Causality Interfaces and Compositional Causality Analysis [1]

Edward A. Lee, Haiyang Zheng, Ye Zhou

{*eal, hyzheng, zhouye*}*@eecs.berkeley.edu*
Center for Hybrid and Embedded Software Systems (CHESS)
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

Abstract

In this paper, 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 transfer of control. Examples of such models of computation include the classical actor model, synchronous languages, dataflow models, and discrete-event models. 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 discrete-event models for uniqueness of behaviors, synchronous models for causality loops, and dataflow models for schedulability.

Key words: Actors, causality, dataflow, discrete-event models, synchronous languages, compositional analysis, interfaces, behavioral types

1 Introduction

Although prevailing component architecture techniques in software are object oriented, a number of researchers have been advocating a family of complementary

[1] 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 and the following companies: Agilent, General Motors, Hewlett-Packard, Honeywell, Infineon, Samsung, and Toyota.

approaches that we collectively call *actor oriented* [35]. In practice, the components of object-oriented design interact principally through transfer of control (method calls). The components are passive, and things get done to them, much like physical "objects" from which the name arises. [2] "Actors" are concurrent, in charge of their own actions. Their environment (which can include other actors) provides them with data, and they react and possibly provide the environment with additional data. 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" [2,28] 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. In fact, several other communities also use similar ways of defining components. In the synchronous/reactive languages [7], for example, components react at ticks of a global clock, rather than reacting when other components invoke their methods. In the synchronous language Esterel [11], components exchange data through variables whose values are determined by solving fixed point equations. The Lustre [27] and Signal [9] languages have a more dataflow flavor, where components consume inputs and produce outputs. Asynchronous dataflow models are also actor-oriented in our sense, including both Kahn-MacQueen process networks [31], where each component has its own thread of control, and Dennis-style dataflow [19], where components (also called "actors" in the original literature) "fire" in response to the availability of input data.

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 [37,34]. Discrete-event (DE) systems, for example, are commonly used in circuit design and in modeling and design of communication networks [15,5]. 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 [48], 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 [30,13,44], discrete-event formalisms [52,33], and semantics for continuous time models [41]. A few formalisms are rich enough to embrace a significant variety of actor-oriented models, including inter-

[2] So called "active objects" add to the basic object-oriented model threads, but as a component technology, active objects are primitive compared to the actor-oriented techniques we describe.

action categories [1], behavioral types [40,4], interaction semantics [46], and the tagged-signal model [39]. Some software frameworks also embrace a multiplicity of actor-oriented component architectures, including abstract behavior types (complementing the object-oriented abstract data types) [4], Reo [3], Ptolemy II [22], PECOS [50], and Metropolis [25]. Finally, a number of researchers have argued strongly for separation between the semantics of functionality (what is computed) from that of interaction between components [14,32,26,49].

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 [51] 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 [36] with inheritance mechanisms.

A very general way of talking about interfaces for actor-oriented designs is in the notion of *interface theories* [18]. Some concrete applications of such theories are given in resource interfaces [16] and behavioral type systems [40].

This paper concentrates on a particular family of interface theories that capture *causality* properties of actor-oriented designs. Causality properties reflect in the interface the dependence that particular outputs have on particular inputs. The work here is closest in spirit to the component interfaces of Broy in [12], where causality properties of stream functions are formalized. In this paper, however, we follow the spirit of de Alfaro and Henzinger's interface theories [18] to create a rather specialized theory (of causality only) that is orthogonal to other semantic properties. So, whereas the work of Broy is tightly coupled to stream semantics, ours here can be applied to streams as well as to other concurrent semantics such as that of the synchronous languages, discrete-event models, and continuous-time models.

As in object-oriented design, *composition* and *abstraction* are two central concepts in actor-oriented design. Actors can be composed to form new actors, which are called *composite actors*. Actors that are not composite actors are called *atomic actors*; they may be predefined (as is typical, for example, in the synchronous languages), or they may be user-defined, as is typical in coordination languages [3,43,17]. In a compositional formalism, a composite actor is itself an actor, and hence its interface(s) must be those of an actor. A major focus of this paper is on how causality properties of composite actors can be determined from their component actors.

Following [18] 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 [51]. 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 [40].

In this paper, we consider a particular kind of behavioral interface that we call a *causality interface*. 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 stream-oriented dataflow models, it can be used to analyze compositions of actors for deadlock or livelock [13,38]. In discrete-event models, it can be used to ensure deterministic processing of simultaneous events [33,52]. 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 [45,10,20]. In all three cases, the causality properties of components determine existence and uniqueness of the behavior of a particular composition.

2 Causality

We begin by reviewing 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 define causality interfaces and show how they can be used in the models of computation of interest.

2.1 The Tagged Signal Model

The tagged-signal model [39] 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 [4], interaction categories [1], and interaction semantics [46]. As with all three of these, the tagged signal model seeks to model a variety of interaction styles between concurrent components. Our notation here is adapted from [8].

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.

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."

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 $\sigma(p) \in S$.

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

$$a \subset [P_a \to S],$$

where $[X \to Y]$ denotes the set of (total) functions with domain X and range contained by Y. 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.

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.

In many actor-oriented 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. 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

$$F_a \colon [P_i \to S] \rightharpoonup [P_o \to S]. \tag{1}$$

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*. A composition is *determinate* if it is functional.

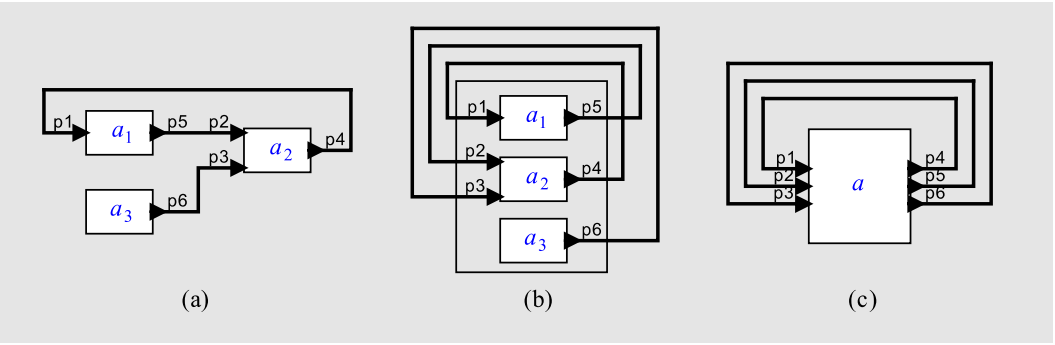

Fig. 1. A composition of three actors and its interpretation as a feedback system.

A key question in many actor-oriented 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.

2.2 Syntax

Actor-oriented languages can be either self-contained programming languages (e.g. Esterel, Lustre) or coordination languages (e.g. 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 textual syntaxes. Ports and connectors are syntactically represented in these languages by variable names. Using 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.

2.3 Semantics

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 block 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 (1) giving the behaviors of a. Then the behavior of the feedback composition in figure 1(c) is a function

$$f: \{p1, p2, p3\} \rightarrow S$$

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 [30,13,38], 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) [30]. 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, continuousness 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 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 [38]. The causality interfaces we define here provide sufficient conditions that will often determine whether a composition yields a deadlock condition.

In synchronous/reactive models (as in the synchronous languages Esterel and Lustre), 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 [45,10,20]. 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 causal-

ity loop exists [20]. The required information is exactly what is represented by our causality interfaces.

In discrete-event models, it is customary to define semantics somewhat differently, by defining a metric space on the set S of signals [52,33]. We are interested in existence and uniqueness of the behavior of a feedback composition like that in figure 1(c). It is sufficient for the composite actor a to be functional and for the function F_a to be a contraction map in the metric space. Our causality interfaces help determine whether a composition yields a contraction map.

In all three cases (dataflow, synchronous, and discrete-events), the difficulty that can arise in figure 1(c) is a dependency cycle that is not inductive (a causality loop), and our causality interfaces are specifically intended to help to determine when such causality loops exist.

2.4 Causality Interfaces

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

$$\delta\colon P_i \times P_o \to D, \tag{2}$$

where D is an ordered set with two binary operations $\otimes$ and $\oplus$ that satisfy the axioms given below. First, we require that the operators $\oplus$ and $\otimes$ be associative,

$$\forall d_1, d_2, d_3 \in D, \quad (d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3),$$
$$\forall d_1, d_2, d_3 \in D, \quad (d_1 \otimes d_2) \otimes d_3 = d_1 \otimes (d_2 \otimes d_3),$$

commutative,

$$\forall d_1, d_2 \in D, \quad d_1 \oplus d_2 = d_2 \oplus d_1,$$
$$\forall d_1, d_2 \in D, \quad d_1 \otimes d_2 = d_2 \otimes d_1,$$

and distributive,

$$\forall d_1, d_2, d_3 \in D, \quad d_1 \otimes (d_2 \oplus d_3) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3).$$

In addition, we require an additive and multiplicative identity, called $\mathbf{0}$ and $\mathbf{1}$, respectively. That is

$$\exists\, \mathbf{0} \in D \text{ such that} \qquad \forall\, d \in D, \quad d \oplus \mathbf{0} = d$$
$$\exists\, \mathbf{1} \in D \text{ such that} \qquad \forall\, d \in D, \quad d \otimes \mathbf{1} = d$$
$$\forall\, d \in D, \quad d \otimes \mathbf{0} = \mathbf{0}$$
$$\forall\, d \in D, \quad d \oplus d = d$$

The ordering relation $<$ on the set D is a total order, meaning, as usual,

$$\forall d \in D, \qquad d \not< d$$
$$\forall d_1, d_2 \in D, \qquad d_1 \not< d_2 \text{ and } d_2 \not< d_1 \;\Rightarrow\; d_1 = d_2$$
$$\forall d_1, d_2, d_3 \in D, \quad d_1 < d_2 \text{ and } d_2 < d_3 \;\Rightarrow\; d_1 < d_3.$$

As usual, the $=$ relation is ordinary equality, $d \neq d$ is shorthand for an assertion that $d < d$ is false, and $d_1 \leq d_2$ is shorthand for $(d_1 < d_2) \vee (d_1 = d_2)$.

Finally, the key axioms of D relate the operators and the order as follows.

$$\forall d_1, d_2 \in D, \quad d_1 \leq (d_1 \otimes d_2)$$

$$\forall d_1, d_2 \in D, \quad (d_1 \oplus d_2) \leq d_1$$

The elements of D are called *dependencies*, and $\delta(p_1, p_2)$ denotes the dependency that port p_2 has on p_1.

We will be interested in two specific cases. In the *boolean dependency* case, $D = \{\text{true}, \text{false}\}$, $\oplus$ is logical and, $\otimes$ is logical or, false $<$ true, $\mathbf{0} =$ true, and $\mathbf{1} =$ false. With these definitions, all of the above axioms are satisfied.

In the *weighted dependency* case, $D = \mathbb{R}_+ \cup \{\infty\}$, the non-negative real numbers plus infinity, $\oplus$ is the $\min$ function, $\otimes$ is addition, $<$ is ordinary numerical ordering, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$[3]. Again, with these definitions, all of the above axioms are satisfied.

2.5 *Composition Analysis*

Given a set A of actors, a set C of connectors, and the causality interfaces for the actors, we can determine the causality interface of the composition. Consider the example in figure 1. To determine the causality interface of the composite actor a, we need to determine the function

$$\delta_a \colon \{p1, p2, p3\} \times \{p4, p5, p6\} \to D.$$

To do this, we form a graph of ports, and observe that the paths between ports traverse both actors and connectors. To determine the value of $\delta_a(p1, p4)$, for example, we need to consider all the paths between $p1$ and $p4$. A path consists of links provided by connectors and actors. The dependencies of the links and actors are combined using the $\otimes$ operator for series compositions and the $\oplus$ operator for parallel compositions.

Consider the example in figure 1. From figure 1(b) we can immediately conclude that

$$\delta_a(p1, p5) = \delta_1(p1, p5),$$

where δ_1 is the causality interface of actor a_1. Where there are no dependencies, the causality interface yields the additive identity, so

$$\delta_a(p1, p4) = \mathbf{0}.$$

Suppose for example that figure 1 represents a synchronous program in, say, Lustre. For synchronous programs, we use boolean dependencies, where $D = \{\text{true}, \text{false}\}$. For ports p and p', $\delta(p, p') =$ false is interpreted to mean that at a tick of the clock, the value at p' depends on the value at p. A dependency value true is interpreted to mean that there is no such dependency. Suppose we annotate the

[3] In this case, the dependency set is also called a min-plus algebra [6].

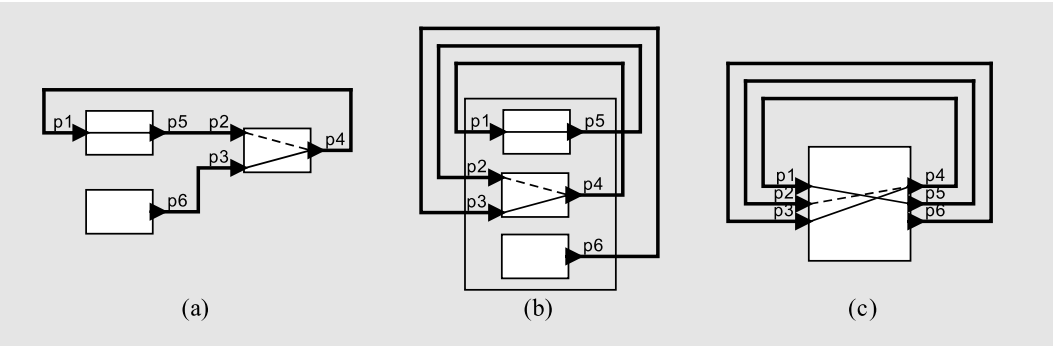

Fig. 2. The composition of figure 1 annotated with boolean dependencies. If the dependencies in (a) are given, the dependencies in (c) can be systematically inferred.

diagram as shown in figure 2, where a solid line denotes a dependency with value false, a dashed line denotes a dependency with value true, and no line denotes a dependency with value $\mathbf{0} =$ true. Hence, for instance, the dashed line between $p2$ and $p4$ might mean that there is a "pre" operator, which decouples the value at $p4$ from that at $p2$ in each tick of the clock. The dashed line, therefore, is equivalent to no line. Then the causality of the interface of the composite actor a becomes as shown in 2(c).

Connectors have particularly simple causality interfaces. A connector $c \in C$ linking output port p and input port p' yield the dependency

$$\delta(p, p') = \mathbf{1},$$

the multiplicative identity.

We can now analyze the feedback composition for causality loops. A causality loop exists if the dependency between any port and itself is false. Consider for example port $p4$. Using the fact that connectors yield the multiplicative identity for dependencies, we can write this as

$$\begin{aligned} \delta(p4, p4) &= \delta_1(p1, p5) \otimes \delta_2(p2, p4) \\ &= \text{false} \otimes \text{true} \\ &= \text{true} \end{aligned}$$

where we have used the fact that in boolean dependencies, $\otimes$ is logical or. We can similarly check every port to determine that this composition has no causality loops.

Consider the same diagram, but now representing a discrete-event model. For discrete-event models, we use a weighted dependency model, where $D = \mathbb{R}_+ \cup \{\infty\}$, the non-negative real numbers plus infinity, $\oplus$ is the min function, $\otimes$ is addition, $<$ is ordinary numerical ordering, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$. Each dependency represents a time delay. In figure 2, we now interpret solid lines to represent a delay of zero, so for example

$$\delta_1(p1, p5) = 0.$$

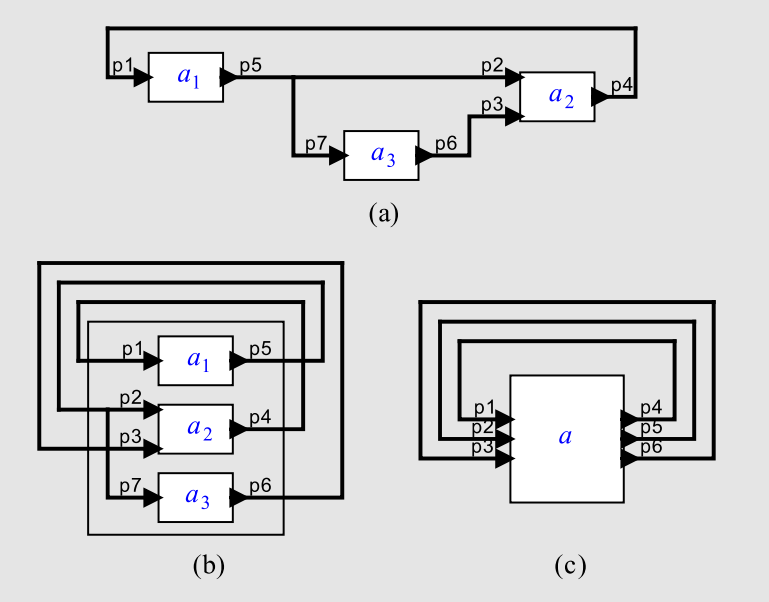

Fig. 3. A more complicated composition.

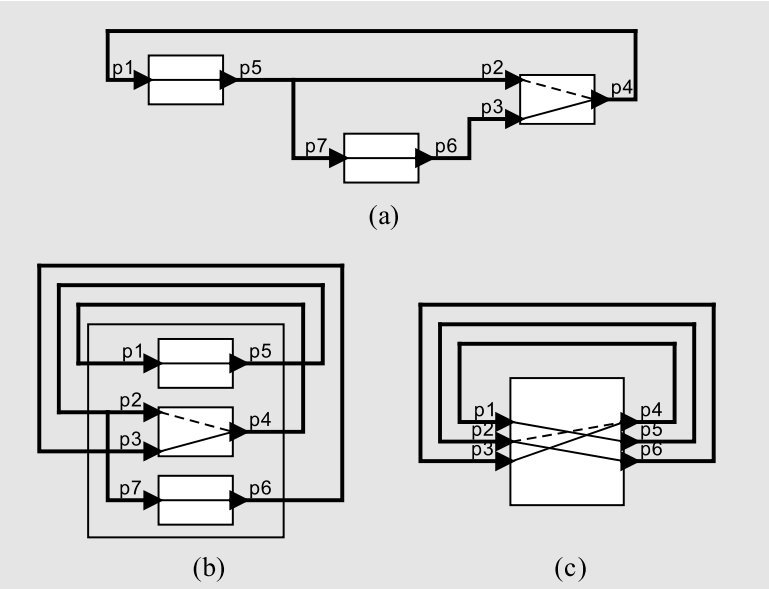

Fig. 4. Dependencies for the composition in figure 3. If the dependencies in (a) are given, the dependencies in (c) can be systematically inferred.

A causality loop occurs if the dependency from any port to itself is zero. A similar analysis again yields the fact that the model in figure 2 has no causality loops.

Consider the slightly more complicated composition shown in figure 3. This example has a more complicated link, joining ports $p5$, $p2$, and $p7$. It also has parallel paths. Suppose that the dependencies are as shown in figure 4. Then we can perform the composition analysis to determine that port $p4$ has a causality loop.

In particular, assuming a synchronous model,

$$\begin{aligned}\delta(p4, p4) &= \delta_1(p1, p5) \otimes (\delta_2(p2, p4) \oplus (\delta_3(p7, p6) \otimes \delta_2(p3, p4))) \\ &= \text{false} \otimes (\text{true} \oplus (\text{false} \otimes \text{false})) \\ &= \text{false}.\end{aligned}$$

Just as the $\otimes$ operator is used to compose causality of chained links, the $\oplus$ operator is used to compose causality of parallel links. The same analysis would reveal a causality loop in a discrete-event model.

2.6 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* [24]. 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

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

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 $p_o \in P_o$,

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

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 [24]. 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.

2.7 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 [23,29,42]) 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 [21] or StreamIT [47] that is more amenable to such analysis.

3 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 and to discrete-event models, and have suggested how to apply it to dataflow models.

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

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

[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 edition, 2000.

[6] F. Baccelli, G. Cohen, G. J. Olster, and J. P. Quadrat. *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York, 1992.

[7] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[8] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. 2003.

[9] A. Benveniste and P. L. Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Tr. on Automatic Control*, 35(5):525–546, 1990.

[10] G. Berry. *The Constructive Semantics of Pure Esterel*. Book Draft, 1996.

[11] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[12] M. Broy. Advanced component interface specification. Technical Report Unmarked, Technical University of Munich, 1995.

[13] M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258:99–129, 2001.

[14] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, 1994.

[15] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[16] A. Chakrabarti, L. d. Alfaro, and T. A. Henzinger. Resource interfaces. In R. Alur and I. Lee, editors, *EMSOFT*, volume LNCS 2855, pages 117–133, Philadelphia, PA, 2003. Springer.

[17] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2), 1996.

[18] L. deAlfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, pages 148–165, Lake Tahoe, CA, 2001. Springer-Verlag.

[19] J. B. Dennis. First version data flow procedure language. Technical Report MAC TM61, MIT Laboratory for Computer Science, 1974.

[20] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1), 2003.

[21] J. Eker and J. W. Janneck. Cal language report: Specification of the cal actor language. Technical Report Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA, December 1 2003.

[22] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 91(2), 2003.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions On Programming Languages And Systems*, 9(3):319–349, 1987.

[24] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), 1999.

[25] G. Goessler and A. Sangiovanni-Vincentelli. Compositional modeling in metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Grenoble, France, 2002. Springer-Verlag.

[26] G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55, 2005.

[27] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[28] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323363, 1977.

[29] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume SIGPLAN Notices 23(7), pages 35–46, Atlanta, Georgia, 1988.

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

[31] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*. North-Holland Publishing Co., 1977.

[32] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), 2000.

[33] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[34] E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.

[35] 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.

[36] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, CA, USA, 2004.

[37] 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.

[38] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[39] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.

[40] E. A. Lee and Y. Xiong. A behavioral type system and its application in ptolemy ii. *Formal Aspects of Computing Journal*, special issue on Semantic Foundations of Engineering Design Languages, 2004.

[41] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS, Zurich, Switzerland, 2005. Springer.

[42] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notices*, 19(5):177–184, 1984.

[43] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers - The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.

[44] J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[45] 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.

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

[47] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *11th International Conference on Compiler Construction*, volume LNCS 2304, Grenoble, France, 2002. Springer-Verlag.

[48] M. M. Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[49] P. Wegner, F. Arbab, D. Goldin, P. McBurney, M. Luck, and D. Roberson. The role of agent interaction in models of computation (panel summary). In *Workshop on Foundations of Interactive Computation*, Edinburgh, 2005.

[50] M. Winter, T. Genssler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arvalo, P. Mller, C. Stich, and B. Schnhage. 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)*, Mlaga, Spain, 2002.

[51] Y. Xiong. 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 2002.

[52] 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.