott 1996]. As with all three of these, the tagged signal model seeks to model a variety of interaction styles between concurrent components.

Interactions between actors are tagged signals, which are sets of (tag, value) pairs. The tags come from a partially or totally ordered set $\mathcal{T}$, the structure of which depends on the model of computation. For example, in a simple (perhaps overly simple) discrete-event model of computation, $\mathcal{T}$ might be equal to the set of non-negative real numbers with their ordinary numerical ordering, representing time. In such a DE model, interactions between actors consist of (time, value) pairs.

Formally, 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 $\mathcal{E} = \mathcal{T} \times \mathcal{V}$. Following [Liu 2005; Liu and Lee 2006], a *signal* s is a function from a down set of $\mathcal{T}$ to $\mathcal{V}$. A down set $T \subseteq \mathcal{T}$ is a subset that satisfies

$$t \in T \Rightarrow \forall\ \tau \in \mathcal{T} \text{ where } \tau \leq t,\ \tau \in T.$$

Such a down set T where a signal s is defined is also called the *preimage* of s, written as $\text{dom}(s)$. A signal is called *complete* if $\text{dom}(s) = \mathcal{T}$. We use $\mathcal{D}(\mathcal{T})$ to denote the set of down sets of $\mathcal{T}$. The following property comes from [Liu 2005].

PROPERTY 1. *Let $\mathcal{D}(\mathcal{T})$ be the set of all down sets of a partially-ordered set $\mathcal{T}$.*

(1) $(\mathcal{D}(\mathcal{T}), \subseteq)$ *is a* complete partial order *(CPO).*
(2) $(\mathcal{D}(\mathcal{T}), \subseteq)$ *is a complete lattice.*
(3) If $\mathcal{T}$ is totally-ordered, then $(\mathcal{D}(\mathcal{T}), \subseteq)$ is also totally-ordered.

We assume for simplicity one tag set $\mathcal{T}$ and one value set $\mathcal{V}$ for all signals, but nothing significant changes in our formalism if distinct signals have different tag and value sets. We write the set of all signals $\mathcal{S}$.

The graph of a signal $s \in \mathcal{S}$ is

$$\text{graph}(s) = \{(t, v) \in \mathcal{T} \times \mathcal{V} \mid s(t) \text{ is defined and } s(t) = v\}.$$

We define a *prefix order* on signals as follows. Given $s_1, s_2 \in \mathcal{S}$, $s_1 \sqsubseteq s_2$ (read s_1 is a prefix of s_2) if $\text{graph}(s_1) \subseteq \text{graph}(s_2)$. $(\mathcal{S}, \sqsubseteq)$ is also a CPO [Liu 2005]. The least element of $\mathcal{S}$ is the empty signal, denoted $\perp$. The set of N-tuples of signals is $\mathcal{S}^N$. The prefix order extends naturally to $\mathcal{S}^N$, and $\mathcal{S}^N$ is also a CPO.

Actors receive and produce signals on *ports*. An *actor* a with N ports is a subset of $\mathcal{S}^N$. A particular $s \in \mathcal{S}^N$ is said to satisfy the actor if $s \in a$, and s is called a *behavior* of the actor. Thus an actor is a set of possible behaviors. An actor therefore asserts constraints on the signals at its ports.

A *connector* c between ports P_c is a particulary simple actor where signals at each port $p \in P_c$ are constrained to be identical. The ports in P_c are said to be *connected*.

A set A of actors and a set C of connectors defines a *composite actor*. The composite actor is defined to be the intersection of all possible behaviors of the actors A and connectors C [Lee and Sangiovanni-Vincentelli 1998].

In many actor-oriented formalisms, ports are either inputs or outputs to an actor but not both. Consider an actor $a \subseteq \mathcal{S}^N$ where $I \subseteq \{1, ..., N\}$ denotes the indices of the input ports, and $O \subseteq \{1, ..., N\}$ denotes the indices of the output ports. We assume that $I \cup O = \{1, ..., N\}$ and $I \cap O = \emptyset$. Given a signal tuple $s \in a$, we define $\pi_I(s)$ to be the projection of s on a's input ports, and $\pi_O(s)$ on output ports. The actor is said to be *functional* if

$$\forall\ s, s' \in a, \quad \pi_I(s) = \pi_I(s') \Rightarrow \pi_O(s) = \pi_O(s').$$

Such an actor can be viewed as a function from input signals to output signals. Specifically, given a functional actor a with $|I|$ input ports and $|O|$ output ports, we can define an *actor function* with the form

$$F_a \colon \mathcal{S}^{|I|} \to \mathcal{S}^{|O|}, \tag{1}$$

where $|\cdot|$ denotes the size of a set. When it creates no confusion, we make no distinction between the actor a (a set of behaviors) and the actor function F_a.

A *source* actor is an actor with no input ports (only output ports). It is functional if and only if its behavior set is a singleton set. That is, it has only one behavior. A *sink* actor is an actor with no output ports, and it is always functional.

A composite actor is itself an actor. In addition to the set P of ports contained by the composite actor a, the actor may have a set of Q of external ports, where $Q \bigcap P = \emptyset$ (see figure 1). Input ports in Q may be connected to any input port in P that is not already connected. Output ports in Q may be connected to any single output port in P. If the composite actor has no (external) input ports, it is said to be *closed*. Otherwise it is *open*.

2.2 Syntax

Actor-oriented languages can be either self-contained programming languages (e.g. Esterel, Lustre, LabVIEW) or coordination languages (e.g. Manifold [Papadopoulos et al. 2006], Simulink, Ptolemy II). In the former case, the "atomic actors" are the language primitives. In the latter case, the "atomic actors" are defined in a host language that is typically not actor oriented (but is often object oriented). Actor-oriented design is amenable to either textual syntaxes, which resemble those of more traditional computer programs, and visual syntaxes, with "boxes" representing actors and "wires" representing connections. The synchronous languages Esterel, Lustre, and Signal, for example, have principally textual syntaxes, although recently visual syntaxes for some of them have started to catch on. Ports and connectors are syntactically represented in these languages by variable names. Using

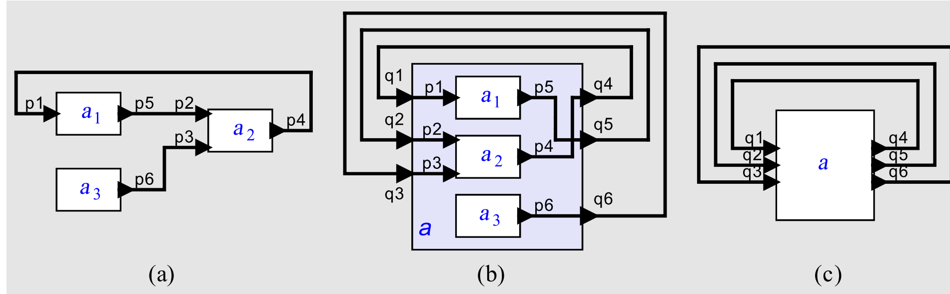

Fig. 1. A composition of three actors and its interpretation as a feedback system. $P = \{p1, p2, p3, p4, p5, p6\}$ is the set of ports contained by the composite actor a. $Q = \{q1, q2, q3, q4, q5, q6\}$ is the set of external ports of a.

the same variable name in two modules implicitly defines ports for those modules and a connection between those ports. Visual syntaxes are more explicit about this architecture. Examples with visual syntaxes include Simulink, LabVIEW, and Ptolemy II.

A visual syntax for a simple three-actor composition is shown in figure 1(a). Here, the actors are rendered as boxes, the ports as triangles, and the connectors as wires between ports. The ports pointing into the boxes are input ports and the ports pointing out of the boxes are output ports. A textual syntax for the same composition might associate a language primitive or a user-defined module with each of the boxes and a variable name with each of the wires.

The composition in figure 1(a) is closed. In figure 1(b), we have added a level of hierarchy by creating an open composite actor a with external ports $\{q_1, q_2, ..., q_6\}$. In figure 1(c), the internal structure of the composite actor is hidden. Using the techniques introduced in this paper, we are able to do that without losing essential causality information of composite actor a.

In fact, any network of actors can be converted to an equivalent hierarchical network, where the composite actor internally has no directed cycles, like that in figure 1(c). A constructive procedure that performs this conversion is easy to develop. Just create one input port and one output port for each signal in the original network. E.g., in figure 1(a), the signal going from $p5$ to $p2$ induces ports $q5$ and $q2$ in figure 1(b) and (c). Then connect the output port providing the signal value ($p5$ in this example) to the new output port ($q5$), and connect the new input port ($q2$) to any input ports that observe the signal ($p2$). This can be done for any network, always resulting in a structure like that in figure 1(c).

2.3 Fixed Point Semantics

It is 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 actor function for actor a. Assuming the component actors are functional, it has the form

$$F_a : \mathcal{S}^3 \to \mathcal{S}^3.$$

The feedback connectors in figure 1(c) require the signals at the input ports of a to be the same as the signals at its outputs. Thus the behavior of the feedback

composition in figure 1(c) is $s \in \mathcal{S}^3$ that is a fixed point of F_a. That is,

$$F_a(s) = s.$$

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?). In quite a few models of computation, including synchronous language compositions, timed models and dataflow models, we define the semantics of the diagram to be the least fixed point (least in the prefix order), if it exists. The least fixed point is assured of existing if F_a is monotonic (order preserving), and a constructive procedure exists for finding that least fixed point if F_a is also (Scott) continuous (in the prefix order) [Davey and Priestly 1990]. It is easy to show that if a_1, a_2, and a_3 in figure 1(b) have continuous actor functions, then so does a in figure 1(c). Continuity is a property that composes easily.

However, 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 wish to ensure that for a particular network of actors, if all sources of data are complete ($\forall$ input signal s, $\text{dom}(s) = \mathcal{T}$), then all signals in the network are complete. A network that satisfies this requirement is said to be *live*.

Whether such a liveness condition exists may be harder to determine than whether the composition yields a continuous function. In fact, Buck showed in [Buck 1993] that boolean dataflow is Turing complete, and therefore liveness is undecidable for boolean dataflow models. It follows that in general this question is undecidable since boolean dataflow is a special case. The causality interfaces we define here provide necessary and sufficient conditions for the liveness condition. Due to the fundamental undecidability, our necessary and sufficient conditions cannot always be statically checked. But we will show that for some concurrent models of computations, they can always be checked.

3. DEPENDENCY ALGEBRA

In this section, we introduce the *dependency algebra* $(D, \leq, \oplus, \otimes)$. The dependency set D is a partially ordered set with two binary operations $\oplus$ and $\otimes$ that satisfy the axioms given below. The elements of D are called *dependencies*, which represent the dependency relations between ports.

First, we require that the operators $\oplus$ and $\otimes$ be associative,

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3), \tag{2}$$

$$\forall\ d_1, d_2, d_3 \in D, \quad (d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3). \tag{3}$$

Second, we require that $\oplus$ (but not $\otimes$) be commutative,

$$\forall\ d_1, d_2 \in D, \quad d_1 \oplus d_2 = d_2 \oplus d_1, \tag{4}$$

and idempotent,

$$\forall\ d \in D, \quad d \oplus d = d. \tag{5}$$

In addition, we require an additive and a multiplicative identity, called $\mathbf{0}$ and $\mathbf{1}$,

respectively, that satisfy:

$$\begin{array}{lll} \exists\, \mathbf{0} \in D \text{ such that} & \forall\, d \in D, & d \oplus \mathbf{0} = d \\ \exists\, \mathbf{1} \in D \text{ such that} & \forall\, d \in D, & d \otimes \mathbf{1} = \mathbf{1} \otimes d = d \\ \forall\, d \in D, \quad d \otimes \mathbf{0} = \mathbf{0}. & & \end{array}$$

The ordering relation $\leq$ on the set D is a partial order, meaning, as usual,

$$\begin{array}{ll} \forall\, d \in D, & d \leq d \\ \forall\, d_1, d_2 \in D, & d_1 \leq d_2 \text{ and } d_2 \leq d_1 \;\Rightarrow\; d_1 = d_2 \\ \forall\, d_1, d_2, d_3 \in D, & d_1 \leq d_2 \text{ and } d_2 \leq d_3 \;\Rightarrow\; d_1 \leq d_3. \end{array}$$

We use $d_1 < d_2$ to mean $(d_1 \leq d_2) \wedge (d_1 \neq d_2)$.

Finally, a key axiom of D relates the operators and the order as follows.

$$\forall\, d_1, d_2 \in D, \quad d_1 \oplus d_2 \leq d_1. \tag{6}$$

Using these axioms, we get the following property:

PROPERTY 2. *The additive identity* $\mathbf{0}$ *is the top element of the partial order* $(D, \leq)$.

PROOF. Using (6), let $d_1 = \mathbf{0}$, from which we conclude

$$\forall\, d_2 \in D, \quad d_2 \leq \mathbf{0}.$$

□

4. CAUSALITY INTERFACES

4.1 Definition

A *causality interface* for an actor a with input ports P_i and outports P_o is a function

$$\delta \colon P_i \times P_o \to D, \tag{7}$$

where D is a dependency algebra as defined in the previous section. Ports connected by connectors will always have causality interface $\mathbf{1}$, and lack of dependency between ports will be modeled with causality interface $\mathbf{0}$.

4.2 Causality Interfaces for Least Fixed Point Semantics

How these causality interfaces are used depends on the semantics of the model computation. In this subsection, we give a dependency algebra that can be used for models of computation with least fixed point semantics. This includes synchronous languages, timed models and dataflow models.

We define the dependency set D to be a set of functions:

$$D = (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})), \tag{8}$$

where $(X \to Y)$ denotes the set of total functions with domain X and range contained by Y. Recall from section 2.1 that $\mathcal{D}(\mathcal{T})$ is the set of down sets of the tag set $\mathcal{T}$. With appropriate choices for an order and $\oplus$ and $\otimes$ operators, the set D forms a dependency algebra.

We define the order relation $\leq$ such that $\forall\, d_1, d_2 \in D$, $d_1 \leq d_2$ if $\forall\, T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subseteq d_2(T)$.

The $\oplus$ operation computes the greatest lower bound of two elements in D. I.e., $\forall\ d_1, d_2 \in D$, the function $(d_1 \oplus d_2)\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is defined by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (d_1 \oplus d_2)(T) = d_1(T) \cap d_2(T). \tag{9}$$

To see that (9) computes the greatest lower bound of d_1 and d_2, first note $\forall\ T \in \mathcal{D}(\mathcal{T})$, $(d_1 \oplus d_2)(T) \subseteq d_1(T)$ and $(d_1 \oplus d_2)(T) \subseteq d_2(T)$. Therefore $(d_1 \oplus d_2) \leq d_1$ and $(d_1 \oplus d_2) \leq d_2$, so $(d_1 \oplus d_2)$ is a lower bound of $\{d_1, d_2\}$. Now consider another lower bound d of $\{d_1, d_2\}$. Thus, $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d(T) \subseteq d_1(T)$ and $d(T) \subseteq d_2(T)$. Therefore $\forall\ T \in \mathcal{D}(\mathcal{T})$, $d(T) \subseteq d_1(T) \cap d_2(T) = (d_1 \oplus d_2)(T)$. This leads to $d \leq (d_1 \oplus d_2)$. Therefore $(d_1 \oplus d_2)$ is the greatest lower bound of $\{d_1, d_2\}$.

The $\otimes$ operator is function composition. I.e., $\forall\ d_1, d_2 \in D$, the function $(d_1 \otimes d_2)\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is defined by

$$d_1 \otimes d_2 = d_2 \circ d_1$$

or

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad (d_1 \otimes d_2)(T) = d_2(d_1(T)).$$

The additive identity $\mathbf{0}$ is the *top function*, $d_\top\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad d_\top(T) = \mathcal{T}.$$

The multiplicative identity $\mathbf{1}$ is the *identity function*, $d_I\colon \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad d_I(T) = T.$$

With these definitions, the dependency set (8) satisfies all of the axioms described in section 3.

Recall that actors respond to events at input ports by producing events at output ports. For input port p and output port p' of an actor a, the causality interface $\delta_a(p, p')$ is interpreted to mean that a signal defined on $T \in \mathcal{D}(\mathcal{T})$ at port p can affect the signal defined on $(\delta_a(p, p'))(T)$ at port p'. That is, there is a causal relationship between the portion of the input signal defined on T and the portion of the output signal defined on $(\delta_a(p, p'))(T)$. To make this precise, first consider an actor a with one input port p, one output port p', and actor function $F_a : \mathcal{S} \to \mathcal{S}$. Then, $\delta_a(p, p')$ is the largest function such that $\forall\ s_1, s_2 \in \mathcal{S},\ \forall\ T \in \mathcal{D}(\mathcal{T})$,

$$s_1 \downarrow T = s_2 \downarrow T \quad \Rightarrow \quad F_a(s_1) \downarrow (\delta_a(p, p'))(T) = F_a(s_2) \downarrow (\delta_a(p, p'))(T),$$

where $s \downarrow T$ means the function s is restricted to a subset T of $\mathcal{T}$ (recall that a signal is a function from a down set of $\mathcal{T}$ to $\mathcal{V}$). We can generalize this to actors with multiple input and output ports. The concept is similarly simple, although the notation is more complex. As in section 2.1, let $a \subseteq \mathcal{S}^N$ be an actor with N ports. Let $I \subseteq \{1, ..., N\}$ and $O \subseteq \{1, ..., N\}$ denote the indices of the input and output ports, where $I \cap O = \emptyset$ and $I \cup O = \{1, ..., N\}$. Let $F_a : \mathcal{S}^{|I|} \to \mathcal{S}^{|O|}$ denote the actor function. Consider an $s \in \mathcal{S}^N$, and $\forall\ i \in \{1, ..., N\}$, let s_i be the projection of s on port i. For any $i \in I$ and $o \in O$, the causality interface $\delta(p_i, p_o)$ is the largest function such that $\forall\ T \in \mathcal{D}(\mathcal{T})$,

$$\begin{aligned}&\forall\ s, s' \in a, \quad s_i \downarrow T = s'_i \downarrow T, \quad \text{and } \forall\ j \in I, j \neq i, s_j = s'_j,\\ &\Rightarrow s_o \downarrow (\delta_a(p_i, p_o))(T) = s'_o \downarrow (\delta_a(p_i, p_o))(T).\end{aligned}$$

That is, if the inputs of s and s' are same at port i on the down set T and same on all other input ports, then the output signals at port o are same on the down set $(\delta(p_i, p_o))(T)$.

Recall that a functional source actor is an actor with no input ports and exactly one behavior. To give it a causality interface, we define a fictional *absent input port* ε, and for any output port p_o, $\delta_a(\varepsilon, p_o)$ is given by

$$\forall\, T \in \mathcal{D}(\mathcal{T}), \quad (\delta_a(\varepsilon, p_o))(T) = \mathrm{dom}(s),$$

where s is the unique signal that satisfies the actor at p_o. If s is complete, $\mathrm{dom}(s) = \mathcal{T}$, then $\delta_a(\varepsilon, p_o) = d_\top$.

A sink actor is one with no output ports. Similarly, we define the causality interface of a sink actor to be a function that maps an input port p_i of the actor and a fictional *absent output port* to the *bottom function*. I.e.,

$$\delta_a(p_i, \varepsilon) = d_\bot,$$

where $d_\bot(T) = \emptyset, \forall\, T \in \mathcal{D}(\mathcal{T})$.

The causality interface for a connector is simply the multiplicative identity $\mathbf{1} = d_I$.

A causality interface $\delta(p, p')$ is said to satisfy the *liveness condition* if $\delta(p, p')(\mathcal{T}) = \mathcal{T}$. An actor is said to be live if all of its causality interfaces satisfy the liveness condition. I.e., a complete input yields a complete output. We say that a composition of actors is live if, given complete signals on all external inputs, then all signals that satisfy the composition are complete. For a live composition, every causality interface is live, except those of sink actors.

A (functional) actor a with input ports P_i is said to be *monotonic* (or order preserving) if

$$\forall\, s_1, s_2 \in \mathcal{S}^{|P_i|}, \quad s_1 \sqsubseteq s_2 \;\Rightarrow\; F_a(s_1) \sqsubseteq F_a(s_2),$$

where F_a is the actor function of a. Intuitively, monotonicity says that if the input signal is extended to a larger down set, the output signal can only be changed by extending it to a larger down set. Thus we have the following property:

PROPERTY 3. *Let p be an input port and p' be an output port of a monotonic actor a. Then $\delta_a(p, p')$ is monotonic.*

For the purpose of this paper, we assume all actors are (Scott) continuous, a stronger property than monotonicity. A *chain* in a CPO is a totally ordered subset of the CPO. In a CPO, every chain C has a least upper bound, written $\bigvee C$ (this is what makes the CPO "complete"). An actor a is said to be (Scott) *continuous* if for all chains $C \subseteq \mathcal{S}^{|P_i|}$, the *least upper bound* $\bigvee F_a(C)$ exists and

$$F_a(\bigvee C) = \bigvee F_a(C).$$

Here it is understood that $F_a(C) = \{F_a(s) \mid s \in C\}$.

Since the domains of the signals in a chain C also form a chain in $\mathcal{D}(\mathcal{T})$ (a CPO with set inclusion order), it is easy to see that the following property holds:

PROPERTY 4. *Let p be an input port and p' be an output port of a (Scott) continuous actor a. Then $\delta_a(p, p')$ is (Scott) continuous.*

Continuity implies monotonicity [Davey and Priestly 1990], so it follows that the causality interfaces of a continuous actor are also monotonic.

We will establish necessary and sufficient conditions for a composition of actors to be live. To do this, we need some technical results for functions on down sets. First, we define a new relation $\prec$ on D as follows. $\forall\, d_1, d_2 \in D$, $d_1 \prec d_2$ if

(1) $d_1 \neq d_2$, and,

(2) for each $T \in \mathcal{D}(\mathcal{T})$, $d_1(T) \subset d_2(T) \vee d_1(T) = d_2(T) = \mathcal{T}$, where $\subset$ denotes a strict subset.

The relation $\prec$ is a strict partial order, meaning, as usual, that it is

- irreflexive: $\forall\, d \in D$, $d \not\prec d$
- antisymmetric: $\forall\, d_1, d_2 \in D$, $d_1 \prec d_2 \Rightarrow d_2 \not\prec d_1$
- transitive: $\forall\, d_1, d_2, d_3 \in D$, $d_1 \prec d_2$ and $d_2 \prec d_3 \Rightarrow d_1 \prec d_3$.

It is easy to see irreflexivity and transitivity hold for the $\prec$ relation. To see antisymmetry, consider two functions $d_1, d_2 \in D$. If $d_1 \prec d_2$, then $\exists\, T \in \mathcal{D}(\mathcal{T})$ such that $d_1(T) \subset d_2(T)$. Therefore, $d_2 \not\prec d_1$.

The following theorem and corollary will prove useful in this paper.

Theorem 1. *If $d : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is a continuous function, then*

(1) d has a least fixed point T_0, given by $\bigwedge\{T \in \mathcal{D}(\mathcal{T}) \mid d(T) \subseteq T\}$.

(2) If $d_I \prec d$, where $d_I = \mathbf{1}$ is the multiplicative identity, then the least fixed point of d is $\mathcal{T}$.

Proof. Note that $\mathcal{D}(\mathcal{T})$ is a complete lattice. Part (1) comes directly from the Knaster-Tarski fixed point theorem [Davey and Priestly 1990].

Part (2): If $d_I \prec d$, then $\forall\, T \in \mathcal{D}(\mathcal{T})$, $T \neq \mathcal{T}$, $d_I(T) = T \neq \mathcal{T}$. Therefore, $T = d_I(T) \subset d(T)$. Since $d(\mathcal{T}) \subseteq \mathcal{T} = d_I(\mathcal{T})$, then $d(\mathcal{T}) = d_I(\mathcal{T}) = \mathcal{T}$. Therefore, $T_0 = \bigwedge\{T \in \mathcal{D}(\mathcal{T}) \mid d(T) \subseteq T\} = \mathcal{T}$. $\square$

Corollary 1. *If $d : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ is a continuous function, where $\mathcal{D}(\mathcal{T})$ is totally-ordered, then the least fixed point of d is $\mathcal{T}$ if and only if $d_I \prec d$.*

Proof. The backward implication is identical to Theorem 1. We now prove the forward implication. Since the least fixed point of d is $\bigwedge\{T \in \mathcal{D}(\mathcal{T}) \mid d(T) \subseteq T\} = \mathcal{T}$, then $\forall\, T \subset \mathcal{T}$, $d(T) \not\subseteq T$. Since $\mathcal{D}(\mathcal{T})$ is totally-ordered, $d(T) \not\subseteq T = d_I(T) \Leftrightarrow d_I(T) \subset d(T)$. Since $\mathcal{T}$ is the least fixed point of d, this means $d(\mathcal{T}) = \mathcal{T} = d_I(\mathcal{T})$. I.e., $d_I \prec d$. $\square$

5. COMPOSITION OF CAUSALITY INTERFACES

Given a set A of actors, a set C of connectors, and the causality interfaces for the actors and the connectors, we can determine the causality interfaces of the composition and whether the composition is live. To do this, we form a *dependency graph* of ports, and observe that the paths between ports traverse both actors and connectors. We will first discuss feedforward compositions and then deal with feedback compositions.

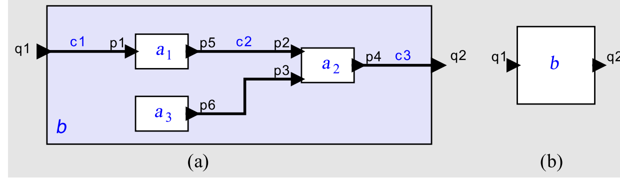

Fig. 2. A feedforward composition.

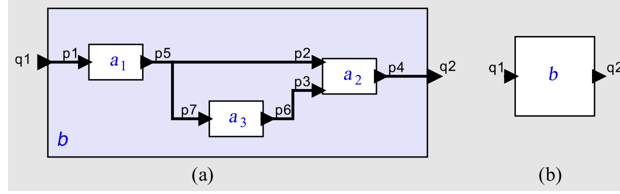

Fig. 3. A feedforward composition with parallel paths.

5.1 Causality Interfaces for Feedforward Compositions

A feedforward system does not have any cycles in its dependency graph. It is easy to see that a feedforward composition of live actors is always live. To determine the causality interfaces of a composite actor abstracting the feedforward composition, we use the $\otimes$ operator for series composition and the $\oplus$ operator for parallel composition. For example, figure 2 shows a feedforward composition, which is abstracted into a single actor b with external input port $q1$ and output port $q2$. To determine the causality interface of actor b, we need to consider all the paths from $q1$ to $q2$, and $\delta_b(q1, q2)$ is given by

$$\delta_b(q1, q2) = \delta_{c1}(q1, p1) \otimes \delta_{a_1}(p1, p5) \otimes \delta_{c2}(p5, p2) \otimes \delta_{a_2}(p2, p4) \otimes \delta_{c3}(p4, q2),$$

where δ_{a_1} and δ_{a_2} are the causality interfaces for actors a_1 and a_2, respectively, and $\delta_{c1}, \delta_{c2}, \delta_{c3}$ are the causality interfaces for connectors $c1, c2, c3$, respectively. Since connectors have causality interface $\mathbf{1}$, the above equation simplifies to

$$\delta_b(q1, q2) = \delta_{a_1}(p1, p5) \otimes \delta_{a_2}(p2, p4).$$

Figure 3 shows a slightly more complicated example, where there are two parallel paths from port $p5$ to port $p4$. We get

$$\delta_b(q1, q2) = \delta_{a_1}(p1, p5) \otimes [\delta_{a_2}(p2, p4) \oplus (\delta_{a_3}(p7, p6) \otimes \delta_{a_2}(p3, p4))], \quad (10)$$

where we have omitted the causality interfaces for connectors.

5.2 Causality Interfaces for Feedback Compositions

The dependency graph of a feedback system contains cyclic paths. Given a cyclic path $c = (p_1, p_2, ..., p_n, p_1)$, where p_i's $(1 \leq i \leq n)$ are ports of the composition, we define the *gain* of c to be

$$g_c = \delta(p_1, p_2) \otimes \delta(p_2, p_3) \otimes ... \otimes \delta(p_n, p_1).$$

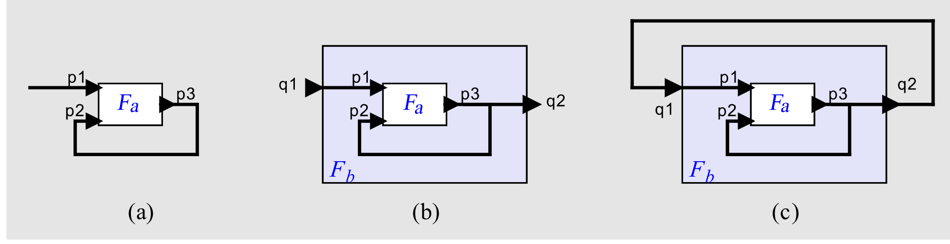

Fig. 4. An open composition with feedback loops.

Note that $c' = (p_i, ..., p_n, p_1, ..., p_i)$ is also a cyclic path, and $g_c \neq g_{c'}$ in general. The ordering of ports of path c' is only a shifted version of that of c. We say that c and c' are two different paths of the same *cycle*.

A *simple cyclic path* is a cyclic path that does not include other cyclic paths. A *simple cycle* is a cycle that does not include other cycles.

How to compose causality interfaces for feedback systems depends on the semantics of the model of computation. In this paper, we focus on models of computation with least fixed point semantics.

We now begin by considering simple cases of feedback systems and build up to the general case. Consider the composition shown in figure 4, where actor a is a feedforward composite actor. From section 5.1, we can determine its causality interfaces and we know it is live if its component actors are live.

The following two lemmas are useful. The first is an adaptation of Lemma 8.10 in [Winskel 1993]:

LEMMA 1. *Consider two CPOs S_1 and S_2, and a continuous function*

$$F_a \colon S_1 \times S_2 \to S_2.$$

For a given $s_1 \in S_1$, we define the function $F_a(s_1) \colon S_2 \to S_2$ such that

$$\forall\ s_2 \in S_2, \quad (F_a(s_1))(s_2) = F_a(s_1, s_2).$$

Then for all $s_1 \in S_1$, $F_a(s_1)$ is continuous.

In the context of figure 4(a), this first lemma tells us that if F_a is continuous, then given an input $s_1 \in \mathcal{S}$ at port $p1$, the function $F_a(s_1)$ from port $p2$ to port $p3$ is continuous. Thus, for each s_1, $F_a(s_1)$ has a unique least fixed point, and that fixed point is $\bigvee\{(F_a(s_1))^n(\perp) \mid n \in \mathbb{N}\}$ [Davey and Priestly 1990].

The second lemma comes from [Liu and Lee 2006]:

LEMMA 2. *Consider two CPOs S_1 and S_2, and a continuous function $F_a \colon S_1 \times S_2 \to S_2$. Define a function $F_b \colon S_1 \to S_2$ such that*

$$\forall\ s_1 \in S_1, \quad F_b(s_1) = \bigvee\{(F_a(s_1))^n(\perp_{S_2}) \mid n \in \mathbb{N}\},$$

where $\perp_{S_2}$ is the least element of S_2. F_b is continuous.

This second lemma tells us that under a least fixed point semantics the composition in figure 4(b) defines a continuous function F_b from port $q1$ to port $q2$.

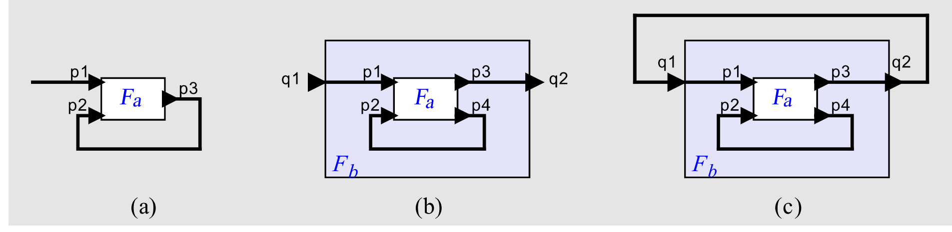

Fig. 5. An open system with a feedback connection that has the structure of figure 1.

We now want to find the causality interface for actor b. Given input signal s_1 at port $p1$ and s_2 at $p2$, where $\mathrm{dom}(s_1) = T_1$ and $\mathrm{dom}(s_2) = T_2$,

$$\mathrm{dom}(F_a(s_1, s_2)) = \delta_a(p1, p3)(T_1) \cap \delta_a(p2, p3)(T_2).$$

For each $T_1 \in \mathcal{D}(\mathcal{T})$, we define a function $f_a(T_1) : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ such that

$$\forall\ T_2 \in \mathcal{D}(\mathcal{T}), \quad (f_a(T_1))(T_2) = \delta_a(p1, p3)(T_1) \cap \delta_a(p2, p3)(T_2).$$

The function $f_a(T_1)$ is continuous and,

$$\begin{aligned} \delta_b(q1, q2)(T_1) = \mathrm{dom}(F_b(s_1)) &= \mathrm{dom}(\bigvee\{(F_a(s_1))^n(\perp) \mid n \in \mathbb{N}\}) \\ &= \bigvee\{\mathrm{dom}((F_a(s_1))^n(\perp)) \mid n \in \mathbb{N}\} \\ &= \bigvee\{(f_a(T_1))^n(\emptyset) \mid n \in \mathbb{N}\} \end{aligned} \tag{11}$$

I.e., $\delta_b(q1, q2)(T_1)$ is the least fixed point of $f_a(T_1)$.

If actor a is live and $\mathbf{1} \prec \delta_a(p2, p3)$, where $\mathbf{1} = d_I$ is the multiplicative identity, then $f_a(\mathcal{T}) = \delta_a(p2, p3)$. Then the least fixed point of $f_a(\mathcal{T})$ is $\mathcal{T}$. Hence actor b is live.

Given the causality interface for actor b, as shown in (11), we now form the nested feedback composition of figure 4(c). We are assured that since b is continuous, this has a unique least fixed point. The composition will be live if $\mathbf{1} \prec \delta_b(q1, q2)$.

Working towards the structure of figure 1, we add an additional output port to actor a in figure 5. We can easily adapt Lemmas 1 and 2 to this situation. Nothing significant changes. We continue to add ports to the actor a, each time creating a nested composite. Since every network can be put into the structure of figure 1(c), we can determine from the causality interfaces of a, whether a composition is live.

If $\mathcal{D}(\mathcal{T})$ is totally-ordered, the following lemma helps us to give the causality interface of feedback composition in a much simpler form than (11).

Lemma 3. *Consider a continuous function $\delta : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$, where $\mathcal{D}(\mathcal{T})$ is totally ordered, and a set $K \in \mathcal{D}(\mathcal{T})$. We define a function $g : \mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T})$ such that*

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad g(T) = K \cap \delta(T).$$

Then g has a least fixed point given by $T_1 = K \cap T_0$, where T_0 is the least fixed point of δ.

Proof.

$$g(T_1) = K \cap \delta(K \cap T_0).$$

Note $\mathcal{D}(\mathcal{T})$ is totally-ordered, then either $K \subset T_0$ or $T_0 \subseteq K$.

(1) If $K \subset T_0$, then $K \cap T_0 = K$ and $K \subset \delta(K)$ (because T_0 is the least fixed point of δ). Therefore, $g(T_1) = K \cap \delta(K) = K = T_1$.

(2) If $T_0 \subseteq K$, then $K \cap T_0 = T_0$. Therefore, $g(T_1) = K \cap \delta(T_0) = K \cap T_0 = T_0 = T_1$.

Therefore T_1 is a fixed point of g. Note that for every down set $T \subset T_1$ where $T_1 = K \cap T_0$, $T \subset K$ and $T \subset T_0$. Since T_0 is the least fixed point of δ, $T \subset \delta(T)$. Therefore, we have

$$T \subset (K \cap \delta(T)),$$

where $K \cap \delta(T) = g(T)$, as defined. I.e., $T \subset g(T)$. Therefore T_1 is the least fixed point of g. $\square$

COROLLARY 2. *Given the composite actor b as shown in figure 4(b), and assuming $\mathcal{D}(\mathcal{T})$ is totally-ordred,*

(1) The causality interface of b is given by

$$\forall\ T \in \mathcal{D}(\mathcal{T}), \quad \delta_b(q1, q2)(T) = \delta_a(p1, p3)(T) \cap T_0$$

where T_0 is the least fixed point of $\delta_a(p2, p3)$.

(2) Actor b is live if and only if actor a is live and $\mathbf{1} \prec \delta_a(p2, p3)$, where $\mathbf{1} = d_I$ is the multiplicative identity.

PROOF. Part (1) comes directly by applying $f_a(T)$ to g in Lemma 3.

Part (2): If $\mathbf{1} \prec \delta_a(p2, p3)$, then $T_0 = \mathcal{T}$. Therefore $\delta_b(q1, q2) = \delta_a(p1, p3)$. Then b is live if a is live.

On the other hand, if b is live, then $\delta_b(q1, q2)(\mathcal{T}) = \delta_a(p1, p3)(\mathcal{T}) \cap T_0 = \mathcal{T}$. Therefore $T_0 = \mathcal{T}$. Due to Corollary 1, this means $\mathbf{1} \prec \delta_a(p2, p3)$. Since $T_0 = \mathcal{T}$, $\delta_a(p1, p3) = \delta_b(q1, q2)$. So actor a is live. $\square$

If $\mathcal{D}(\mathcal{T})$ is totally-ordered, it is easy to prove that distributivity holds for the $(\oplus, \otimes)$ algebra on the subset of monotonic functions of D. I.e., for any monotonic functions $d_1, d_2, d_3 \in D$,

$$d_1 \otimes (d_2 \oplus d_3) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3) \tag{12}$$

$$(d_2 \oplus d_3) \otimes d_1 = (d_2 \otimes d_1) \oplus (d_3 \otimes d_1). \tag{13}$$

This suggests that intersecting cyclic paths can be considered independently. Thus we have reached the most important theorem of this paper:

THEOREM 2. *A finite network of continuous and live actors where the tag set $\mathcal{T}$ is totally-ordered is continuous and live if and only if for every cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

We now give some examples of models of computation where this theorem can be used.

6. APPLICATION TO TIMED SYSTEMS

Timed systems have a tag set $\mathcal{T}$ that is totally-ordered. Since $\mathcal{T}$ is totally-ordered, then $\mathcal{D}(\mathcal{T})$ is also totally-ordered. Examples of timed systems include discrete-event models, continuous-time models and synchronous/reactive (SR) models. For

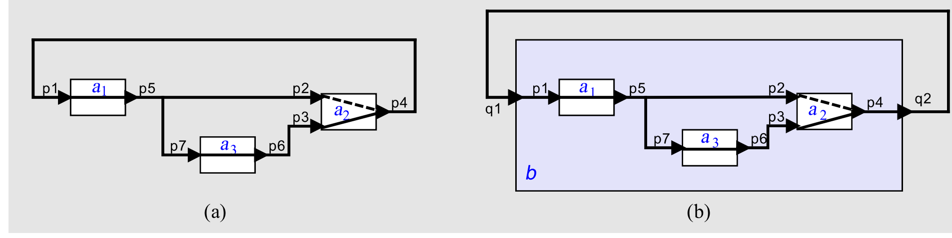

Fig. 6. A timed model with a feedback loop.

discrete-event and continuous-time models, the tag set is $\mathbb{R}_+ = [0, \infty)$, the non-negative reals, or $\mathbb{R}_+ \times \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, ...\}$ is the natural numbers. For SR models, the tag set is $\mathbb{N}$. In this last case, the dependency algebra can be further simplified. It is easy to see that $(\mathcal{D}(\mathbb{N}), \subseteq)$ and $(\mathbb{N}_\infty, \leq)$ are isomorphic, where $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, and $\leq$ is the usual numerical ordering. Therefore, for SR models, the dependency algebra can be simplified to $D = (\mathbb{N}_\infty \to \mathbb{N}_\infty)$.

In all three cases, the tag sets are totally-ordered. Therefore, Theorem 2 of this paper can be easily applied to all three models of computation.

6.1 Causality

Causality is a key concept in timed systems. Intuitively, it means the time of output events cannot be earlier than the time of input events that caused them. Causality interfaces offer a formalization of this intuition.

A port p' is said to have a *causal* dependency on port p if $d_I \leq \delta(p, p')$. A timed actor with at least one input port is said to be causal if every output port has a causal dependency on every input port. A source actor, of course, is always causal. A causal actor is live. Causality implies mononicity but not continuity [Liu 2005].

A port p' is said to have a *strict causal* dependency on port p if $d_I \prec \delta(p, p')$.

Consider again the example in figure 4. From Corollary 2, we know that the causality interface of actor b in figure 4(b) is given by:

$$\forall\, T \in \mathcal{D}(\mathcal{T}), \quad \delta_b(q1, q2)(T) = \delta_a(p1, p3) \cap T_0,$$

where T_0 is the least fixed point of $\delta_a(p2, p3)$. If $d_I \prec \delta_a(p2, p3)$, then $T_0 = \mathcal{T}$, and therefore $\delta_b(q1, q2) = \delta_a(p1, p3)$. Hence actor b is causal (and therefore live) if and only if a is causal. If $d_I \not\prec \delta_a(p2, p3)$, then $T_0 \subset \mathcal{T}$, and therefore b is not live neither causal.

We can continue to add ports to actor a, as described in figure 5, to construct any actor networks. The above analysis on causal dependencies can be adapted easily. Thus we have the following theorem about causality, a stronger property than liveness:

THEOREM 3. *A finite network of continuous and causal timed actors is continuous and causal if and only if for every cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

Consider the example in figure 6(a). We use dashed line to denote a strict causal dependency, and a solid line to denote a causality interface of d_I.

First, we notice that there are two cyclic paths starting from $p1$, namely: $c_1 = (p1, p5, p2, p4, p1)$, and $c_2 = (p1, p5, p7, p6, p3, p4, p1)$, where

$$\begin{aligned} g_{c_1} &= \delta_{a_1}(p1, p5) \otimes \delta_{a_2}(p2, p4) \\ g_{c_2} &= \delta_{a_1}(p1, p5) \otimes \delta_{a_3}(p7, p6) \otimes \delta_{a_2}(p3, p4) \end{aligned}$$

and we want to check whether $\mathbf{1} \prec g_{c_1}$ and $\mathbf{1} \prec g_{c_2}$. (Below we show that checking c_1 and c_2 is sufficient to conclude liveness. Checks on cyclic paths starting from other ports are unnecessary.)

A second way to view this model is to create a hierarchy, as shown in figure 6(b), and there is only one cycle between $q1$ and $q2$. The causality interface of actor b is given in (10), and we want to check whether $\mathbf{1} \prec \delta_b(q1, q2)$. In fact, we find that $\delta_b(q1, q2) = g_{c_1} \oplus g_{c_2}$ (due to distributivity of monotonic functions). Therefore $\mathbf{1} \prec \delta_b(q1, q2) \Leftrightarrow \mathbf{1} \prec g_{c_1} \wedge \mathbf{1} \prec g_{c_2}$. I.e., both approaches check for the same condition. Thus, our technique achieves a measure of modularity, in that details of a composite system can be hidden; it is only necessary to expose the causality interface of the composite.

Using the second approach we get:

$$\begin{aligned} \delta_b(q1, q2) &= \delta_{a_1}(p1, p5) \otimes [\delta_{a_2}(p2, p4) \oplus (\delta_{a_3}(p7, p6) \otimes \delta_{a_2}(p3, p4))] \\ &= d_I \otimes [\delta_{a_2}(p2, p4) \oplus (d_I \otimes d_I)] \\ &= d_I \end{aligned}$$

Thus we conclude that the model has a causality loop and the composition is not live.

In this example, we do not need to know exactly $\delta_{a_2}(p2, p4)$ but whether it is strictly causal, i.e., whether $d_I \prec \delta_{a_2}(p2, p4)$. In other words, given all the component actors are causal, we are interested in whether there is at least one strictly causal interface in every cycle.

7. APPLICATION TO DATAFLOW

In dataflow, the signals are streams of data tokens. Actors execute in response to the availability of data tokens. The tag set $\mathcal{T}$ of dataflow is $\mathbb{N}$. Since $(\mathcal{D}(\mathbb{N}), \subseteq)$ and $(\mathbb{N}_\infty, \leq)$ are isomorphic, we simplify the dependency algebra to $D = (\mathbb{N}_\infty \to \mathbb{N}_\infty)$. For input port p and output p' of an actor a, $\delta_a(p, p') = d$ is interpreted to mean that given n tokens at port p, there will be $d(n)$ tokens at port p'. That is, given an input stream of length n, the output stream has length $(\delta_a(p, p'))(n)$. Note that, in general, $\delta_a(p, p')$ may depend on the input tokens themselves. This fact is the source of expressiveness that leads to undecidability of liveness. However, as we will show, many situations prove decidable.

Since $\mathbb{N}$ is totally-ordered, we have the following theorem:

THEOREM 4. *A finite network of continuous and live dataflow actors is continuous and live if and only if for every cyclic path c in the dependency graph, $\mathbf{1} \prec g_c$, where $\mathbf{1} = d_I$ is the multiplicative identity.*

Wadge [Wadge 1981] uses an element $n \in \mathbb{N}_\infty$ to represent the dependency between ports, where $n_{ij} \in \mathbb{N}_\infty$ means that the first k tokens at the j-th port depend on at most the first $k - n_{ij}$ tokens of the i-th port. However, Wadge's technique is only good for homogeneous synchronous dataflow, where every actor consumes and

produces exactly one token on every port in every firing. Our causality information is captured by a function (rather than a number), which is richer and enough to handle multirate dataflow.

7.1 Decidability

One question that might arise concerns decidability of deadlock. The above theorem gives us necessary and sufficient conditions for a dataflow network to be live. However, deadlock is generally undecidable for dataflow models. These statements are not in conflict. Our necessary and sufficient conditions may not be decidable. In particular, the causality interfaces for some actors, e.g., *boolean select* and *boolean switch* [Buck 1993], are in fact dependent on the data provided to them at the control port. They cannot be statically known by examining the syntactic specification of the dataflow network unless the input stream at the control port can be statically determined. Theorem 4 implies that if for every cyclic path c, $\mathbf{1} \prec g_c$ is decidable, then deadlock is decidable. More precisely, if we can prove for every c, $\mathbf{1} \prec g_c$, then the model is live. If we can prove there exists a cyclic path c such that $\mathbf{1} \not\prec g_c$, then there is at least one (local) deadlock in the model. If we can prove neither of these, then we can draw no conclusion about deadlock.

Certain special cases of the dataflow model of computation make deadlock decidable. For example, in the synchronous dataflow (SDF) model of computation [Lee and Messerschmitt 1987], every actor executes as a sequence of firings, where each firing consumes a fixed, specified number of tokens on each input port, and produces a fixed, specified number of tokens on each output port. In addition, an actor may produce a fixed, specified number of tokens on an output port at initialization. Given an SDF actor a with input port p_i and output port p_o, the causality interface function $\delta_a(p_i, p_o)$ is given by

$$\forall\, n \in \mathbb{N}_\infty, \quad (\delta_a(p_i, p_o))(n) = \begin{cases} \lfloor n/N \rfloor \cdot M + I, & \text{if } n < \infty \\ \infty, & \text{if } n = \infty, \end{cases} \tag{14}$$

where N is the number of tokens consumed at p_i in a firing, M is the number of tokens produced at p_o, and I is the number of initial tokens produced at p_o at initialization. Using this, we get the following theorem.

THEOREM 5. *Deadlock is decidable for synchronous dataflow models with a finite number of actors.*

PROOF. Since distributivity holds for continuous dataflow actors, it is easy to see that the gain of any cyclic path can be written in the form

$$g = \bigoplus\left(\bigotimes \delta_a(p_i, p_o)\right), \tag{15}$$

where each $\delta_a(p_i, p_o)$ is in the form of (14), and the $\otimes$ and $\oplus$ operators operate on a finite number of δ's.

We first note that for each function δ in the form of (14), the following property holds:

$$\forall\, k, r \in \mathbb{N}, \quad \delta(kN + r) = \delta(r) + kM, \tag{16}$$

which means

$$\delta(kN + r) - (kN + r) = \delta(r) - r + k(M - N).$$

Therefore, $\mathbf{1} \prec \delta$ if and only if $N \leq M$ and $\forall\, r \in \{0, 1, ..., N-1\}$, $r < \delta(r)$, which can be determined in finite time. Thus $\mathbf{1} \prec \delta$ is decidable.

Now consider two causality interfaces δ_a and δ_b of some SDF actors, where

$$\forall\, k, r \in \mathbb{N}, \quad \begin{aligned} \delta_a(kN_a + r) &= \delta_a(r) + kM_a \\ \delta_b(kN_b + r) &= \delta_b(r) + kM_b \end{aligned}$$

where we have omitted mention of the ports for notational simplicity. A cascade of δ_a and δ_b would therefore satisfy

$$(\delta_a \otimes \delta_b)(kN_aN_b + r) = (\delta_a \otimes \delta_b)(r) + kM_aM_b,$$

which is also in the form of (16). We can continue to compose any finite number of causality interfaces with the $\otimes$ operator to get an expression of the form $(\otimes\delta)$, where each δ is a causality interface in the form of (14), and $(\otimes\delta)$ satisfies (16). Thus $\mathbf{1} \prec (\otimes\delta)$ is decidable.

Now consider the $\oplus$ operation on two functions δ_1 and δ_2 for which we know whether $\mathbf{1} \prec \delta_1$ and $\mathbf{1} \prec \delta_2$. Since $\oplus$ computes the greatest lower bound,

$$\mathbf{1} \prec (\delta_1 \oplus \delta_2) \Leftrightarrow \mathbf{1} \prec \delta_1 \wedge \mathbf{1} \prec \delta_2.$$

Thus $\mathbf{1} \prec (\delta_1 \oplus \delta_2)$ is decidable. This generalizes easily to any expression of the form of (15) over a finite number of actors. $\square$

In [Lee and Messerschmitt 1987], it is shown that if a synchronous dataflow model is consistent, then deadlock is decidable. In particular, this is shown by following a scheduling procedure that provably terminates. Our theory applies to both consistent and inconsistent SDF models, and hence is more general. Moreover, it is more straightforward to check whether $\mathbf{1} \prec g$ than to execute the scheduling procedure described in [Lee and Messerschmitt 1987].

We now reconsider the example in figure 6 as a dataflow model. Assume all the ports produce and consume one token on each firing of the corresponding actor, and that port $p5$ produces $I \in \mathbb{N}$ initial tokens, and all other ports produce zero initial tokens. We get

$$\begin{aligned} \delta_b(q1, q2) &= \delta_{a_1}(p1, p5) \otimes [\delta_{a_2}(p2, p4) \oplus (\delta_{a_3}(p7, p6) \otimes \delta_{a_2}(p3, p4))] \\ &= (d_I + I) \otimes [d_I \oplus (d_I \otimes d_I)] \\ &= d_I + I \end{aligned}$$

If $I = 0$, then $\mathbf{1} = \delta_b(q1, q2)$, and the model deadlocks. If $I > 0$, then $\mathbf{1} \prec \delta_b(q1, q2)$. The model is live.

This example also shows that our causality interfaces can help in designing a system by properly allocating correct number of initial tokens to prevent deadlock.

7.2 Relationship to Partial Metrics

Matthews uses a metric-space approach to treat deadlock [Matthews 1995]. He defines a partial metric, which is a distance function:

$$f : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}_+,$$

where $\mathcal{S}$ is the set of all sequences and $\mathbb{R}_+$ is the non-negative real numbers. Given two sequences $s_1, s_2 \in \mathcal{S}$,

$$f(s_1, s_2) = 2^{-n},$$

where n is the length of the longest common prefix of s_1 and s_2 (if the two sequences are infinite and identical, $f(s_1, s_2) = 0$). The pair $(\mathcal{S}, f)$ is a complete partial metric space.

We first consider a simple scenario of a continuous dataflow actor a with one input port p_i and one output port p_o and a feedback connection from p_o to p_i. The actor function is F_a and the causality interface is δ_a. According to Theorem 4.1 in [Matthews 1995], this feedback system is deadlock free if F_a is a contraction map in this complete partial metric space, meaning

$$\begin{array}{ll} \exists\ c \in \mathbb{R}_0, \quad 0 \leq c < 1, & \text{such that} \\ \forall\ s_1, s_2 \in \mathcal{S}, & f(F_a(s_1), F_a(s_2)) \leq cf(s_1, s_2). \end{array}$$

Theorem 6. *Let a be a continuous dataflow actor with one input port p_i and one output port p_o. The actor function of a is F_a. Then $\mathbf{1} \prec \delta_a(p_i, p_o) \Leftrightarrow F_a$ is a contraction map in the Matthews partial metric space.*

Proof. Since there is only one relevant causality interface, we abbreviate $\delta_a(p_i, p_o)$ by δ_a (without showing the dependency on the ports). We begin by showing the forward implication.

Given $s_1, s_2 \in \mathcal{S}$, let s be their longest common prefix, and let $n = |s|$ be its length. Then $|F_a(s)| = \delta_a(n) \geq n + 1$. By monotonicity, $F_a(s)$ is a prefix of $F_a(s_1)$ and $F_a(s_2)$. Therefore,

$$f(F_a(s_1), F_a(s_2)) \leq 2^{-\delta_a(n)} \leq 2^{-(n+1)} = \frac{1}{2} \cdot f(s_1, s_2),$$

so F_a is a contraction map.

We next show the backward implication. Consider two signals s_1 and $s_2 \in \mathcal{S}$, where $|s_1| = n < \infty$ and s_1 is a strict prefix of s_2. Therefore, we have,

$$\begin{array}{l} f(s_1, s_2) = 2^{-n}, \\ f(F_a(s_1), F_a(s_2)) = 2^{-\delta_a(n)}. \end{array}$$

If F_a is a contraction map, then,

$$2^{-\delta_a(n)} < 2^{-n}.$$

Since we can arbitrarily choose s_1 (as long as $|s_1|$ is finite), it follows that $\forall\ n \in \mathbb{N}, n < \delta_a(n) \leq \delta_a(\infty)$. This concludes that $\mathbf{1} \prec \delta_a$. $\square$

In Theorem 5.1 in [Matthews 1995], Matthews gives a sufficient condition for liveness for compositions with more than one feedback loop. We can similarly prove that this sufficient condition is equivalent to the condition in Theorem 4 of this paper. Our Theorem 4 shows that it is also a necessary condition for liveness.

8. COMPUTATION

It is stated in Theorem 2 that an actor network where the tag set $\mathcal{T}$ is totally-ordered is live if and only if for every cyclic path c, $\mathbf{1} \prec g_c$. We now ask a more practical question. Do we need to verify $\mathbf{1} \prec g_c$ for every cyclic path c?

Consider a non-simple cyclic path $c = (p_1, ..., p_i, q_1, ..., q_m,\ p_i, ..., p_n, p_1)$. Therefore $c_1 = (p_1, ..., p_i, p_{i+1}, ..., p_n, p_1)$ and $c_2 = (p_i, q_1, ..., q_m, p_i)$ are two cyclic paths.

Let $d_1 = \delta(p_1, p_2) \otimes ... \otimes \delta(p_{i-1}, p_i)$, $d_2 = \delta(p_i, p_{i+1}) \otimes ... \otimes \delta(p_n, p_1)$. Then,

$$g_{c_1} = d_1 \otimes d_2$$
$$g_c = d_1 \otimes g_{c_2} \otimes d_2.$$

If $\mathbf{1} \prec g_{c_1}$ and $\mathbf{1} \prec g_{c_2}$, then, $\mathbf{1} \prec g_{c_1} = d_1 \otimes d_2 \prec d_1 \otimes g_{c_2} \otimes d_2 = g_c$. I.e., checking g_{c_1} and g_{c_2} is sufficient. If c_1 or c_2 are non-simple cyclic paths, we can further decompose them into simple cyclic paths. Thus checking only simple cyclic paths is sufficient.

Now we consider two cyclic paths $c_1 = (p_1, p_2, ..., p_n, p_1)$ and $c_2 = (p_i, ..., p_n, p_1, ..., p_i)$ of the same cycle. Let $d_1 = \delta(p_1, p_2) \otimes ... \otimes \delta(p_{i-1}, p_i)$, $d_2 = \delta(p_i, p_{i+1}) \otimes ... \otimes \delta(p_n, p_1)$. d_1 and d_2 are continuous, and,

$$g_{c_1} = d_1 \otimes d_2$$
$$g_{c_2} = d_2 \otimes d_1.$$

Since commutativity does not hold for the $\otimes$ operator, $g_{c_1} \neq g_{c_2}$ in general. However, if the tag set $\mathcal{T}$ is totally-ordered, we have the following lemma:

LEMMA 4. *Let $\delta_1, \delta_2 \in (\mathcal{D}(\mathcal{T}) \to \mathcal{D}(\mathcal{T}))$ be two continuous functions, where $\mathcal{D}(\mathcal{T})$ is totally-ordered, and δ_1, δ_2 satisfy the liveness condition, then $\mathbf{1} \prec \delta_1 \otimes \delta_2 \Leftrightarrow \mathbf{1} \prec \delta_2 \otimes \delta_1$.*

PROOF. If $\mathbf{1} \prec \delta_1 \otimes \delta_2$, then

$$\forall\ T \in \mathcal{D}(\mathcal{T}), T \neq \mathcal{T}, \quad T \subset \delta_2(\delta_1(T)). \tag{17}$$

Suppose, contrary to this lemma, that $\mathbf{1} \not\prec \delta_2 \otimes \delta_1$, which implies $\exists$ a down set $T_0 \neq \mathcal{T}$ s.t. $\delta_1(\delta_2(T_0)) \subseteq T_0$. Since δ_2 is monotonic (due to Property 3),

$$\delta_2(\delta_1(\delta_2(T_0)) \subseteq \delta_2(T_0). \tag{18}$$

If $\delta_2(T_0) \neq \mathcal{T}$, then (18) contradicts (17). If $\delta_2(T_0) = \mathcal{T}$, then $\delta_1(\mathcal{T}) \subseteq T_0 \subset \mathcal{T}$. This contradicts the fact that δ_1 satisfies the liveness condition, i.e., $\delta_1(\mathcal{T}) = \mathcal{T}$. Therefore $\mathbf{1} \prec \delta_2 \otimes \delta_1$. □

Thus, if the tag set $\mathcal{T}$ is totally-ordered, it is sufficient to compute the gain of one cyclic path for each simple cycle to check liveness for a finite network of continuous and live actors.

9. DYNAMIC DEPENDENCIES

In the above examples, the dependencies are static (they do not change during execution of the program). This situation is excessively restrictive in practice. One simple way to model dynamically changing dependencies is to use *modal models* [Girault et al. 1999]. In a modal model, an actor is associated with a state machine, and its interface can depend on the state of the state machine. In particular, the actor could have a different causality interface in each state of the state machine. In particular, let X denote the set of states of the state machine. Then the causality interfaces are given by a function

$$\delta'_a \colon P_i \times P_o \times X \to D.$$

A simple conservative analysis would combine the causality interfaces in all the states to get a conservative causality for the actor. Specifically, for an input port $p_i \in P_i$ and an output port $p_o \in P_o$ of actor a,

$$\delta_a(p_i, p_o) = \bigoplus_{x \in X} \delta'_a(p_i, p_o, x).$$

This is conservative because causality analysis based on this interface may reveal a causality loop that is illusory, for example if the state in which the causality loop occurs is not reachable.

Depending on the model of computation and the semantics of modal models, the reachability of states in the state machine may be undecidable [Girault et al. 1999]. Hence, a more precise analysis may not always be possible. Nonetheless, it is easy to imagine circumstances in which a precise analysis could be carried out. We leave this to the imagination of the reader (Hint: The heterochronous dataflow model of computation given in [Girault et al. 1999] has such a property).

10. DETERMINING CAUSALITY INTERFACES FOR ATOMIC ACTORS

The causality analysis technique we have given determines the causality interface of a composition based on causality interfaces of the components and their interconnections. An interesting question arises: how do we determine the causality interfaces of atomic actors? If the atomic actors are language primitives, as in the synchronous langauges, then the causality interfaces of the primitives are simply part of the language definition. They would be enumerated for use by a compiler. However, in the case of coordination languages, the causality interfaces might be difficult to infer. If the atomic actors are defined in a conventional imperative language, then standard compiler techniques such as program dependence graphs (see for example [Ferrante et al. 1987; Horwitz et al. 1988; Ottenstein and Ottenstein 1984]) might be usable. However, given the Turing completeness of such languages, such analysis is likely to have to be conservative. A better alternative is probably to use an actor definition language such as Cal [Eker and Janneck 2003] or StreamIT [Thies et al. 2002] that is more amenable to such analysis.

11. CONCLUSION

We have given an interface theory that abstractly represents causality of actors and that easily composes to get causality interfaces of composite actors. The theory appears to be applicable to a wide range of actor-oriented models. We have given examples of its application to synchronous languages, discrete-event, and dataflow models.

REFERENCES

Abramsky, S., Gay, S. J., and Nagarajan, R. 1995. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, M. Broy, Ed. NATO ASI Series F. Springer-Verlag.

Agha, G. 1990. Concurrent object-oriented programming. *Communications of the ACM 33,* 9, 125–140.

Arbab, F. 2004. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14,* 3, 329–366.

ARBAB, F. 2005. Abstract behavior types : A foundation model for components and their composition. *Science of Computer Programming 55*, 3–52.

ARMSTRONG, J. R. AND GRAY, F. G. 2000. *VHDL Design Representation and Synthesis*, Second ed. Prentice-Hall.

BENVENISTE, A. AND BERRY, G. 1991. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE 79,* 9, 1270–1282.

BENVENISTE, A. AND GUERNIC, P. L. 1990. Hybrid dynamical systems theory and the signal language. *IEEE Tr. on Automatic Control 35,* 5, 525–546.

BERRY, G. 1996. *The Constructive Semantics of Pure Esterel.* Book Draft.

BERRY, G. AND GONTHIER, G. 1992. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

BROY, M. AND STEFANESCU, G. 2001. The algebra of stream processing functions. *Theoretical Computer Science 258*, 99–129.

BUCK, J. T. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Technical Memorandum UCB/ERL 93/69, EECS Department, University of California, Berkeley.

BUCK, J. T., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development" 4,* 155–182.

CASSANDRAS, C. G. 1993. *Discrete Event Systems, Modeling and Performance Analysis.* Irwin.

CHAKRABARTI, A., DE ALFARO, L., AND HENZINGER, T. A. 2003. Resource interfaces. In *EMSOFT*, R. Alur and I. Lee, Eds. Vol. LNCS 2855. Springer, Philadelphia, PA, 117–133.

DAVEY, B. A. AND PRIESTLY, H. A. 1990. *Introduction to Lattices and Order.* Cambridge University Press.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT).* Vol. LNCS 2211. Springer-Verlag, Lake Tahoe, CA, 148–165.

DE KOCK, E. A., ESSINK, G., SMITS, W. J. M., VAN DER WOLF, P., BRUNEL, J.-Y., KRUIJTZER, W., LIEVERSE, P., AND VISSERS, K. A. 2000. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference (DAC'00).* Los Angeles, CA, 402–405.

DENNIS, J. B. 1974. First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science.

EDWARDS, S. A. AND LEE, E. A. 2003. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming 48,* 1.

EKER, J. AND JANNECK, J. W. 2003. CAL language report: Specification of the CAL actor language. Tech. Rep. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA. December 1.

EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE 91,* 1, 127–144.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Transactions On Programming Languages And Systems 9,* 3, 319–349.

GIRAULT, A., LEE, B., AND LEE, E. A. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems 18,* 6, 742–760.

GÖESSLER, G. AND SANGIOVANNI-VINCENTELLI, A. 2002. Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT).* Springer-Verlag, Grenoble, France.

GÖESSLER, G. AND SIFAKIS, J. 2005. Composition for component-based modeling. *Science of Computer Programming 55.*

HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. 1991. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE 79,* 9, 1305–1319.

HEWITT, C. 1977. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence 8,* 3, 323–363.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1988. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.* Vol. SIGPLAN Notices 23(7). Atlanta, Georgia, 35–46.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co.

KAHN, G. AND MACQUEEN, D. B. 1977. Coroutines and networks of parallel processes. In *Information Processing*, B. Gilchrist, Ed. North-Holland Publishing Co.

KEUTZER, K., MALIK, S., NEWTON, A. R., RABAEY, J., AND SANGIOVANNI-VINCENTELLI, A. 2000. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19,* 12.

LEE, E. A. 1999. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering 7*, 25–45.

LEE, E. A. 2002. Embedded software. In *Advances in Computers*, M. Zelkowitz, Ed. Vol. 56. Academic Press.

LEE, E. A. 2003. 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.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous data flow. *Proceedings of the IEEE 75,* 9 (September), 1235–1245.

LEE, E. A. AND NEUENDORFFER, S. 2004. Classes and subclasses in actor-oriented design. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*. San Diego, CA, USA.

LEE, E. A., NEUENDORFFER, S., AND WIRTHLIN, M. J. 2003. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers 12,* 3, 231–260.

LEE, E. A. AND PARKS, T. M. 1995. Dataflow process networks. *Proceedings of the IEEE 83,* 5, 773–801.

LEE, E. A. AND SANGIOVANNI-VINCENTELLI, A. 1998. A framework for comparing models of computation. *IEEE Transactions on CAD 17,* 12.

LEE, E. A. AND XIONG, Y. 2004. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing Journal 16,* 3, 210–237.

LEE, E. A. AND ZHENG, H. 2005. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, M. Morari and L. Thiele, Eds. Vol. LNCS 3414. Springer-Verlag, Zurich, Switzerland, pp. 25–53.

LEE, E. A., ZHENG, H., AND ZHOU, Y. 2005. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT), Satellite to CONCUR*. San Francisco, CA.

LIU, X. 2005. Semantic foundation of the tagged signal model. Ph.D. Thesis Technical Memorandum UCB/EECS-2005-31, EECS Department, University of California, Berkeley. December 20.

LIU, X. AND LEE, E. A. 2006. CPO semantics of timed interactive actor networks. Tech. Rep. UCB/EECS-2006-67, EECS Department, University of California, Berkeley. May 18.

MATTHEWS, S. G. 1995. An extensional treatment of lazy data flow deadlock. *Theoretical Computer Science 151,* 1, 195–205.

OTTENSTEIN, K. J. AND OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. *SIGPLAN Notices 19,* 5, 177–184.

PAPADOPOULOS, G. A., STAVROU, A., AND PAPAPETROU, O. 2006. An implementation framework for software architectures based on the coordination paradigm. *Science of Computer Programming 60,* 1, 27–67.

RUTTEN, J. J. M. M. 2005. A coinductive calculus of streams. *Mathematical Structures in Computer Science 15,* 1, 93–147.

SCHNEIDER, K., BRANDT, J., AND SCHUELE, T. 2004. Causality analysis of synchronous programs with delayed actions. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. Washington DC, USA.

TALCOTT, C. L. 1996. Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*. Vol. LNCS 2304. Springer-Verlag, Grenoble, France.

TILLER, M. M. 2001. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers.

WADGE, W. 1981. An extensional treatment of dataflow deadlock. *Theoretical Computer Science 13,* 1, 3–15.

WEGNER, P., ARBAB, F., GOLDIN, D., MCBURNEY, P., LUCK, M., AND ROBERSON, D. 2005. The role of agent interaction in models of computation (panel summary). In *Workshop on Foundations of Interactive Computation*. Edinburgh.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA.

WINTER, M., GENSSLER, T., CHRISTOPH, A., NIERSTRASZ, O., DUCASSE, S., WUYTS, R., ARÉVALO, G., MÜLLER, P., STICH, C., AND SCHÖNHAGE, B. 2002. Components for embedded software – the PECOS approach. In *Second International Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP)*. Málaga, Spain.

XIONG, Y. 2002. An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720. May 1.

YATES, R. K. 1993. Networks of real-time processes. In *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, E. Best, Ed. Vol. LNCS 715. Springer-Verlag.